ԵՐԵՎԱՆԻ ՊԵՏԱԿԱՆ ՀԱՄԱԼՍԱՐԱՆ

Ս.Գ. ՍԱՐԳՍՅԱՆ, Ա.Ս. ՀՈՎԱԿԻՄՅԱՆ,
Կ.Ս. ԴԱՐԲԻՆՅԱՆ,
Ս.Ա. ԱՎԵՏԻՍՅԱՆ, Ն.Հ. ԻՍՊԻՐՅԱՆ

ՏՎՅԱԼՆԵՐԻ ԿԱՌՈՒՑՎԱԾՔՆԵՐ

ՈՒՍՈՒՄՆԱԿԱՆ ՁԵՌՆԱՐԿ

ԵՐԵՎԱՆ
ԵՊՀ ՀՐԱՏԱՐԱԿՉՈՒԹՅՈՒՆ
2010

ՀՏԴ 51 : 681.3/ 5 : 004 (07)
ԳՄԴ 22.1 + 32.973 g7
S 900

Հրատարակության է երաշխավորել
ԵՊՀ ինֆորմատիկայի և կիրառական
մաթեմատիկայի ֆակուլտետի խորհուրդը

Ս.Գ. ՍԱՐԳՍՅԱՆ, Ա.Ս. ՀՈՎԱԿԻՄՅԱՆ,
Կ.Ս. ԴԱՐԲԻՆՅԱՆ և ուրիշներ

S 900 Տվյալների կառուցվածքներ: Ուսումնական ձեռնարկ: – Եր.: ԵՊՀ հրատ., 2010 թ., 188 էջ:

Ձեռնարկում նկարագրված են տվյալների հիմնական աբստրակտ տիպերը՝ գծային ցուցակ, պահունակ, հերթ, ծառեր, աղյուսակներ և դրանք իրականացնող կառուցվածքները: Բերված են այդ կառուցվածքների կիրառությունների օրինակներ. ինչպես նաև լուծված խնդիրներ:

Նախատեսված է ինֆորմատիկայի և կիրառական մաթեմատիկայի ֆակուլտետի ուսանողների համար: Կարող է օգտակար լինել նաև ծրագրավորման խնդիրներում տվյալների տարբեր կառուցվածքներ օգտագործողներին:

ՀՏԴ 51 : 681.3/ 5 : 004 (07)
ԳՄԴ 22.1 + 32.973 g7

ISBN 978-5-8084-1270-5

ՆԱԽԱԲԱՆ

Ծրագրերի նախագծման և իրականացման համար կարևորվում է տարբեր տիպի տվյալների օգտագործումը և տվյալների համապատասխան մոդելի ընտրությունը: Դրանով պայմանավորվում է մշակված ծրագրի աշխատանքի արդյունավետությունը:

Ձեռնարկում նկարագրված է տվյալների աբստրակտ տիպերի հասկացությունը: Դիտարկված են գծային ցուցակ, պահունակ, հերթ, ծառեր, աղյուսակներ աբստրակտ տիպերը և դրանք իրականացնող հիմնական կառուցվածքները՝ զանգվածներ, միակապ, երկկապ, ցիկլիկ, գծային և ոչ գծային ցուցակներ: Բերված են այդ կառուցվածքների կիրառությունների օրինակներ: Դիտարկված է նաև տվյալների նշված աբստրակտ տիպերի իրականացումը C++ լեզվի շաբլոնների STL ստանդարտ գրադարանում:

Յուրաքանչյուր թեմայի վերաբերյալ առաջարկված են խնդիրներ: Ձեռնարկը պարունակում է նաև լուծված խնդիրներ, որոնք նշված են *-ով:

Ձեռնարկը նախատեսված է ինֆորմատիկայի և կիրառական մաթեմատիկայի ֆակուլտետի ուսանողների համար: Այն կարող է օգտակար լինել նաև ծրագրավորման խնդիրներում տվյալների տարբեր կառուցվածքներ օգտագործողների համար:

ՆԵՐԱԾՈՒԹՅՈՒՆ

Ժամանակակից ծրագրային համակարգերը նախագծվում են մոդուլային սկզբունքով: Մոդուլի իրականացման եղանակը սովորաբար թաքնված է օգտագործողից, իսկ համագործակցությունը մոդուլի հետ կատարվում է ինտերֆեյսի՝ մոդուլին փոխանցվող և մոդուլից ստացվող պարամետրերի միջոցով: Այս մոտեցումը համարվում է ֆունկցիոնալ աբստրակցիայի իրականացման եղանակ: Ծրագրային համակարգեր նախագծելիս առկա է նաև տվյալների աբստրակցիան: Ծրագրողին սովորաբար հայտնի են միայն այն գործողությունները, որոնք կարելի է կատարել այդ տվյալների նկատմամբ: Օգտվողին ոչ միշտ են հասանելի ինչպես տվյալների ներկայացման, այնպես էլ գործողությունների իրականացման ձևերը: Տվյալների կազմակերպման այդ սկզբունքի հետ կապված է տվյալների աբստրակտ տիպ հասկացությունը (Abstract Data Type, ADT): ADT ասելով հասկանում ենք տվյալների մաթեմատիկական մոդել այն գործողությունների համախմբով, որոնք կատարվում են մոդելում ընդգրկված տվյալների նկատմամբ:

Օրինակ, տվյալների աբստրակտ տիպեր են հանդիսանում.

1. Գծային ցուցակը, որի հետ կատարվող գործողություններն են տարրի ավելացումը, տարրի հեռացումը, ցուցակների կցումը և այլն:
2. Որոշակի տիպի տարրերի բազմությունը՝ տարրի պատկանելություն ստուգման, բազմությունների միավորման, հատման և այլ գործողությունների համախմբով:
3. Ծառը՝ գագաթների մշակման գործողությունների համախմբով:

Աբստրակտ տիպի տվյալների ներկայացման ձևերը, որոնք աջակցվում են ծրագրավորման լեզվի միջոցներով, անվանում են տվյալների կառուցվածքներ: Տվյալների կառուցվածքով որոշվում է ADT-ի գործողությունների իրականացումը:

ԳԼՈՒԽ 1.

ՏՎՅԱԼՆԵՐԻ ԱԲՍՏՐԱԿՏ ՏԻՊԵՐ

Տվյալների աբստրակտ տիպը (ADT) դիտարկենք որպես մաթեմատիկական մոդել, այդ մոդելում որոշված որոշակի գործողությունների բազմությամբ: ADT-ն կոնկրետ ալգորիթմական լեզվում իրականացնելիս օգտագործվում են համապատասխան տվյալների կառուցվածքներ:

Ծրագրի նախագծման և մշակման համար կարևոր է տվյալների աբստրակցիան: Աբստրակցիան որոշում է տվյալների կառուցվածքը և տվյալների հետ կատարվող գործողությունները: Աբստրակտ տիպի տվյալները կազմում են ծրագրողի կողմից սահմանվող տիպ, որի գործողությունները ցույց են տալիս, թե ինչպես կարելի է աշխատել այդ տվյալների հետ:

ADT-ն նկարագրվում է որոշակի ձևաչափով, որը պարունակում է ADT-ի անունով վերնագիր, տվյալների տիպի նկարագիր և գործողությունների ցուցակ:

Յուրաքանչյուր գործողության համար որոշվում են՝

- Մուտքային արժեքներ (input),
- Նախապայմաններ (preconditions), որոնց բավարարում են տվյալները նախքան նրանց հետ գործողություն կատարելը,
- Պրոցես (process), որը նկարագրում է կատարվող գործողությունը,
- Ելքային արժեքներ (output),
- Վերջնապայման (postconditions), որոնց բավարարում են տվյալները գործողությունը կատարելուց հետո:

ADT-ների մեծ մասն ունեն սկզբնարժեքավորման գործողություններ (initializer), որոնք տվյալներին տալիս են սկզբնական արժեքներ: C++ լեզվում տվյալների սկզբնարժեքավորումը իրականացվում է դասի կոնստրուկտորի (constructor) միջոցով: ADT-ի նկարագրությունը տրվում է հետևյալ ձևով:

ADT անուն

Տվյալներ

Տվյալների կառուցվածքի նկարագիր

Գործողություններ

- Կոնստրուկտոր

  Սկզբնական արժեքներ – տվյալներ, որոնք օգտագործվում են օբյեկտը սկզբնարժեքավորելիս

  Պրոցես – օբյեկտի սկզբնարժեքավորում

- Գործողություն – 1

  Մուտք – մուտքային տվյալներ

  Նախապայման – գործողության իրականացման համար անհրաժեշտ պայմանների հավաքածու

  Պրոցես – տվյալների հետ կատարվող գործողություն

  Ելք – վերադարձվող տվյալներ

  Վերջնապայման – պայմանների հավաքածու, որոնք տեղի ունեն գործողությունը կատարելուց հետո

- Գործողություն – 2

.....

- Գործողություն – n

Վերջ *ADT* անուն

Օրինակ 1.

Dice աբստրակտ տիպով նկարագրենք N հատ զառերի նետումը:

ADT Dice

Տվյալներ

Յուրաքանչյուր նետման դեպքում զառերի N քանակը (N>=1, ամբողջ թիվ է): Վերջին նետման ժամանակ բոլոր զառերի միավորների գումարը (N-ից 6*N միջակայքի ամբողջ թիվ է): Ցուցակ, որը պարունակում է նետման յուրաքանչյուր զառի միավորները: Ցուցակի յուրաքանչյուր տարրի արժեքը գտնվում է 1-ից 6 միջակայքում:

Գործողություններ

Կոնստրուկտոր

Սկզբնական արժեքներ - նետվող զառերի քանակը

Պրոցես – յուրաքանչյուր նետման զառերի քանակի սկզբնարժեքավորում

Toss

Մուտք -չկա

Նախապայման - չկա

Պրոցես - զառերի նետում և բացված միավորների ընդհանուր գումարի հաշվարկ

Ելք - չկա

Վերջնապայման - ամբողջ գումարը պարունակում է բոլոր զառերում բացված միավորների ընդհանուր գումարը, իսկ ցուցակում գտնվում են յուրաքանչյուր զառի միավորները

Total

Մուտք - չկա

Նախապայման -չկա

Պրոցես - գտնում է վերջին նետման ժամանակ բացված միավորների գումարը

Ելք - վերջին նետման միավորների գումարը

Վերջնապայման - չկա

DisplayToss

Մուտք - չկա

Նախապայման - չկա

Պրոցես - տպում է վերջին նետման ժամանակ յուրաքանչյուր զառի բացված միավորների ցուցակը

Ելք - չկա

Վերջնապայման - չկա

Վերջ *ADT Dice*

Օրինակ 2.

Շրջանագիծ նկարագրելու համար մշակենք տվյալների աբստրակտ տիպ, որը պարունակում է շրջանի մակերեսը (Area) և շրջանագծի երկարությունը (Circumference) որոշող գործողություններ:

ADT Circle

Տվյալներ

Ոչ բացասական իրական թիվ, որով տրվում է շրջանագծի շառավիղը:

Գործողություններ

Կոնստրուկտոր

Սկզբնական արժեքներ - շրջանագծի շառավիղ

Պրոցես - արժեքավորվում է շառավիղը

Area
    Մուտք - չկա
    Նախապայման - չկա
    Պրոցես - հաշվարկվում է շրջանի մակերեսը
    Ելք - շրջանի մակերեսը
    Վերջնապայման - չկա
Circumference
    Մուտք - չկա
    Նախապայման - չկա
    Պրոցես - հաշվարկվում է շրջանագծի երկարությունը
    Ելք - շրջանագծի երկարությունը
    Վերջնապայման - չկա
Վերջ *ADT Circle*

Տվյալների աբստրակտ տիպը C++ լեզվում իրականացվում է դասի միջոցով: Դասը կազմված է անդամներից (members), որոնք պարունակում են տվյալներ և գործողություններ: Գործողությունները կոչվում են նաև մեթոդներ (methods): Տվյալ դասի տիպի փոփոխականը կոչվում է այդ դասի օբյեկտ: Դասը պարունակում է բաց (public), փակ (private), պաշտպանված (protected) մասեր (նկ. 1.1): public մասը նկարագրում է այնպիսի ինտերֆեյս, որն օգտվողին թույլատրում է օգտագործել օբյեկտը և գործողությունները: Դասի private և protected մասերը պարունակում են տվյալներ և ներքին գործողություններ, որոնք օգտագործվում են միայն դասի մեթոդների կողմից և առանձնացված են արտաքին միջավայրից:

ԴԱՍ

private:
անդամ տվյալներ
ներքին գործողություններ

public:
կոնստրուկտոր
դեստրուկտոր
գործողություն 1
.....
գործողություն N

Նկ 1.1 Դասի կառուցվածք

ADT Circle–ի համար դասը կունենա հետևյալ տեսքը՝

private:
Radius

public:
Circle()
~Circle()
Area()
Circumference()

Նկ. 1.2 Circle դաս

ԽՆԴԻՐՆԵՐ

1. Մշակել և իրականացնել ADT՝ օրացույց մոդելավորելու համար: Այն պետք է ներկայացնի օրը, ամիսը, տարին ամբողջ թվերի տեսքով: Նախատեսել ամսաթվի մեկով ավելացման գործողությունը և ամսաթվի պատկերման գործողությունը՝ ամիսը տալով թվով կամ անունով:
2. Մշակել և իրականացնել ADT՝ տարրերի վերջավոր բազմություն մոդելավորելու համար: Նախատեսել բազմությունների հավասարության ստուգման, միավորման, հատման գործողությունները: ADT–ն իրականացնել զանգվածի օգտագործմամբ:
3. Մշակել և իրականացնել ADT՝ սիմվոլային տող մոդելավորելու համար: Նախատեսել տողի երկարության հաշվման, տողերի կցման գործողությունները: Սիմվոլային տողը ներկայացնել զանգվածի օգնությամբ:
4. Մշակել և իրականացնել ADT՝ բազմանդամ ներկայացնելու համար: Նախատեսել բազմանդամի կարգի, տրված աստիճանով միանդամի գործակցի վերադարձի, տրված կետում բազմանդամի արժեքի հաշվման գործողությունները: Բազմանդամը ներկայացնել գործակիցների զանգվածի օգնությամբ:
5. Մշակել և իրականացնել ADT՝ հեռախոսահամարների գրքույկ ներկայացնելու համար: Յուրաքանչյուր գրառում պարունակում է անձի անունը, ազգանունը և հեռախոսի համարը: Նախատեսել գրքույկից տվյալ անձի հեռախոսի համարի որոնման և արտածման, նոր գրառման ավելացման գործողությունները:
6. Մշակել և իրականացնել ADT՝ ռացիոնալ թիվ ներկայացնելու համար: Նախատեսել ռացիոնալ թվերի բազմապատկման, բաժանման գործողությունները: Ռացիոնալ թիվը ներկայացվում է անկրճատելի կոտորակի տեսքով:
7. Մշակել և իրականացնել ADT՝ իրական թիվը ֆիքսած ստորակետի ֆորմատով ներկայացնելու համար: Նախատեսել թվերի գումարման, հանման և համեմատման գործողությունները:
8. Մշակել և իրականացնել ADT՝ կոմպլեքս թիվ ներկայացնելու համար: Նախատեսել կոմպլեքս թվերի գումարման, բազմապատկման և արտածման գործողությունները:

ԳԼՈՒԽ 2.

ԳԾԱՅԻՆ ՑՈՒՑԱԿ

2.1. ԳԾԱՅԻՆ ՑՈՒՑԱԿ ՏՎՅԱԼՆԵՐԻ ԱԲՍՏՐԱԿՏ ՏԻՊԸ

Գծային ցուցակ է կոչվում միատիպ տարրերի $a_1, \ldots, a_n$ ($n >= 0$) հաջորդականությունը, որի համար սահմանված է նախորդ տարր և հաջորդ տարր հասկացությունները հետևյալ կերպ՝ a_i -ի համար a_{i+1} -ը հաջորդ տարրն է ($1<=i<n$), իսկ a_{i-1} –ը՝ նախորդը ($1<i<=n$): Գծային ցուցակը կարող է նաև տարրեր չպարունակել ($n= 0$): Այդպիսի ցուցակը կոչվում է դատարկ ցուցակ:

Գծային ցուցակ ADT-ի նկարագրությունը հետևյալն է.

ADT List

Տվյալներ

Գծային ցուցակ

Գործողություններ

Կոնստրուկտոր

մուտք – չկա
նախապայման – չկա
պրոցես - դատարկ ցուցակի սկզբնարժեքավորում
ելք –չկա
վերջնապայման – դատարկ ցուցակ

Length

մուտք – չկա
նախապայման – չկա
պրոցես – ցուցակի տարրերի քանակի որոշում
ելք – ցուցակի տարրերի քանակ
վերջնապայման – չկա

Empty

մուտք – չկա
նախապայման – չկա
պրոցես – ցուցակի դատարկ լինելու պայմանի ստուգում
ելք – 1(true), եթե ցուցակը դատարկ է, 0 (false) հակառակ դեպքում:

վերջնապայման – չկա

Find

մուտք - ցուցակում փնտրվող տարրը
նախապայման – չկա
պրոցես – տարրի փնտրում ցուցակում
ելք – փնտրվող տարրի դիրքը ցուցակում
վերջնապայման – չկա

Insert

մուտք - ցուցակին ավելացվող տարրը և նրա դիրքը
նախապայման – չկա
պրոցես – տարրի ավելացում ցուցակում տրված դիրքում
ելք – չկա
վերջնապայման – ավելացված տարրով ցուցակ
բացառիկ իրավիճակ — ավելացվող տարրի դիրքը թույլատրելի սահմաններում չէ

Delete

մուտք – հեռացվող տարրի դիրքը
նախապայման – ցուցակը դատարկ չէ
պրոցես – տարրի հեռացում ցուցակից
ելք – հեռացված տարրը
վերջնապայման - հեռացված տարրով ցուցակ
բացառիկ իրավիճակ — հեռացվող տարրի դիրքը թույլատրելի սահմաններում չէ

ClearList

մուտք – չկա
նախապայման - չկա
պրոցես – ցուցակի բոլոր տարրերի հեռացում
ելք – չկա
վերջնապայման –դատարկ ցուցակ

վերջ ADT List

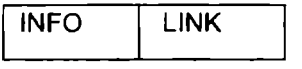
2.2. ԳԾԱՅԻՆ ՑՈՒՑԱԿԻ ԻՐԱԿԱՆԱՑՈՒՄ

Գծային ցուցակը իրականացվում է տվյալների այնպիսի կառուցվածքների միջոցով, որոնք հիմնված են հիշողությունում տվյալների պահպանման հաջորդական կամ կապակցված եղանակների վրա:

Տվյալների պահպանման հաջորդական եղանակի դեպքում ցուցակի իրականացումը կատարվում է զանգվածի միջոցով:

Տվյալների պահպանման կապակցված եղանակի դեպքում գծային ցուցակի իրականացումը կատարվում է կապակցված ցուցակի միջոցով:

Կապակցված ցուցակի հանգույցը ունի հետևյալ կառուցվածքը՝



որտեղ INFO-ն ինֆորմացիոն դաշտն է, իսկ LINK-ը՝ հաջորդ հանգույցի հետ կապի դաշտը:

Եթե x-ը ցուցիչ է հանգույցի վրա, ապա INFO(x)-ով նշանակենք այդ հանգույցի ինֆորմացիոն դաշտը, իսկ LINK(x)–ով՝ կապի դաշտը: Ենթադրվում է, որ ցուցակի տարրերը տեղակայված են դինամիկ բաշխվող հիշողության մեջ՝ կույտում, որը նույնպես իրենից ներկայացնում է նույն տիպի հանգույցներից կազմված կապակցված ցուցակ: AVAIL-ով նշանակենք ցուցիչը կույտի առաջին տարրի վրա:

Առանձնացնենք ցուցակների հետ կապված երկու վիճակներ՝ OVERFLOW (գերհագեցում) և UNDERFLOW (պակասում): OVERFLOW կանվանենք այն իրավիճակը, երբ ցուցակն արդեն լցված է և փորձ է արվում ցուցակին ավելացնել նոր տարր: UNDERFLOW-ն այն իրավիճակն է, երբ փորձ է արվում դատարկ ցուցակից հեռացնել տարր: Այս դեպքերում ծրագիրը համապատասխան ձևով պետք է արձագանքի ստեղծված իրավիճակին:

Դիտարկենք կույտից հանգույց վերցնելու և հանգույցը կույտ վերադարձնելու գործողությունները: NOP–ով նշանակենք դատարկ գործողությունը, x-ին y-ի արժեքը վերագրելու համար կօգտագործենք x←y նշանակումը:

1. x⇐ AVAIL – կույտից վերցնել նոր հանգույց և նրա ցուցիչը գրանցել x-ում: Այս գործողությունը իրականացվում է հետևյալ կերպ.

   եթե (AVAIL = NULL)
   ապա NOP
   այլապես { x ← AVAIL, AVAIL ← LINK (AVAIL)}

2. x ⇒ AVAIL – հանգույցը վերադարձնել կույտ: Այս գործողությունը իրականացվում է հետևյալ կերպ.

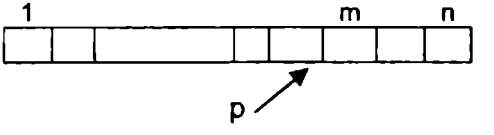
   { LINK(x) ← AVAIL, AVAIL ← x }

Գծային ցուցակի իրականացում զանգվածի միջոցով

Դիցուք $x_1, \ldots, x_n$–ը ցուցակը ներկայացնող զանգվածն է, իսկ m-ը ցուցակի երկարությունը (0<=m<=n):

Նկարագրենք ցուցակի հետ կատարվող հետևյալ գործողությունները:

1. InsertAt(p, item) - item տարրի ավելացում ցուցակի p-րդ դիրքում: Այս դեպքում զանգվածի բոլոր տարրերը սկսած p-րդ դիրքից տեղափոխվում են մեկ դիրքով դեպի աջ:



եթե (m=n)
  ապա OVERFLOW
  այլապես
    եթե (p<1 կամ p>m+1)
      ապա NOP
      այլապես { x[i+1] ← x[i], (i=m,m-1,…,p),
      x[p] ← item, m←m+1}

2. Delete(p) - տարրի հեռացում p -րդ դիրքից: Այս դեպքում զանգվածի բոլոր տարրերը (p +1)-րդ դիրքից սկսած տեղափոխվում են մեկ դիրքով դեպի ձախ:

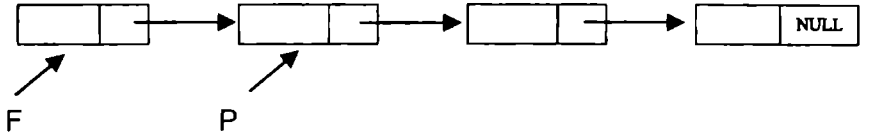
եթե (m=0)
  ապա UNDERFLOW
  այլապես
    եթե *(p<1 կամ p>m)*
      ապա NOP
      այլապես { x[i] ← x[i+1], (i=p,p+1,…,m-1), m←m-1}

Գծային ցուցակը կարելի է իրականացնել գծային միակապ, ցիկլիկ և երկկապ ցուցակների միջոցով: Ստորև բերված են այդպիսի ցուցակների տեսքերը և նկարագրված են նրանց հետ կատարվող որոշ գործողություններ:

Գծային ցուցակի իրականացում միակապ ցուցակի միջոցով

Կապակցված ցուցակների պարզագույն դեպքը միակապ գծային ցուցակն է: Միակապ գծային ցուցակում յուրաքանչյուր հանգույց կապ է պահպանում ցուցակում իրեն հաջորդող հանգույցի հետ: Վերջին հանգույցի կապի դաշտում գրվում է NULL:

Միակապ գծային ցուցակը կարելի է պատկերել հետևյալ ձևով.



Այստեղ F–ը ցուցիչ է ցուցակի առաջին հանգույցի վրա, իսկ P-ն ցուցիչ է ցուցակի որևէ հանգույցի վրա:

Նկարագրենք միակապ ցուցակի հետ կատարվող հետևյալ գործողությունները:

1. InsertAfter(P, item) - ցուցակում P($P \neq \Lambda$) ցուցիչով հանգույցից հետո item տարրի ավելացում:

   x⇐AVAIL, INFO(x) ←item, LINK(x) ←LINK(P), LINK(P) ←x

2. InsertAt(k,item)- item տարրի ավելացում ցուցակի k-րդ դիրքում: Դիրքերը համարակալված են սկսած զրոյից:

   եթե (k=0)

   ապա {x ⇐ AVAIL,INFO(x) ← item, LINK(x) ← F, F ← x}

   այլապես {P←F,

   P←LINK (P) (i=0,1,..., k-1),

   InsertAfter (P, item)}

3. DeleteAfter(P) – P ($P \neq \Lambda$) ցուցիչով հանգույցին հաջորդող հանգույցի հեռացում:

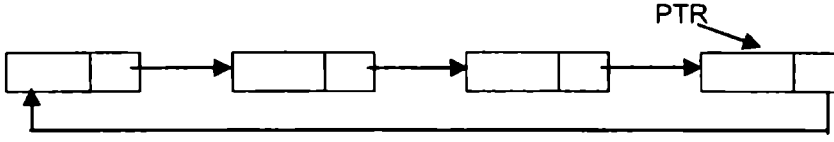
   եթե (LINK(P) = 0)

   ապա NOP

   այլապես {x ← LINK(P), LINK(P) ← LINK(x), x ⇒ AVAIL }

Գծային ցուցակի իրականացում ցիկլիկ ցուցակի միջոցով

Ցիկլիկ ցուցակ կանվանենք այն միակապ գծային ցուցակը, որի վերջին հանգույցի կապի դաշտում գրված է առաջին հանգույցի հասցեն:



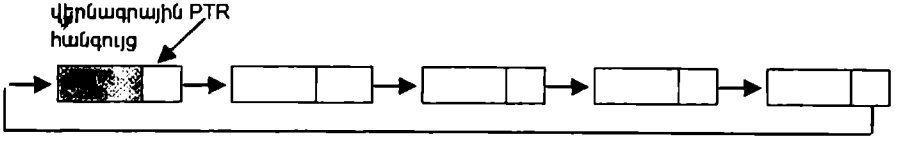
Այստեղ PTR-ը ցուցիչ է ցիկլիկ ցուցակի որևէ հանգույցի վրա:

Նկարագրենք ցիկլիկ ցուցակի հետ կատարվող հետևյալ գործողությունները:

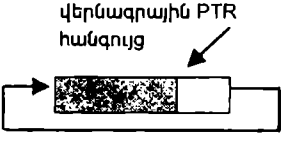
1. InsertLeft(item) - ցիկլիկ ցուցակում PTR ցուցիչով հանգույցից հետո item տարրի ավելացում:
   x⇐AVAIL, INFO (x) ← item,
   եթե (PTR = NULL)
   ապա {LINK (x) ←x, PTR←x}
   այլապես {LINK(x) ← LINK(PTR), LINK(PTR) ← x}
2. InsertRight(item) -ցիկլիկ ցուցակում PTR ցուցիչով հանգույցից հետո item տարրի ավելացում և PTR ցուցչի տեղափոխում ավելացվող x հանգույցի վրա:
   InsertLeft(item), PTR ← x
3. DeleteAfter() - ցիկլիկ ցուցակից PTR ցուցիչով հանգույցին հաջորդող տարրի հեռացում:
   եթե (PTR = NULL)
   ապա UNDERFLOW
   այլապես { x←LINK (PTR), LINK(PTR) ←LINK(x),
   եթե (PTR = x) *ապա* PTR← NULL,
   x ⇒ AVAIL }
4. ClearList() - ցուցակի բոլոր տարրերի հեռացում:
   եթե (PTR = NULL)
   ապա NOP
   այլապես { P← AVAIL, AVAIL← LINK(PTR), LINK(PTR) ←P,
   PTR← NULL }

Քանի որ ցիկլիկ ցուցակում վերջին հանգույցի կապի դաշտը դատարկ չէ, առաջանում է ցուցակի վերջը որոշելու խնդիր: Այս դեպքում պետք է շարժվել ցուցակի վրայով՝ անցնելով մի հանգույցից մյուսին, այնքան ժամանակ, քանի դեռ չենք հասել ցուցակի սկզբնական դիրքին (այն հանգույցին, որից սկսվել էր տարրերի դիտարկումը):

Այս խնդիրը կարելի է լուծել նաև ցիկլիկ ցուցակում ավելացնելով հատուկ նշված հանգույց, որը կանգառի համար հարմար դիրք կհանդիսանա: Ցուցակի այդ հանգույցը կոչվում է վերնագրային հանգույց: Այդպիսի ցուցակների առավելություններից մեկն այն է, որ նրանք երբեք դատարկ չեն լինում: Ստորև պատկերված է վերնագրային հանգույցով ցիկլիկ ցուցակ:



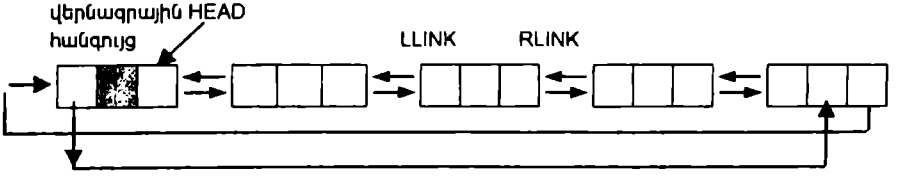
Դատարկ ցուցակը կունենա հետևյալ տեսքը՝



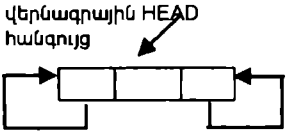
Գծային ցուցակի իրականացում երկկապ ցուցակի միջոցով

Կապակցված գծային ցուցակների տարատեսակներից է երկկապ գծային ցուցակը: Երկկապ գծային ցուցակում յուրաքանչյուր հանգույց կապ է պահպանում նախորդ և հաջորդ հանգույցների հետ՝ համապատասխանաբար LLINK և RLINK ցուցիչների միջոցով:

Ստորև պատկերված է վերնագրային հանգույցով երկկապ գծային ցուցակ: HEAD-ը վերնագրային հանգույցի ցուցիչն է:



Դատարկ ցուցակն այս դեպքում կունենա հետևյալ տեսքը՝



Նկատենք, որ երկկապ ցուցակի բոլոր հանգույցների համար տեղի ունի հետևյալ պայմանը.

RLINK(LLINK(P))=LLINK(RLINK(P))=P,

որտեղ P-ն հանգույցի ցուցիչ է:

Նկարագրենք վերնագրային հանգույցով երկկապ ցուցակի հետ կատարվող հետևյալ գործողությունները:

1. IsEmpty() - ցուցակի դատարկ լինելու ստուգում:
   եթե (LLINK(HEAD)=HEAD)
   ապա TRUE
   այլապես FALSE

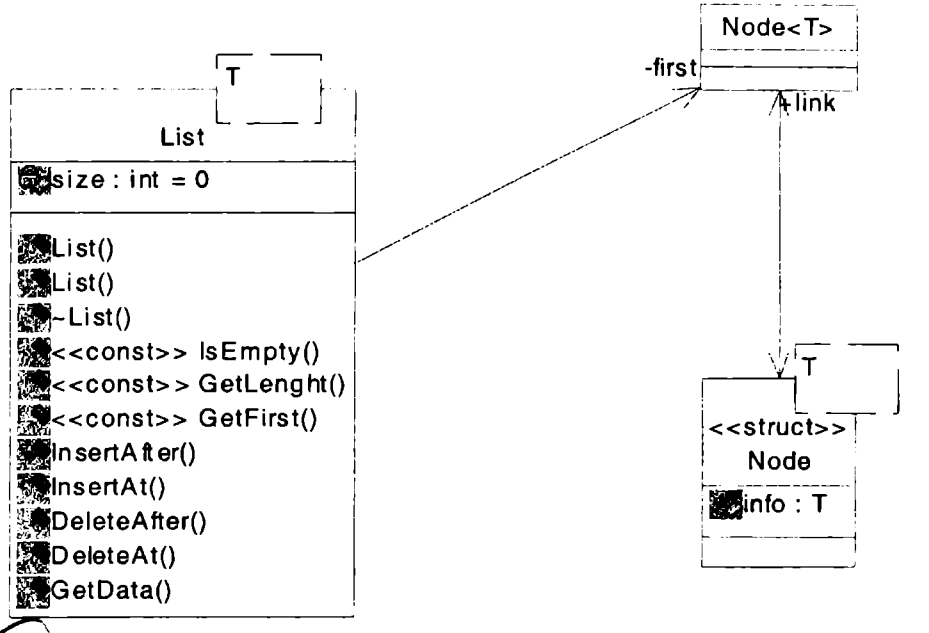
2. InsertRight(P, item) - ցուցակում P (P≠Λ) ցուցիչով հանգույցից հետո item տարրի ավելացում:
   x⇐AVAIL, INFO(x) ←item, RLINK(x) ←RLINK(P),
   LLINK(x) ←P, y←RLINK(P), RLINK(P) ←x, LLINK(y) ←x

3. Delete(P) - P (P≠Λ) ցուցիչով հանգույցի հեռացում:
   եթե (P = HEAD)
   ապա UNDERFLOW
   այլապես {
   RLINK(LLINK(P))← RLINK(P),
   LLINK(RLINK(P))← LLINK(P),
   P ⇒ AVAIL
   }

2.3 ԳԾԱՅԻՆ ՑՈՒՑԱԿԻ ԻՐԱԿԱՆԱՑՈՒՄԸ C++ ԼԵԶՎՈՎ

Ծրագրավորման լեզուներում տվյալների աբստրակտ տիպերը իրականացնելու համար սահմանվում են տվյալների նոր տիպեր, որոնք իրականացվում են դասերի միջոցով: Ենթադրվում է, որ տվյալ-

ների հաջորդականությունն իրականացվում է միակապ գծային ցուցակի միջոցով: C++ լեզվում List դասը ներկայացնենք շաբլոնի տեսքով, որտեղ որպես շաբլոնի պարամետր համարում ենք ցուցակի տարրերի տիպը: Node դասի շաբլոնով տրվում է ցուցակի հանգույցի տիպը: Ստորև ներկայացված է List դասի շաբլոնի UML-դիագրամը:



List դասի հայտարարումը կատարենք List.h ֆայլում:

```
// ցուցակի հանգույցի Node տիպի հայտարարում
template <class T>
struct Node {
        T info;
        Node<T> *link;
};
//LIST դասի հայտարարում
template<class T>
class List{
private:
        Node< T >*first;
        int size;
public:
        List (); // լռությամբ կոնստրուկտոր
        List (const List&); // պատճենի կոնստրուկտոր
        ~ List (); // դեստրուկտոր
        // գործողություններ
```

```
        bool IsEmpty() const;
        int GetLenght() const;
        Node< T >* GetFirst() const;
        void InsertAfter(Node< T >*p, T item);
        void InsertAt(int n, T item);
        void DeleteAfter (Node< T >*p);
        void DeleteAt (int n);
        T & GetData(Node< T >*p);
};
```

Ստորև բերված է List դասի իրականացման List.cpp ֆայլը:

```
#include "List.h"
// լռությամբ կոնստրուկտոր
template <class T>
List <T>::List ():size(0), first(NULL){
}
// պատճենի կոնստրուկտոր
template <class T>
LIst <T>::List(const List& aList){
        // ցուցակի առաջին հանգույցի պատճենում
        Node <T> *p=new Node <T>, *p1= aList.first;
        GetData(p)=aList.GetData(p1);
        first=p; first→link=NULL;
        // ցուցակի մնացած հանգույցների պատճենում
        for(int i=0; i< aList.GetLenght() -1; i++){
                p1=p1→link;
                insertAfter (p, aList.GetData(p1));
                p=p→link;
        }
        size= aList.size;
}

// դեստրուկտոր
template <class T>
List <T>::~List (){
 if (first) {
                while (first → link)
                        DeleteAfter(first);
                delete first;
                first=0;
        }
        size=0;
}
```

```
// IsEmpty()-ստուգում է ցուցակի դատարկ լինելը
template <class T>
bool List <T>:: IsEmpty() const {
        return size==0;
}
// GetLenght() - վերադարձնում է ցուցակի տարրերի քանակը
template <class T>
int List <T>:: GetLenght() const {
        return size;
}
//GetFirst() - վերադարձնում է ցուցիչ ցուցակի առաջին տարրի վրա
template <class T>
Node<T>* List <T>:: GetFirst () const{
        return first;
}
// InsertAfter(p,item)-ցուցակում p ցուցիչով հասցեագրից
// հետո ավելացնում է item տարրը
template <class T>
void List <T>::InsertAfter (Node< T >*p, T item){
        Node <T>* q=new Node<T>;
        q→info=item;
        q→link=p→link;
        p→link=q;
        size++;
}
// InsertAt(n,item) - ցուցակի n-րդ դիրքում ավելացնում է item
// տարրը
template <class T>
void List <T>::InsertAt (int n, T item){
        if (n<0 || n> GetLenght())
        throw "position is out of range";
        if (n==0) {
                Node <T>* q=new Node<T>;
                q→info=item;
                q→link=first;
                first=q;
                size++;
                return;
        }
        Node <T>*p=first;
        for(int i=0;i<n-1;i++)
                p=p→link;
        InsertAfter (p,item);
}
```

```
// DeleteAfter(p)-ցուցակից հեռացնում է p ցուցիչով
// հանգույցին հաջորդող տարրը
template <class T>
void List <T>:: DeleteAfter(Node< T > *p) {
        if (IsEmpty()){
                throw "List is empty";
        }
        if (p->link==NULL) {
                throw "No element to delete";
        }
        Node <T>*q=p->link;
        p->link=q→link;
        delete q;
        size--;
        return;
}

// DeleteAt(n) - հեռացնում է ցուցակի n-րդ դիրքում գտնվող
// տարրը
template <class T>
void List <T>:: DeleteAt(int n) {
        if (n<0 || n>=GetLenght())
                throw "position is out of range";
        if (n==0 && GetLenght()!=0) {
                Node <T>*q=first;
                first= first→link;
                delete q;
                size--;
                return;
        }
        Node <T>*p=first;
        for(int i=0;i<n-1;i++)
                p=p→link;
        DeleteAfter (p,item);
}

// GetData(p) - վերադարձնում է ցուցակի p ցուցիչով
// հանգույցի տարրը
template <class T>
T& List <T>::GetData(Node< T >*p){
        return p→info;
}
```

Բերենք List դասն օգտագործող ծրագրի օրինակ:

```
# include <iostream.h>
# include "List.h"
int main(){
  try{
        const int n=10;
        List <double> L1; // ստեղծվում է դատարկ ցուցակ
        //L1 ցուցակը արժեքավորել ստեղնաշարից ներմուծված
        // double տիպի տարրերով
        double a[ n] ; int i;
        for(i=0; i<n-1; i++){
              cin>> a[ i] ;
              L1.InsertAt(i, a[ i] );
        }
        Node< double >* p;
        List <double> L2=L1; // L2 ցուցակը արժեքավորվում է
                             // L1 ցուցակի տարրերով
        for( i=0, p=
L2.GetFirst();i<L2.GetLenght();i++,p=p->link)
        // արտածվում են L2 ցուցակի տարրերը
              cout<< L2.GetData(p)<<' ' ;
  }     //try բլոկի ավարտ
catch (char* a) {       // բացառիկ իրավիճակի մշակում
        cout<<a<<endl;
  }
  return 0;
}
```

2.4. ԳԾԱՅԻՆ ՑՈՒՑԱԿ ՏՎՅԱԼՆԵՐԻ ԱԲՍՏՐԱԿՏ ՏԻՊԻ ԻՐԱԿԱՆԱՑՈՒՄԸ ՇԱԲԼՈՆՆԵՐԻ ՍՏԱՆԴԱՐՏ ԳՐԱԴԱՐԱՆՈՒՄ

Օբյեկտներին կողմնորոշված ծրագրավորման լեզուներում նախատեսված են ստանդարտ դասեր, որոնք առավել հաճախ են օգտագործվում ծրագրորդների կողմից: Այդ դասերն ապահովում են աշխատանք տարբեր տիպի տվյալների, պատուհանների, ինչպես նաև բազմաբնույթ գրաֆիկական օբյեկտների հետ: Այն դեպքերում, երբ ստանդարտ դասերը և միջոցները չեն բավարարում օգտագործողի պահանջներին, հարկ է լինում մշակել սեփական ծրագրային միջոցներ:

C++ լեզվում տվյալների որոշ աբստրակտ տիպեր իրականացված են շաբլոնների ստանդարտ գրադարանի (STL) դասերի միջոցով: STL գրադարանի տարրերը բաժանվում են երեք խմբի՝ կոնտեյներներ, իտերատորներ և ալգորիթմներ:

Կոնտեյներները օբյեկտներ են, որոնք նախատեսված են այլ օբյեկտների պահպանման համար: STL-ի կոնտեյներները բաժանվում են երկու խմբի՝ հաջորդական և ասոցատիվ: Հաջորդական կոնտեյներները՝ vector, deque, list, stack, queue, priority_queue, ապահովում են նույնատիպ օբյեկտների վերջավոր բազմությունների պահպանումը հաջորդականությունների տեսքով: Ասոցատիվ կոնտեյներները՝ set, multiset, map, multimap, ապահովում են օբյեկտների բազմության տարրի հասանելիությունը ըստ բանալու:

Իտերատորները օբյեկտներ են, որոնք հնարավորություն են տալիս շարժվել կոնտեյների մեջ մտնող օբյեկտների վրայով: Իտերատորի պարզագույն դեպքը ցուցիչներն են, որոնց օգնությամբ կարելի է շարժվել զանգվածի կամ կապակցված ցուցակի վրայով:

Կոնտեյներների նկատմամբ կարելի է կիրառել STL-ի ալգորիթմներ: Դրանցից են կոնտեյների տարրերի կարգավորման, տարրի որոնման, կոնտեյներների միավորման, կոնտեյների մեջ ընդգրկված օբյեկտների մշակման ալգորիթմները և այլն:

STL գրադարանի բոլոր կոնտեյներային դասերը իրականացված են շաբլոնային դասերի միջոցով, որոնք օգտագործում են շաբլոնի երկու պարամետր: Առաջին պարամետրով տրվում է կոնտեյներում ընդգրկված տվյալների տիպը, երկրորդ պարամետրը նշում է հիշողության բաշխման եղանակը, որի համար հաճախ օգտագործվում է լռությամբ սահմանված արժեքը՝ allocator դասի օբյեկտը:

Ստորև ներկայացված է STL-ի list շաբլոնային դասի մի հատված:

```
template<class T, class A = allocator<T> >
class list {
public:
      typedef A allocator_type;
      typedef A::size_type size_type;
      typedef A::reference reference;
      typedef A::value_type value_type;
      typedef T0 iterator;
      explicit list(const A& al = A());
```

```
    explicit list(size_type n, const T& v = T(),
                  const A& al = A());
    list(const list& x);
    list(const_iterator first, const_iterator last,
         const A& al = A());
    iterator begin();
    iterator end();
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    reference front();
    reference back();
    void push_front(const T& x);
    void pop_front();
    void push_back(const T& x);
    void pop_back();
    iterator insert(iterator it, const T& x = T());
    void insert(iterator it, size_type n, const T& x);
    void insert(iterator it, const_iterator first,
                const_iterator last);
    void insert(iterator it, const T *first,
                const T *last);
    iterator erase(iterator it);
    iterator erase(iterator first, iterator last);
    void clear();
    void swap(list x);
    void splice(iterator it, list& x);
    void remove(const T& x);
    void merge(list& x);
    void merge(list& x, greater<T> pr);
    void sort();
    void reverse();
protected:
    A allocator;
};
```

STL-ում յուրաքանչյուր կոնտեյներային դասի համար նախատեսված է iterator դասը, որն ունի յուրատեսակ իրականացում: Բոլոր տիպի կոնտեյներների իտերատորների համար իրականացված են հետևյալ գործողությունները.

1. ապահասցեավորման (*p-ն՝ p իտերատորով մատնանշվող օբյեկտն է),
2. վերագրման (p=q, որտեղ p-ն և q-ն միևնույն տիպի իտերատորներ են),

3. համեմատման (p==q, p!=q, որտեղ p-ն և q-ն միևնույն տիպի իտերատորներ են),
4. ավելացման (++, նախածանցային և վերջածանցային ինկրեմենտ):

Կոնտեյներային դասերը պարունակում են begin() և end() ֆունկցիաներ: Այդ ֆունկցիաները վերադարձնում են իտերատորներ, որոնք համապատասխանաբար մատնանշում են կոնտեյների առաջին տարրը և վերջին տարրին հաջորդող երևակայական տարրը:

Բերենք STL-ի list կոնտեյների օգտագործման օրինակ:

```
#include <list>
#include <iostream>
using namespace std ;
typedef list<int> LISTINT;
void main()
{
      int rgTest1[ ]  = { 5,6,7} ;
      int rgTest2[ ]  = { 10,11,12} ;

      list<int> listInt;// կառուցել listInt դատարկ ցուցակը
      list<int> listAnother; //կառուցել listAnother դատարկ
                             // ցուցակը
      list<int>::iterator i;
      // տարրի ավելացում listInt ցուցակին
      listInt.insert (listInt.begin(), 2);
      listInt.insert (listInt.begin(), 1);
      listInt.insert (listInt.end(), 3);

      // 1 2 3 - listInt ցուցակի պարունակությունը
      for (i = listInt.begin(); i != listInt.end(); ++i)
      cout << *i << " ";
      cout << endl;

      // listInt ցուցակի վերջում ավելացնել 4-ի հավասար 3 տարր
      listInt.insert (listInt.end(), 3, 4);

      // 1 2 3 4 4 4 - ցուցակի պարունակությունը
      for (i = listInt.begin(); i != listInt.end(); ++i)
            cout << *i << " ";
      cout << endl;
```

```
    //listInt ցուցակի վերջում ավելացնել rgTest1 զանգվածի
    // բոլոր տարրերը
    listInt.insert (listInt.end(), rgTest1, rgTest1 + 3);

    //1 2 3 4 4 4 5 6 7 - listInt ցուցակի պարունակությունը
    for (i = listInt.begin(); i != listInt.end(); ++i)
         cout << *i << " ";
    cout << endl;

    //listAnother ցուցակի վերջում ավելացնել rgTest2
    // զանգվածի բոլոր տարրերը
    listAnother.insert (listAnother.begin(), rgTest2,
                        rgTest2+3);

    // listInt ցուցակի վերջից կցել listAnother ցուցակը
    listInt.insert(listInt.end(),listAnother.begin(),
                   listAnother.end());

    // 1 2 3 4 4 4 5 6 7 10 11 12-listInt ցուցակի
    // պարունակությունը
    for (i = listInt.begin(); i != listInt.end(); ++i)
         cout << *i << " ";
    cout << endl;
}
```

Ծրագրի աշխատանքի արդյունքում կարտածվի

```
1 2 3
1 2 3 4 4 4
1 2 3 4 4 4 5 6 7
1 2 3 4 4 4 5 6 7 10 11 12
```

STL գրադարանում ընդգրկված իտերատորները բաժանվում են հինգ խմբի՝ մուտքային իտերատորներ (InputIterator), ելքային իտերատորներ (OutputIterator), ուղիղ իտերատորներ (ForwardIterator), երկկողմանի իտերատորներ (BidirectionalIterator), ուղիղ հասանելիության իտերատորներ (RandomAccessIterator): Իտերատորների խմբերը բնորոշվում են դրանց համար սահմանված թույլատրելի գործողություններով:

Մուտքային իտերատորները հնարավորություն են տալիս կարդալ կոնտեյների մեջ ընդգրկված օբյեկտները: Ելքային իտերատորների օգնությամբ կոնտեյների մեջ կարելի է օբյեկտներ դնել (գրել): Ուղիղ իտերատորները թույլ են տալիս շարժվել կոնտեյների վրա-

յով մեկ ուղղությամբ, ընդ որում կատարել օբյեկտների և՛ կարդալու, և՛ գրելու գործողությունները: Երկկողմանի իտերատորները ընդգրկում են ուղիղ իտերատորների բոլոր հնարավորությունները, ինչպես նաև ապահովում են կոնտեյների վրայով ուղիղ և հակադարձ ուղղություններով շարժվելը: Ուղիղ հասանելիության իտերատորները օժտված են երկկողմանի իտերատորների բոլոր հատկություններով և, բացի դրանից, ապահովում են կոնտեյտերի մեջ մտնող կամայական օբյեկտի հասանելիությունը:

STL-ի ալգորիթմները իրականացված են գլոբալ շաբլոնային ֆունկցիաների տեսքով: Ալգորիթմները կապվում են կոնտեյների հետ իտերատոր-պարամետրերի միջոցով, ուստի կարող են կիրառվել կամայական կոնտեյների նկատմամբ, որն աջակցվում է համապատասխան տիպի իտերատորով: Ստորև բերված է find ալգորիթմն իրականացնող շաբլոնային ֆունկցիայի նախատիպը.

```
template<class InIt, class T>
InIt find(InIt first, InIt last, const T& val);
```

Այստեղ first և last պարամետրերով տրվում են իտերատորներ կոնտեյների դիտարկվող օբյեկտների միջակայքի սկզբի տարրի և վերջին տարրին հաջորդող տարրի վրա: Հեշտ է կռահել, որ first-ը և last-ը պետք է ունենան մուտքային իտերատորների հատկությունները, այսինքն, ալգորիթմը կարող է կիրառվել կամայական կոնտեյների նկատմամբ:Find ֆունկցիան որոշում է, թե արդյո՞ք կոնտեյների [first, last) միջակայքում կա val-ին հավասար տարր: Եթե այդպիսի տարր չկա, ապա ֆունկցիան վերադարձնում է last, հակառակ դեպքում ֆունկցիան վերադարձնում է իտերատոր val-ին հավասար տարրի վրա:

```
#include <algorithm>
#include <list>
#include <iostream>
using namespace std;

void main(){
    const int ARRAY_SIZE = 8 ;
    int IntArray[ ARRAY_SIZE] = {1, 2, 3, 4, 4, 5, 6, 7};
    // ցուցակի արժեքավորումը
    list<int>IntList(IntArray,IntArray+ARRAY_SIZE);
    list<int>::iterator itList;
```

```
    list<int>::iterator location ; //իտերատոր առաջին
                                   //համընկնող տարրի վրա
    int value = 4 ;                //փնտրվող տարր
    // IntList ցուցակի արտածում
    cout << "IntList { " ;
    for(itList=IntList.begin();itList!=IntList.end();
        itList++) cout << *itList<< ", " ;
    cout << " }" << endl ;
    // [ first, last + 1) միջակայքում value արժեքի փնտրում
    location = find(IntList.begin(), IntList.end(), value);
    //որոնման արդյունքի մասին հաղորդագրության արտածում
    if (location!=IntList.end()) // value-ն առկա է
                                 // ցուցակում
      cout <<"The sequence contains an element with value"
           <<value<< endl;
    else           // ցուցակում value-ին համապատասխան արժեք չկա
      cout   <<"The sequence does not contain any elements"
             <<" with value " << value << endl;
}
```

Ծրագիրն արտածում է՝

```
IntList { 1, 2, 3, 4, 4, 5, 6, 7, }
The sequence contains an element with value 4
```

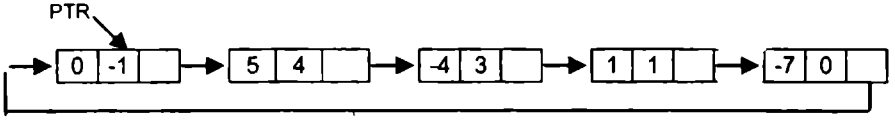
2.5. ԳԾԱՅԻՆ ՑՈՒՑԱԿԻ ԿԻՐԱՌՈՒԹՅՈՒՆՆԵՐ

Դիտարկենք ամբողջ գործակիցներով բազմանդամների գումարման խնդիրը: Բազմանդամները ներկայացնենք վերնագրային հանգույցով կապակցված ցիկլիկ ցուցակների տեսքով, որի հանգույցները բազմանդամի ոչ զրոյական անդամներն են և ունեն հետևյալ կառուցվածքը՝

COEF	PWR	LINK

Այստեղ COEF-ը միանդամի գործակիցն է, իսկ PWR –ը՝ աստիճանը: Ենթադրվում է, որ բազմանդամի անդամներն ընթանում են աստիճանների նվազման կարգով:

Օրինակ, $5x^4-4x^3+x-7$ բազմանդամը ներկայացվում է հետևյալ կերպ.



Այստեղ, PTR-ով նշված վերնագրային հանգույցի PWR դաշտում գրված –1 արժեքն ապահովում է ալգորիթմի բնականոն ավարտը:

Ալգորիթմ (Բազմանդամների գումարում): Այս ալգորիթմը (P) բազմանդամը գումարում է (Q) բազմանդամին, ենթադրելով, որ P-ն և Q-ն ցուցիչներ են, որոնք հղվում են վերը նշված տեսքով բազմանդամներին: P ցուցակը չի փոխվում, իսկ Q-ում ստացվում է գումարը: Ալգորիթմի աշխատանքի վերջում ցուցչային փոփոխականներն ընդունում են իրենց սկզբնական արժեքները: Օգտագործվում են նաև Q1 և Q2 լրացուցիչ ցուցիչներ:

Մուտք - (P) և (Q) բազմանդամներ
Ելք - (P) և (Q) բազմանդամների գումար
Քայլ 1.
[Սկզբնական տեղադրություններ] P←LINK(P), Q1←Q, Q←LINK(Q):
Այժմ P-ն և Q-ն նշում են(հղվում են) բազմանդամների ավագ անդամներին: Այս ալգորիթմում համարյա ամենուրեք Q1 փոփոխականը «մեկ քայլով հետ է մնում» Q-ից, այսինքն՝ Q=LINK(Q1):

Քայլ 2.
[Համեմատել PWR(P)-ն և PWR(Q)-ն]
Եթե (PWR(P) < PWR(Q)), ապա {Q1← Q, Q ← LINK(Q), անցնել *Քայլ 2*}:
Եթե (PWR(P) = PWR(Q)), ապա անցնել *Քայլ 3*:
Եթե (PWR(P) > PWR(Q)), ապա անցնել *Քայլ 5*:

Քայլ 3.
[Գումարել գործակիցները] Գտնվել են հավասար աստիճաններով երկու անդամներ:
Եթե (PWR(P)<0), ապա ալգորիթմն ավարտել, այլապես
COEF(Q)←COEF(Q)+COEF(P):

Եթե (COEF(Q)=0), ապա անցնել *Քայլ 4*, այլապես {Q1←Q, P←LINK(P), Q ← LINK(Q), անցնել *Քայլ 2*}:

Քայլ 4.

[Հեռացնել 0-ական անդամը]

Q2← Q, LINK(Q1) ← Q ← LINK(Q), AVAIL⇐ Q2, P← LINK(P), անցնել *Քայլ 2*.

Քայլ 5.

[Ավելացնել նոր անդամ]

(P) բազմանդամը պարունակում է այնպիսի ցուցիչով անդամ, որը բացակայում է (Q)-ում: Այդ անդամն ավելացվում է (Q)-ին: Q2⇐ AVAIL, COEF(Q2) ← COEF(P), PWR(Q2) ← PWR(P), LINK(Q2) ← Q, LINK(Q1) ← Q2, Q1← Q2, P← LINK(P), անցնել *Քայլ 2*.

ԽՆԴԻՐՆԵՐ

1. Տրված է ամբողջ թվերից կազմված հաջորդականություն: Իրականացնել այդ հաջորդականության մեջ տրված տարրի հեռացման և ավելացման գործողությունները:
2. Տրված է ամբողջ թվերից կազմված կարգավորված հաջորդականություն: Իրականացնել այդ հաջորդականությունից տրված տարրի բինար փնտրման ալգորիթմը:
3. Կապակցված ցուցակի տեսքով ներկայացված բազմանդամի համար իրականացնել բազմանդամի արժեքի հաշվումը տրված կետում:
4. Կապակցված ցուցակի տեսքով ներկայացված բազմանդամի համար իրականացնել բազմանդամների հանումը:
5. Կապակցված ցուցակի տեսքով ներկայացված բազմանդամների համար իրականացնել նրանց բազմապատկման ալգորիթմը:
6. Բազմությունը ներկայացվում է միակապ գծային ցուցակի տեսքով: Յուրաքանչյուր հանգույցում պահվում է բազմության մեկ տարր: Բազմության տարրերը ցուցակում դասավորված են աճման կարգով:
   ա. Տրված երկու բազմությունների համար կառուցել նրանց միավորումը:
   բ. Տրված երկու բազմությունների համար կառուցել նրանց հատումը:
   գ. Տրված երկու բազմությունների համար կառուցել նրանց տարբերությունը:

7. Իրականացնել կապակցված ցուցակի տարրերի կարգավորում պղպջակների ալգորիթմով:
8. Տրված ցուցակից հեռացնել կրկնվող տարրերը:
9. Սիմվոլային տողը ներկայացվում է միակապ գծային ցուցակի տեսքով: Յուրաքանչյուր հանգույցում պահվում է տողի մեկ նիշ: Դարձնել տրված տողի երկարությունը հավասար n-ի՝ դեն նետելով ավելորդ նշանները կամ լրացնելով տողը աջից բացատանիշերով:
10. Գտնել սիմվոլային տողում տրված սիմվոլի ամենաձախ հանդիպման ինդեքսը:
11. Իրական թվեր պարունակող տրված կարգավորված ցուցակում ավելացնել նոր տարր: Եթե ցուցակում կա տրված արժեքով տարրը, ապա նրա կրկնումների քանակը մեկով ավելացնել, հակառակ դեպքում այդ տարրն ավելացնել ցուցակի վերջում:
12. Իրական թվեր պարունակող տրված ցուցակից հեռացնել մեծագույն տարրը պարունակող բոլոր հանգույցները:
13. Եթե տրված ցուցակում կա տրված արժեքով տարրը, ապա նրա կրկնումների քանակը մեկով ավելացնել, հակառակ դեպքում այդ տարրն ավելացնել ցուցակի սկզբում:
14. Տրված են երկու ցուցակներ: Ստանալ երրորդ ցուցակը որպես այդ երկու ցուցակների կցման արդյունք:
15. Սիմվոլային տողը ներկայացվում է միակապ գծային ցուցակի տեսքով: Յուրաքանչյուր հանգույցում պահվում է տողի մեկ նիշ: Տրված երկու տողերի համար իրականացնել առաջինում երկրորդի ենթատող հանդիսանալու պայմանի ստուգումը:
16. Կապակցված գծային ցուցակի համար իրականացնել ListIterator դասը, որը ապահովում է ցուցակի տարրերի հասանելիություն ցուցիչի օգնությամբ: Այդ դասի համար սահմանեք հաջորդ տարրին անցնելու (ինկրեմենտի), ցուցակի սկզբի և վերջի վրա իտերատորի գտնվելու ստուգման գործողությունները:
17. Իրականացնել երկկապ ցիկլիկ գծային ցուցակը՝ ներառելով տրված հանգույցից հետո նոր տարրի ավելացման և տրված հանգույցի հեռացման գործողությունները:
18. Երկկապ ցիկլիկ ցուցակից հեռացնել տրված տարրը պարունակող բոլոր հանգույցները:
19. Իրականացնել երկկապ ցուցակի տարրերի կարգավորում պղպջակների ալգորիթմով:

20. Իրականացնել վերնագրային հանգույցով երկկապ գծային ցուցակը՝ ներառելով տրված հանգույցից հետո նոր տարրի ավելացման և տրված հանգույցի հեռացման գործողությունները:
21. Վերնագրային հանգույցով երկկապ ցիկլիկ ցուցակից հեռացնել տրված տարրը պարունակող բոլոր հանգույցները:
22. Երկկապ գծային ցուցակի համար իրականացնել DoubleListIterator դասը, որի օբյեկտն ապահովում է ցուցակի տարրերի հասանելիություն: Այդ դասի համար սահմանել հաջորդ և նախորդ տարրերին անցնելու (ինկրեմենտ, դեկրեմենտ), ապահասցեավորման, ցուցակի սկզբի և վերջի վրա իտերատորի գտնվելու ստուգման գործողությունները:
23. Տրված է ցիկլիկ կապակցված ցուցակ, որի տարրերը դասավորված են աճման կարգով: Ptr ցուցիչը ցույց է տալիս վերջին հանգույցի վրա: Ցուցակի տարրերը կարգավորել նվազման կարգով՝ առանց օգտագործելու տարրի ավելացման և հեռացման գործողությունները:
24. Կապակցված կառուցվածքները երբեմն օգտակար է իրականացնել չօգտվելով ցուցիչներից: Այդպիսի կառուցվածքի դերում կարող է հանդես գալ զանգվածը: Ամեն մի հանգույց պահվում է զանգվածի մեկ տարրի տեսքով, որը բաղկացած է երկու դաշտերից՝ item և next: item-ում պահվում է հերթական տարրի արժեքը, next-ում՝ ցուցակի հաջորդ հանգույցի ինդեքսը: Վերջին հանգույցի next անդամը հավասար է -1-ի: head ամբողջ փոփոխականը պարունակում է ցուցակի առաջին հանգույցի ինդեքսը: Զանգվածի այն տարրերը, որոնք տվյալ պահին չեն հանդիսանում կապակցված ցուցակի հանգույցներ, ձևավորում են ազատ հանգույցների ցուցակը: Այս ցուցակի առաջին հանգույցի ինդեքսը պարունակվում է free ամբողջ տիպի փոփոխականի մեջ: Կապակցված ցուցակին նոր տարր ավելացնելու համար հարկավոր է հանել հանգույց ազատ ցուցակի սկզբից՝ այնտեղ գրելով ավելացվող տարրի արժեքը: Տարրը կապակցված ցուցակից հեռացնելու համար հարկավոր է այն դնել ազատ ցուցակի սկզբում: Այս մոտեցումը թույլ կտա չտեղաշարժել զանգվածի տարրերը: Իրականացնել այս եղանակով ներկայացված գծային ցուցակը:
25. * Բազմանդամը ներկայացվում է վերնագրային հանգույցով միակապ գծային ցուցակի տեսքով: Յուրաքանչյուր հանգույցում նշվում է հերթական միանդամի գործակիցը, աստիճանը և կապը հաջորդ հանգույցի հետ: Աստիճանները դասավորված են միանդամների աստիճանների նվազման կարգով: Իրականացնել երկու բազմանդամների գումարման և բազմանդամի արտածման գործողությունները:

ԳԼՈՒԽ 3.

3.1 ՊԱՀՈՒՆԱԿ

3.1. ՊԱՀՈՒՆԱԿ ՏՎՅԱԼՆԵՐԻ ԱԲՍՏՐԱԿՏ ՏԻՊԸ

Պահունակն առավել հաճախ օգտագործվող և կարևոր տվյալների կառուցվածք է: Պահունակն օգտագործվում է, օրինակ, կոմպիլյատորներում ծրագրերի շարահյուսությունը ստուգելիս, արտահայտությունների արժեքները հաշվելիս, ֆունկցիաների կանչերն իրականացնելիս և այլն: Պահունակը պարունակում է տարրեր, և ունի իր գագաթը նշող դիրք (Top): Պահունակի հետ կատարվող հիմնական գործողություններն են տարրի ավելացումը (Push) և հեռացումը (Pop), որոնք իրականացվում են պահունակի գագաթից: Պահունակում վերջին ավելացրած տարրը հեռացվում է առաջինը: Այդ պատճառով պահունակի համար ասում են, որ նա ունի LIFO կարգ (last-in, first-out՝ վերջինը մտավ, առաջինը դուրս ելավ): Երբ պահունակում այլևս հնարավոր չէ տարր ավելացնել, ասում են, որ պահունակը լիքն է, իսկ երբ հնարավոր չէ տարր հեռացնել պահունակը դատարկ է համարվում:

ADT Stack

Տվյալներ

Տարրերի ցուցակ՝ պահունակի գագաթը նշող Top դիրքով:

Գործողություններ

Կոնստրուկտոր

նախապայման - չկա

պրոցես - պահունակի սկզբնարժեքավորում

Empty

մուտք - չկա

նախապայման - չկա

պրոցես - ստուգում, դատարկ է արդյոք պահունակը

ելք - true, եթե պահունակը դատարկ է, false՝ հակառակ դեպքում

վերջնապայման - չկա

Push

մուտք – պահունակին ավելացվող տարրը
նախապայման – չկա
պրոցես – տարրի ավելացում պահունակին
ելք – չկա
վերջնապայման – ավելացված տարրով պահունակ
բացառիկ իրավիճակ – նոր տարրի ավելացում լցված պահունակին

Pop

մուտք - չկա
նախապայման – պահունակը դատարկ չէ
պրոցես – տարրի հեռացում պահունակից
ելք – պահունակի գագաթի տարրը
վերջնապայման – հեռացված տարրով պահունակ
բացառիկ իրավիճակ – տարրի հեռացում դատարկ պահունակից

Peek

մուտք - չկա
նախապայման – պահունակը դատարկ չէ
պրոցես - պահունակի գագաթի տարրի ընտրում
ելք - պահունակի գագաթի տարրը
վերջնապայման – պահունակն անփոփոխ է
բացառիկ իրավիճակ – տարրի ընտրում դատարկ պահունակից

ClearStack

մուտք - չկա
նախապայման - չկա
պրոցես - պահունակի բոլոր տարրերի հեռացում
ելք – չկա
վերջնապայման – դատարկ պահունակ

վերջ ADT Stack

3.2. ՊԱՀՈՒՆԱԿԻ ԻՐԱԿԱՆԱՑՈՒՄ

Պահունակը, ինչպես և գծային ցուցակը, կարելի է իրականացնել երկու եղանակով՝ զանգվածի և կապակցված ցուցակի միջոցով:

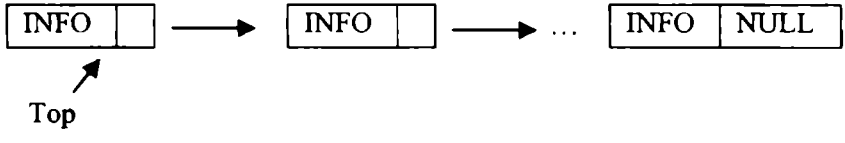
Պահունակի իրականացում զանգվածի միջոցով

Ենթադրենք ունենք $x_1 . . . . x_m$ զանգվածը և Top փոփոխականը, որով տրվում է պահունակի գագաթի ինդեքսը՝ $0 \le Top \le m$:

Նկարագրենք պահունակի հետ կատարվող գործողությունները:

1. Empty() - ստուգել պահունակի դատարկ լինելը:
   եթե (Top = 0)
   ապա TRUE
   այլապես FALSE
2. Push(y) - պահունակում ավելացնել նոր տարր:
   եթե (Top = m)
   ապա OVERFLOW
   այլապես { Top ← Top + 1, x[Top] ← y}
3. Pop() - պահունակից հեռացնել տարր:
   եթե (Top = 0)
   ապա UNDERFLOW
   այլապես Top ← Top – 1
4. GetTop(y) - ստանալ պահունակի գագաթի տարրը:
   եթե (Top = 0)
   ապա UNDERFLOW
   այլապես y ← x[Top]
5. ClearStack() - պահունակը դատարկել:
   Top ← 0

Պահունակի իրականացում կապակցված ցուցակի միջոցով



Այստեղ Top – ը պահունակի գագաթի ցուցիչն է: Եթե պահունակը դատարկ է, ապա ցուցիչն ունի NULL արժեքը:

Նկարագրենք պահունակի հետ կատարվող գործողությունները:

1. Empty() - ստուգել պահունակի դատարկ լինելը:
   եթե (Top = NULL)
   ապա TRUE
   այլապես FALSE

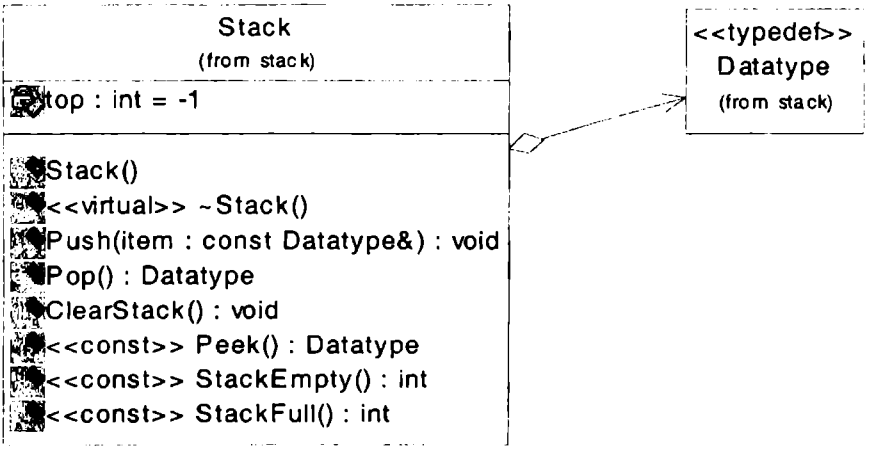
2. Push(y) - պահունակում ավելացնել նոր տարր:
   x ⇐ AVAIL, LINK(x) ← Top, INFO(x) ← y, Top ← x
3. Pop() - պահունակից հեռացնել տարր:
   եթե (Top = NULL)
   ապա UNDERFLOW
   այլապես {P ← Top, Top ← LINK(Top), p ⇒ AVAIL}
4. GetTop(y) – ստանալ պահունակի գագաթի տարրը:
   եթե (Top = NULL)
   ապա UNDERFLOW
   այլապես y ← INFO(Top)
5. ClearStack() - պահունակը դատարկել:
   Top ← NULL

3.3. ՊԱՀՈՒՆԱԿԻ ԻՐԱԿԱՆԱՑՈՒՄԸ C++ ԼԵԶՎՈՎ

Ստորև ներկայացված են պահունակի իրականացման տարբեր եղանակներ C++ լեզվով.

1. Պահունակի իրականացում զանգվածի միջոցով:
2. Պահունակի իրականացում կապակցված ցուցակի միջոցով:

Պահունակի իրականացում զանգվածի միջոցով: Պահունակն իրականացված է Stack դասի միջոցով: Ներկայացված է Stack դասի UML-դիագրամը:



Պահունակի տարրերը պահելու համար օգտագործվում է stacklist[MaxStackSize] զանգվածը (MaxStackSize=50): Պահունակի տարրի տիպը տրվում է typedef դիրեկտիվով որոշվող տիպի Datatype անունով: Պահունակի գագաթի տարրի դիրքը նշվում է top փոփոխականով: Սկզբում պահունակը դատարկ է և top-ի արժեքը -1 է: Զանգվածի միջոցով իրականացված պահունակի մեջ տարրերն ավելացվում են գագաթի փոփոխականի աճման կարգով (top=0,1,2) և պահունակից հանվում՝ գագաթի փոփոխականի նվազման կարգով (top=2,1,0):

Stack դասի հայտարարումը կատարենք Stack.h ֆայլում:

```
const int MaxStackSize = 50;
class Stack{
private:
      // պահունակի զանգվածը և գագաթը
      Datatype stacklist [ MaxStackSize] ;
      int top;
public:
      Stack();// կոնստրուկտոր
      // պահունակի փոփոխման գործողություններ
      void Push(const Datatype & item);
      Datatype Pop();
      void Clearstack();
      // պահունակի հասանելիության գործողություն
      Datatype Peek() const;
      // պահունակի վիճակի ստուգման գործողություններ
      int StackEmpty() const;
      int StackFull() const;
};
```

Ստորև բերված է Stack դասի իրականացման Stack.cpp ֆայլը:

```
#include "Stack.h"
// լռությամբ կոնստրուկտոր
Stack:: Stack (): top (- 1)
{ }

// պահունակում տարր ավելացնելու Push(item) գործողությունը
void Stack:: Push(const datatype & item) {
      // եթե պահունակը լիքն է, ծրագրի կատարումն ավարտել
      if (top == MaxStackSize -1) {
            cerr << "պահունակի հագեցում" << endl;
            exit (1);
      }
```

```
      // ավելացնել top ինդեքսը և պատճենել item-ը stacklist
      // զանգվածում
      top++;
      stacklist [ top] = item;
}

// պահուճակից տարր հեռացնելու Pop() գործողությունը
Datatype Stack:: Pop(){
      Datatype temp;
      // եթե պահուճակը դատարկ է, ծրագիրն ավարտել
      if (top == -1) {
            cerr << "փորձ է արվել դիմել դատարկ պահուճակին"
            << endl;
            exit(1);
      }
      temp = stacklist [ top] ; // կարդալ պահուճակի գագաթի
                                // տարրը
      top -- ; // նվազեցնել top-ը
      return temp;
}

// պահուճակից տարր կարդալու գործողությունը Peek()-ն է,
// այն վերադարձնում է պահուճակի գագաթի տվյալը
Datatype Stack:: Peek () const{
      // եթե պահուճակը դատարկ է, ծրագիրն ավարտել
      if (top == -1) {
            cerr << "փորձ է արվել դատարկ պահուճակից տվյալ
            վերցնել" <<endl;
            exit (1);
      }
      return stacklist [ top] ;
}

// StackEmpty() - usուգում է, արդյո՞ք պահուճակը դատարկ է
int Stack:: StackEmpty() const{
      return top == -1; // վերադարձնել top == -1-ի
                        // տրամաբանական արժեքը
}

// StackFull() - usուգում է, լի՞ք է արդյոք պահուճակը
int Stack:: StackFull () const{
      return top == maxStackSize - 1;
}
```

```
// ClearStack()-դատարկում է պահունակը՝ վերականգնելով
// կոնստրուկտորով որոշված սկզբնական վիճակը
void Stack::ClearStack(void) {     // դատարկել պահունակը
     top = -1;
}
```

Դիտարկենք պահունակի օգտագործման օրինակ:

```
typedef int Datatype;
#include "stack.h"
int main(){
     Stack S;            // հայտարարել Stack տիպի օբյեկտ
     S.Push(10);         // S պահունակում ավելացնել 10
     cout << S.Peek() << endl; // արտածել 10
     // պահունակից հանել 10-ը և արտածել
     if (!S.StackEmpty()){
          temp = S.Pop();
          cout << temp << endl;
     }
return 0;
}
```

Այժմ հաջորդական պահունակն իրականացնենք LIFO դասի շաբլոնի միջոցով, որտեղ շաբլոնի պարամետրեր են համարվում պահունակի տարրերի տիպը և պահունակի տարրերը պահելու համար նախատեսված զանգվածի չափը: Լցված պահունակին նոր տարր ավելացնելու և դատարկ պահունակից տարր հեռացնելու փորձեր անելու դեպքում գեներացվում է բացառիկ իրավիճակ:

```
template<class T, int maxsize>
class LIFO    {
private:
       T st_mas[ maxsize] ;
       int top; // պահունակի գագաթ
public:
       LIFO():top(-1){}
       ~LIFO(){}
       bool IsEmpty(){ return top==-1;}
       bool IsFull(){ return top==maxsize-1;}
       void push(T item);
       void pop();
       T& getTop();
       void print();
};
```

Ստորև բերված է LIFO դասի շաբլոնի իրականացումը:

```
template <class T, int maxsize>
void LIFO <T, maxsize>::push(T item) {
       if(IsFull())
              throw "LIFO is full";
       else
              st_mas[ ++top] =item;
}
template <class T, int maxsize>
void LIFO <T, maxsize>::pop(){
       if(IsEmpty())
              throw "LIFO is empty";
       else
              top--;
}
template <class T, int maxsize>
T& LIFO <T, maxsize>:: getTop(){
       if(IsEmpty())
              throw "LIFO is empty";
       return st_mas[ top] ;
}
template <class T, int maxsize>
void LIFO <T, maxsize>:: print(){
       for(int i=0;i<top+1;i++)
              cout<<st_mas[ i] ;
       cout<<endl;
}
```

Դիտարկենք պահունակի օգտագործման հետևյալ օրինակը:

```
int main(){
    try{
       LIFO <int,5> x;
       cout<<"Is empty? "<<x.IsEmpty()<<endl;
       x.push(2); x.push(3); x.print();
       cout<<"Is empty? "<<x.IsEmpty()<<endl;
       x.pop(); x.print();
       cout<<"top "<<x.getTop()<<endl;
       LIFO <char *,50> y;
       cout<<y.IsEmpty()<<endl;
       y.push("First element"); y.push("First element");
       cout<<y.IsEmpty()<<endl;
       cout<<"top "<<y.getTop()<<endl;
    }catch(LIFO_Exception & a) {
       a.show();
    }
    return 0;
}
```

Պահունակի իրականացում կապակցված ցուցակի միջոցով: Իրականացնենք պահունակը կապակցված ցուցակի միջոցով: Կառուցենք LIFO դասի շաբլոնը, որտեղ որպես շաբլոնի պարամետր համարում ենք պահունակի տարրերի տիպը: Ցուցակի հանգույցը նկարագրվում է Node տիպով: Լցված պահունակին նոր տարր ավելացնելու և դատարկ պահունակից տարր հեռացնելու փորձեր անելու դեպքում գեներացվում է բացառիկ իրավիճակ, որը ներկայացվում է LIFO_Exception օբյեկտի միջոցով:

```
#include <iostream>
#include <string>
#include <exception>
using namespace std;
// LIFO_Exception դաս
class LIFO_Exception: public exception{
private:
    char * name;
public:
    LIFO_Exception(){}
    LIFO_Exception(char *s) {
         name=new char[ strlen(s)+1] ;
         strcpy(name,s);
    }
    ~LIFO_Exception(){
         delete [ ]name;
    }
    void show()const{
         cout<<name<<endl;
    }
} ;

//LIFO դասի շաբլոն
template<class T>
struct Node{
    T info;
    Node <T>* link;
} ;

template<class T>
class LIFO{
private:
     Node<T>* top; // top-ը ցուցիչ է պահունակի գագաթի տարրի
                   // վրա
```

```
public:
    LIFO():top(0){}
    ~LIFO(){}
    bool IsEmpty(){return top==0;}
    void push(T item);
    void pop();
    T getTop();
    void print();
};

template <class T>
void LIFO <T>::push(T item) {
    Node<T>* ptr=new Node<T>; //uswGwL Gnp hwGqnujg Ynujshg
    if (!ptr)
           throw LIFO_Exception("Overload in heap");
    ptr->info=item;
    ptr->link=top;
    top=ptr;
}
template <class T>
T LIFO <T>:: getTop(){
    if(IsEmpty())
           throw LIFO_Exception("LIFO is empty");
    return top->info;
}

template <class T>
void LIFO <T>:: pop(){
    if(IsEmpty())
           throw LIFO_Exception("LIFO is empty");
    Node<T>* ptr=top;
    top=top->link;
    delete ptr;
    return;
}

template <class T>
void LIFO <T>::print(){
    Node<T>* ptr=top;
    while(ptr!=0) {
           cout<<ptr->info<<' ';
           ptr=ptr->link;
    }
    cout<<endl;
}
```

Օգտագործենք պահունակը հետևյալ օրինակում:

```
int main(){
    try{
          LIFO <int> x;
          cout<<"Is empty? "<<x.IsEmpty()<<endl;
          x.push(2); x.push(3);
          x.push(4);x.push(5);
          x.print();
          cout<<"Is empty? "<<x.IsEmpty()<<endl;
          x.pop();
          x.print();
    }catch(LIFO_Exception & a{
          a.show();
    }
    return 0;
}
```

3.4. ՊԱՀՈՒՆԱԿԻ ԻՐԱԿԱՆԱՑՈՒՄԸ ՇԱԲԼՈՆՆԵՐԻ ՄԻՋՈՑՈՎ ՍՏԱՆԴԱՐՏ ԳՐԱԴԱՐԱՆՈՒՄ

STL գրադարանում պահունակն իրականացված է stack կոնտեյների միջոցով: Այն համարվում է հարմարեցված կոնտեյներ (կոնտեյների ադապտեր), քանի որ թույլ է տալիս vector, deque, list հաջորդական կոնտեյներների վրա հիմնված իրականացումներ:

STL-ի stack դասը շաբլոնային է: Այն ունի շաբլոնի երկու պարամետրեր: Առաջին պարամետրով տրվում է պահունակի տարրերի տիպը, երկրորդ պարամետրը նշում է այն հիմնական հաջորդական կոնտեյները, որն օգտագործվում է պահունակի իրականացման համար: Այդ պարամետրի լռությամբ տրվող արժեքը deque–ն է:

Ստորև ներկայացված է STL-ի stack դասի շաբլոնը:

```
template<class T, class Cont = deque<T> >
class stack {
public:
    typedef Cont::allocator_type allocator_type;
    typedef Cont::value_type value_type;
    typedef Cont::size_type size_type;
    explicit stack(const allocator_type&
              al=allocator_type()) const;
    bool empty() const;
    size_type size() const;
    allocator_type get_allocator() const;
```

```
    value_type& top();
    const value_type& top() const;
    void push(const value_type& x);
    void pop();
protected:
    Cont c;
};
```

Օգտագործենք stack կոնտեյները ծրագրում, որը թվեր է ներմուծում ֆայլից և արտածում դրանք էկրանին հակառակ կարգով:

```
#include <iostream>
#include <stack>
#include <fstream>
using namespace std;

int main(){
    ifstream in("inpnum.txt");
    stack<int,list<int> >s;// հայտարարվում է list-ի վրա
                           // հիմնված պահունակ
    int x;
    while(in>>x)
          s.push(x);
    while(!s.empty()){
          cout<<s.top<<' ';
          s.pop();
    }
    return 0;
}
```

3.5. ՊԱՀՈՒՆԱԿԻ ԿԻՐԱՌՈՒԹՅՈՒՆՆԵՐ

Ինչպես արդեն նշվել է, պահունակը կիրառվում է թվաբանական արտահայտությունների ներկայացման տարբեր ձևերն իրականացնելիս, մի ձևից մյուսին անցնելիս, արտահայտությունների արժեքները հաշվելիս և այլն:

Դիտարկենք թվաբանական արտահայտությունների ներկայացման միջածանցային (infix), նախածանցային (prefix) և վերջածանցային (postfix) ձևերը:

Միջածանցային ձև: Արտահայտության ներկայացման միջածանցային ձևում գործողության նշանը դրվում է օպերանդների միջև, օրինակ՝ 2+3*5:

Արտահայտությունների ներկայացման միջածանցային ձևում կարևորվում են գործողությունների նախապատվությունները (priority), որոնք որոշում են գործողությունների կատարման կարգը: Որոշ գործողություններ ունեն համարժեք նախապատվություններ: Այդ դեպքում դրանք կատարվում են ձախից դեպի աջ կամ աջից դեպի ձախ: Ստորև թվարկվում են գործողությունները ըստ նախապատվությունների նվազման.

- աստիճան բարձրացնելու գործողություն (↑)
- բազմապատկման(*) և բաժանման(/) գործողություններ
- գումարման(+) և հանման(-) գործողություններ

Գործողությունների կատարման կարգը փոխում են փակագծերի միջոցով:

Նախածանցային ձև: Այս դեպքում գործողության նշանը դրվում է օպերանդներից առաջ: Դիցուք AωB միջածանցային տիպի արտահայտություն է, որտեղ ω-ն գործողության նշան է, իսկ A-ն և B-ն այդ գործողության օպերանդներն են:

prefix (AωB) = ωprefix(A)prefix(B)
prefix (A) = A, որտեղ A-ն թիվ է կամ փոփոխական:
Օրինակ.
prefix(a+b)= +ab
prefix(a+b*c)= +a*bc
prefix(a*(b+c)↑2-5) = -prefix(a*(b+c)↑2)prefix(5)=
- * prefix(a)prefix((b+c)↑2)5 = - * a↑prefix(b+c)prefix(2)5 = - * a↑+bc25

Վերջածանցային ձև: Արտահայտության ներկայացման վերջածանցային ձևում օպերանդները նախորդում են գործողություններին: Այդ գրելաձևը անվանում են նաև լեհական ներկայացում՝

RPN (Reverse Polish Notation):
postfix(AωB) = postfix(A)postfix(B)ω,
postfix(A) = A, որտեղ A-ն թիվ է կամ փոփոխական:

Օրինակ.

postfix(a+b)= ab+
postfix(a+b*c)= abc*+
postfix(a*(b+c)↑2-5) = postfix(a(b+c)↑2)postfix(5) - = postfix(a)
postfix((b+c)↑2)*5- = aposfix(b+c)postfix(2)↑*5 - = abc+2↑*5-

Նախածանցային և վերջածանցային ձևերում փակագծեր չեն օգտագործվում:

Արտահայտության ձևափոխումը միջածանցային ձևից վերջածանցային ձևի

Նկարագրենք արտահայտության միջածանցային ձևից վերջածանցայինին ձևափոխման ալգորիթմը:

Միջածանցային արտահայտությունը տրվում է S տողով, որը կարող է կարդացվել ստեղնաշարից կամ տեքստային ֆայլից: Ալգորիթմի աշխատանքի արդյունքում վերջածանցային ձևը ստացվում է U տողում, որը կարող է արտածվել էկրանին կամ տեքստային ֆայլում: priority(ω) ֆունկցիան վերադարձնում է ω գործողության նախապատվությունը: Օգտագործվում է գործողությունների opstk պահունակը:

Մուտք - միջածանցային արտահայտությունը S տողում
Ելք - վերջածանցային արտահայտությունը U տողում

Քայլ 1.
[սկզբնարժեքավորում] դատարկել opstk պահունակը, ստեղծել դատարկ U տող

Քայլ 2.
[հերթական սիմվոլի ներմուծում] S-ից կարդացվող հերթական
սիմվոլը գրվում է c փոփոխականում S → c;

Քայլ 3.
[մուտքի վերջ]
եթե (S–ի բոլոր սիմվոլները կարդացվել են)
ապա անցնել *Քայլ 5*

Քայլ 4.
[c-ի ստուգում]
եթե (c-ն օպերանդ է)
ապա {c → U (c-ն գրել U-ում), անցնել *Քայլ 2*}
եթե (c-ն հավասար է ' (')
ապա {c ⇒ opstk, անցնել *Քայլ 2*}
եթե (c-ն գործողության նշան է)

```
        ապա
    {քանի դեռ (opstk-ը դատարկ չէ և գագաթի տարրը հավասար
    չէ ' (' )
        { x ⇐ opstk;
        եթե (priority(x) ≥ priority(c))
            ապա x→U
            այլապես {x ⇒ opstk, ելք gիկլից }
        }
        c ⇒ opstk;
    }
    եթե (c-ն հավասար է ' ) ')
        ապա { x ⇐ opstk
            քանի դեռ (x ≠ '( ')
                { x → U, x ⇐ opstk }
            }
        անցնել Քայլ 2
Քայլ 5.
[դատարկել opstk -ը]
        քանի դեռ (opstk != Λ)
            { x⇐ opstk, x → U }
```

Ալգորիթմը կիրառենք օրինակների վրա:

*Օրինակ1. A+B*C*

Ստորև բերված են վերջածանցային U տողի և opstk-ի պարունակությունները միջածանցային S տողի յուրաքանչյուր սիմվոլ դիտարկելուց հետո:

կարդացվող սիմվոլի համարը	կարդացվող սիմվոլ	վերջածանցային U տող	գործողությունների opstk պահունակ
1	A	A	
2	+	A	+
3	B	AB	+
4	*	AB	+*
5	C	ABC	+*
6		ABC*	+
7		ABC*+	

*Օրինակ 2. ((A-(B+C))*D)↑(E+F)*

կարդացվող սիմվոլի համարը	կարդացվող սիմվոլ	վերջածանցային U տող	գործողությունների opstk պահունակ
1	(		(
2	(		((
3	A	A	((
4	-	A	((-
5	(	A	((-(
6	B	AB	((-(
7	+	AB	((-(+
8	C	ABC	((-(+
9	)	ABC+	((-
10	)	ABC+-	(
11	*	ABC+-	(*
12	D	ABC+-D	(*
13	)	ABC+-D*	
14	↑	ABC+-D*	↑
15	(	ABC+-D*	↑(
16	E	ABC+-D*E	↑(
17	+	ABC+-D*E	↑(+
18	F	ABC+-D*EF	↑(+
19	)	ABC+-D*EF+	↑
		ABC+-D*EF+↑	

Վերջածանցային ձևով տրված արտահայտության արժեքի հաշվումը

Դիտարկենք վերջածանցային ձևով տրված արտահայտության արժեքի հաշվման ալգորիթմը: Վերջածանցային տողում հանդիպած գործողությունը կատարվում է իրեն նախորդող երկու օպերանդների նկատմամբ: Օպերանդներից մեկը կարող է լինել նախորդ գործողության արդյունք:

Վերջածանցային արտահայտությունը տրվում է U տողով, որը կարող է կարդացվել ստեղնաշարից կամ տեքստային ֆայլից: Օգտագործվում է օպերանդների stk պահունակը: U-ից կարդացվող հերթական օպերանդը գրվում է stk պահունակում: Եթե U-ից կարդացվում է գործողության նշան, ապա գործողությունը կատարվում է պահունակի գագաթի երկու օպերանդների նկատմամբ և ստացված արդյունքը հիշվում է պահունակում:

Ալգորիթմի աշխատանքի արդյունքում պահունակում ստացվում է տրված արտահայտության արժեքը:

Մուտք - վերջածանցային արտահայտությունը U տողում
Ելք - արտահայտության արժեքը x փոփոխականում
Քայլ 1. [սկզբնարժեքավորում]
դատարկել stk պահունակը
Քայլ 2. [հերթական սիմվոլի ներմուծում U տողից]
U → c
Քայլ 3. [մուտքի վերջ]
եթե (U-ի բոլոր սիմվոլները կարդացվել են)
ապա անցնել *Քայլ 5*
Քայլ 4. [սիմվոլի մշակում]
եթե (c-ն օպերանդ է)
ապա c ⇒ stk
եթե (c-ն ω գործողության նշան է)
ապա
{ x ⇐ stk, y ⇐ stk, z=y ω x, z ⇒ stk }
անցնել *Քայլ 2*
Քայլ 5. [ավարտ]
x ⇐ stk

Օրինակ: Հաշվել 623+-382/+*2↑3+ վերջածանցային արտահայտության արժեքը: Ենթադրվում է, որ օպերանդները միանիշ թվեր են:

կարդացվող սիմվոլ	օպերանդ 1	օպերանդ 2	ընթացիկ արժեք	stk պահունակ
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
↑	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

ԽՆԴԻՐՆԵՐ

1. * Իրականացնել պահունակը դինամիկ զանգվածի միջոցով:
2. Իրականացնել պահունակ, դինամիկ զանգվածի օգնությամբ: Պահունակի գերհագեցման դեպքում կրկնակի ավելացնել զանգվածի չափը:
3. Իրականացնել երկու պահունակների DoubleStack դասը, որն օգտագործում է զանգված: Մի պահունակը տեղադրվում է զանգվածի սկզբից, իսկ մյուսը՝ վերջից: Պահունակների գագաթների ինդեքսները տրվում են Top1 և Top2 ամբողջ տիպի փոփոխականներով: Այս երկու պահունակների համար իրականացնել պահունակներին բնորոշ բոլոր գործողությունները:
4. * Տրված են ամբողջ թվերի S1, սիմվոլների S2 և իրական թվերի S3 պահունակները: Մեկումեջ տպել այդ պահունակների տարրերը մինչև բոլոր պահունակների դատարկվելը:
5. Տրված են S1 և S2 պահունակները: Կառուցել S3 պահունակը, որի հատակում S1-ի տարրերն են, իսկ վերևում դասավորված են S2-ի տարրերը՝ շրջված կարգով:
6. Տրված է S1 պահունակը: Ստանալ S2 պահունակը, որում S1-ի տարրերը դասավորված են չնվազման կարգով:
7. Տրված են S1 և S2 պահունակները: Կառուցել S3 պահունակը, որի մեջ մեկումեջ դրվում են S1 և S2 պահունակների տարրերը, և վերջում ավելացվում են չդատարկված պահունակի տարրերը:
8. Տրված է S1 պահունակը: Ստանալ S2 պահունակը, որում S1-ի տարրերն են առանց կրկնումների:
9. Պահունակի օգտագործմամբ որոշել տրված սիմվոլային տողը պալինդրո՞մ է, թե՞ ոչ: Պալինդրոմ է կոչվում այն տողը, որը սկզբից և վերջից կարդացվում է նույն ձևով:
10. Պահունակի օգտագործմամբ տրված բնական թիվը տասական համակարգից բերել p-ական (p>1) համակարգի:
11. * Թվաբանական արտահայտությունը ներկայացվում է նիշերի հաջորդականության միջոցով: Ստուգել նրանում երեք տեսակի { }, [ ], ( ) փակագծերի ներդրվածության ճշտությունը:
12. Միանիշ թվեր պարունակող միջածանցային տեսքով ներկայացված շարահյուսորեն ճիշտ թվաբանական արտահայտության համար ստանալ նրա վերջածանցային ձևը:
13. Միանիշ թվեր պարունակող միջածանցային տեսքով ներկայացված շարահյուսորեն ճիշտ թվաբանական արտահայտության համար ստանալ նրա նախածանցային ձևը:
14. Միջածանցային տեսքով ներկայացված թվաբանական արտահայտության համար ստանալ նրա վերջածանցային ձևը:
15. Հաշվել միանիշ թվեր պարունակող վերջածանցային տեսքով ներկայացված թվաբանական արտահայտության արժեքը:
16. Հաշվել միանիշ թվեր պարունակող նախածանցային տեսքով ներկայացված թվաբանական արտահայտության արժեքը:

ԳԼՈՒԽ 4.

ՀԵՐԹ

4.1. ՀԵՐԹ ՏՎՅԱԼՆԵՐԻ ԱԲՍՏՐԱԿՏ ՏԻՊԸ

Հերթը (Queue) գծային ցուցակ է, որտեղ տվյալների հասանելիությունն իրականացվում է ցուցակի երկու եզրերից: Տարրն ավելացվում է հերթի վերջում (հերթի վերջ՝ rear) և հեռացվում է հերթի սկզբից (հերթի սկիզբ՝ front): Հերթից տարրերը հեռացվում են այն կարգով ինչպես պահված են, հետևաբար հերթը ապահովում է FIFO կարգ (first-in / first-out՝ առաջինը-մտավ / առաջինը-ելավ):

ADT Queue

Տվյալներ

տարրերի ցուցակ
front – հերթում առաջին տարրի դիրքը
rear – հերթում վերջին տարրին հաջորդող դիրքը
count – հերթի տարրերի քանակը

Գործողություններ

Կոնստրուկտոր

մուտք - չկա
նախապայման – չկա
պրոցես - հերթի սկզբնարժեքավորում
վերջնապայման – դատարկ հերթ

Length

մուտք – չկա
նախապայման – չկա
պրոցես – հերթի տարրերի քանակի որոշում
ելք – հերթի տարրերի քանակը
վերջնապայման – չկա

Empty

մուտք – չկա
նախապայման – չկա

պրոցես – հերթի դատարկ լինելու ստուգում
ելք – true, եթե հերթը դատարկ է, false՝ հակառակ դեպքում
վերջնապայման – չկա

Insert

մուտք - հերթին ավելացվող տարրը
նախապայման – չկա
պրոցես – տարրի ավելացում հերթին
ելք – չկա
վերջնապայման – ավելացված տարրով հերթ
բացառիկ իրավիճակ – տարրի ավելացում լցված հերթին

Delete

մուտք – չկա
նախապայման – հերթը դատարկ չէ
պրոցես – տարրի հեռացում հերթից
ելք – չկա
վերջնապայման – հեռացված տարրով հերթ
բացառիկ իրավիճակ – տարրի հեռացում դատարկ հերթից

Front

մուտք – չկա
նախապայման - հերթը դատարկ չէ
պրոցես – հերթի սկզբի տարրի ընտրում
ելք – հերթի սկզբի տարրը
վերջնապայման – չկա
բացառիկ իրավիճակ – տարրի ընտրում դատարկ հերթից

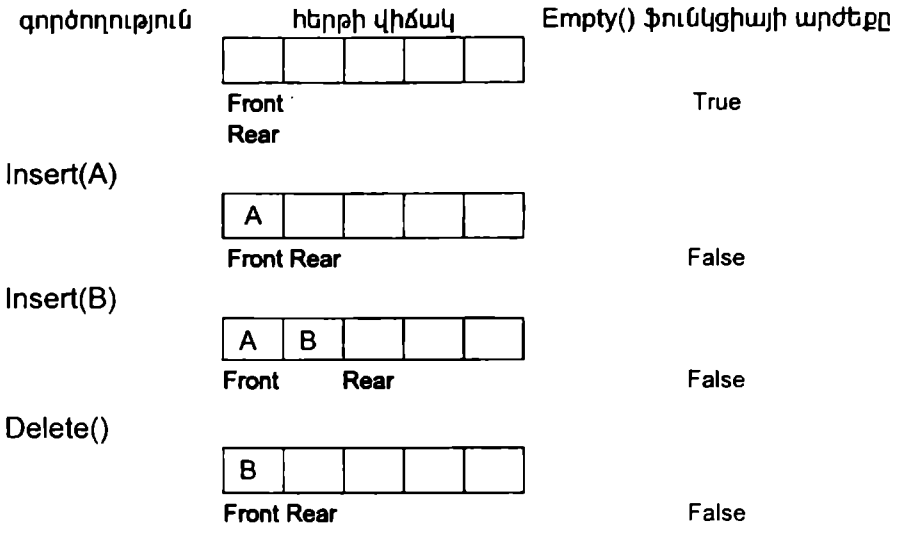
ClearQueue

մուտք – չկա
նախապայման - չկա
պրոցես – հերթի բոլոր տարրերի հեռացում
ելք – չկա
վերջնապայման – դատարկ հերթ

վերջ ADT Queue

Օրինակ:

Նկարագրենք հերթի փոփոխությունը նշված գործողություննե-րի իրականացումից հետո (նկ. 4.1):



Նկ. 4.1 Հերթի վիճակը նշված գործողություններից հետո

Երկկողմանի հերթը (դեկ, DeQueue) այնպիսի գծային ցուցակ է, որում տարրերի և ավելացման, և հեռացման գործողությունները կատարվում են երկու ծայրերից: Օգտագործում են նաև դեկի մասնավոր դեպքեր՝ դեկ սահմանափակ մուտքով (տվյալների ավելացումը կատարվում է միայն մի ծայրից) և դեկ սահմանափակ ելքով (տվյալների հեռացումը կատարվում է միայն մի ծայրից):

4.2 ՀԵՐԹԻ ԻՐԱԿԱՆԱՑՈՒՄ

Հերթը, ինչպես և գծային ցուցակը, կարելի է իրականացնել երկու եղանակով՝ զանգվածի և կապակցված ցուցակի միջոցով: Զանգվածի միջոցով հերթի իրականացումը արդյունավետ չէ, քանի որ հերթից տարր հեռացնելիս, եթե հերթի սկզբի ցուցիչը մնում է անփոփոխ, ապա պահանջվում է մնացած տարրերի տեղաշարժ:

Եթե հերթի սկզբի ցուցիչը տարր հեռացնելիս փոփոխվում է, ապա հերթի հետ աշխատելիս կարող է ստեղծվել գերհագեցման իրավիճակ, մինչդեռ հերթն իրականացնող զանգվածի սկզբում կարող են լինել ազատ դիրքեր:

Հերթի արդյունավետ իրականացում կարելի է ապահովել ցիկլիկ զանգվածի օգնությամբ: Այս դեպքում տարրերի հեռացման արդյունքում մնացած տարրերի տեղաշարժ չի կատարվում և ազատված դիրքերի կրկին օգտագործման շնորհիվ, որոշ իմաստով լուծվում է հերթի գերհագեցման հարցը: Տարրի հեռացման/ավելացման դեպքում front և rear փոփոխականները փոփոխվում են ցիկլիկ ձևով:

Հերթի իրականացում ցիկլիկ զանգվածի միջոցով

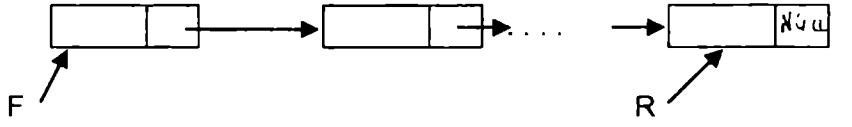
Ենթադրենք, որ հերթը ներկայացված է $x_1, \ldots, x_m$ զանգվածով ($m \geq 1$), որտեղ x_m -ին հաջորդում է x_1-ը (ցիկլիկ զանգված): front-ը հերթի առաջին տարրի դիրքն է, rear-ը՝ վերջին տարրին հաջորդող դիրքը, count-ը հերթի տարրերի քանակն է:

Նկարագրենք հերթի հետ կատարվող գործողությունները:

1. Empty() - ստուգել հերթը դատարկ է, թե ոչ:
   եթե (count = 0)
   ապա TRUE
   այլապես FALSE
2. Insert(item) - item տարրն ավելացնել հերթին:
   եթե (count = m)
   ապա OVERFLOW
   այլապես { x[rear] ← item,
   եթե (rear≠ m) *ապա* rear ← rear + 1
   այլապես rear ← 1,
   count ← count +1 }
3. Delete() - հերթից տարր հեռացնել:
   եթե (count = 0)
   ապա UNDERFLOW
   այլապես {
   եթե (front≠ m) *ապա* front ← front + 1
   այլապես front ←1,
   count ← count – 1}
4. GetFirst(item) – ստանալ հերթի սկզբի տարրը:
   եթե (count = 0)
   ապա UNDERFLOW
   այլապես item ← x[front]
5. ClearQueue() - հերթը դատարկել:
   count ← 0, front ← 1, rear← 1:

Հերթի իրականացում կապակցված ցուցակի միջոցով

Այս դեպքում հերթը կազմված է առանձին հանգույցներից և ունի երկու ցուցիչ՝ F և R, որոնք համապատասխանաբար նշում են հերթի առաջին և վերջին տարրերը:



Նկարագրենք հերթի հետ կատարվող որոշ գործողություններ:

1. Empty() - ստուգել հերթը դատարկ է, թե ոչ:
   եթե (F =NULL)
   ապա TRUE
   այլապես FALSE

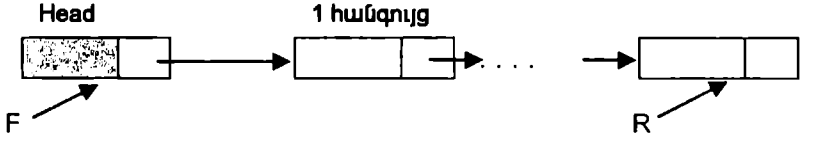
2. Insert(item) - item տարրն ավելացնել հերթին:
   x ⇐ AVAIL, INFO(x) ← item, LINK(x) ← NULL
   եթե (F ≠ NULL)
   ապա { LINK(R) ← x, R ← x}
   այլապես {R ← x, F ← x}

3. Delete() - հերթից տարր հեռացնել:
   եթե (F = NULL)
   ապա UNDERFLOW
   այլապես {x ← F, F ← LINK(F),
   եթե (F = NULL) *ապա* R ← NULL,
   x ⇒ AVAIL}

4. ClearQueue() - հերթը դատարկել:
   եթե (F = NULL)
   ապա NOP
   այլապես { LINK(R) ← AVAIL, AVAIL ← F, F ← NULL,
   R ← NULL }

Հերթի իրականացում վերնագրային հանգույցով կապակցված ցուցակի միջոցով

Հերթը ներկայացնող կապակցված ցուցակը կարող է ունենալ վերնագրային հանգույց (Head):



Այսպիսի հերթերում վերնագրային հանգույցի INFO դաշտում գրված ինֆորմացիան անտեսվում է, իսկ դատարկ հերթը կազմված է լինում միայն վերնագրային հանգույցից, որի LINK դաշտում գրված է NULL:

Վերնագրային հանգույցով հերթի հետ կատարվող գործողություններն ունեն հետևյալ իրականացումները:

1. Empty() - ստուգել հերթը դատարկ է, թե ոչ:
   եթե (F = R)
   ապա TRUE
   այլապես FALSE

2. Insert(item) - item տարրն ավելացնել հերթին:
   x ⇐ AVAIL, INFO(x) ← item, LINK(x) ← NULL, LINK(R) ← x,
   R ← x

3. Delete() - հերթից տարր հեռացնել:
   եթե (F = R)
   ապա UNDERFLOW
   այլապես {x←LINK(F), LINK(F)←LINK(x),
   եթե (x = R) *ապա* { R←F}
   x⇒AVAIL}

4. ClearQueue() - հերթը դատարկել:
   եթե (F = R)
   ապա NOP
   այլապես { LINK(R) ← AVAIL, AVAIL ← LINK(F),
   LINK(F) ← NULL, R ← F}

4.3 Հերթի իրականացումը C++ լեզվով

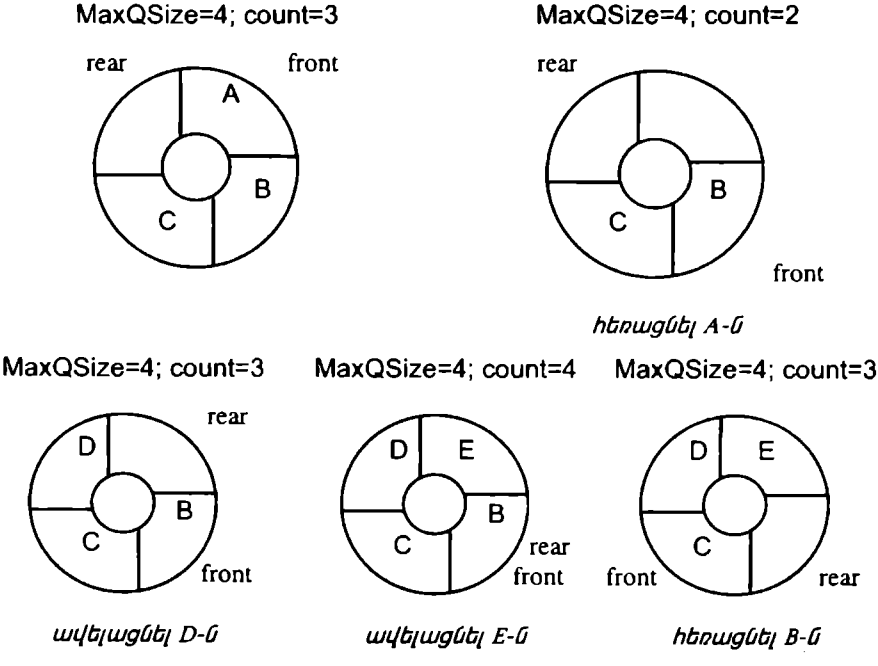
Իրականացնենք հերթը C++ լեզվով ըստ հերթի շրջանածև մոդելի և վերնագրային հանգույցով գծային ցուցակի:

Հերթի իրականացում շրջանածև մոդելում

Շրջանածև մոդելում հերթն իրականացվում է ցիկլիկ զանգվածի միջոցով:

Զանգվածի տարրերի քանակը տրվում է MaxQSize հաստատունով: front-ը (0<=front< MaxQSize) ցույց է տալիս հերթի առաջին տարրի դիրքը և տարրի հեռացման դեպքում այն շարժվում է շրջանագծով: rear-ը (0<=rear< MaxQSize) ցույց է տալիս հերթի մեջ տարր ավելացնելու դիրքը: Տարր ավելացնելուց հետո rear-ը նույնպես շարժվում է շրջանագծով: count-ը հերթի տարրերի քանակն է: Երբ count=MaxQSize՝ հերթը լիքն է:

rear և front փոփոխականները փոխվում են հետևյալ կերպ՝
rear = (rear+1) % MaxQSize, front = (front+1) % MaxQSize:



Նկ. 4.2 Հերթի շրջանածև մոդել

Կազմենք Queue դասը, որն իրականացնում է հերթը օգտագործելով qlist ցիկլիկ զանգված: front, rear փոփոխականները որոշում են հերթի սկիզբը և վերջը, իսկ count-ը՝ հերթի տարրերի քանակը: DataType պարամետրացված տիպը հնարավորություն է տալիս հերթն իրականացնել տարբեր տիպի տվյալների համար:

Queue դասի հայտարարումը կատարենք aqueue.h ֆայլում:

```
// հերթն իրականացնող զանգվածի չափը
const int MaxQSize =50;
class Queue{
private:
      // փոփոխականներ, զանգված
      int front, rear, count;
      DataType qlist[ MaxQSize] ;
public:
      // կոնստրուկտոր
      Queue();
      // հերթի փոփոխման գործողություններ
      void QInsert (const DataType & item);
      DataType QDelete();
      void ClearQ();
      // հերթի հասանելիության գործողություն
      DataType QFront() const;
      // հերթի վիճակի ստուգման գործողություններ
      int QEmpty() const;
      int QFull() const;
      int QLength() const;
};
```

Բերենք հերթի օգտագործման օրինակ:

```
typedef int DataType;
#include "aqueue.h"      // ավելացնել Queue դասի
                            նկարագիրը
int main()
{
    Queue Q;             // հայտարարել Queue տիպի
                         // օբյեկտ
    Q.QInsert(30);       // 30-ը ավելացնել հերթին
    Q.QInsert(70);       // 70-ը ավելացնել հերթին
```

```
    cout<<Q.QLength()<<endl;    // տպել հերթի տարրերի
                                // քանակը՝ 2
    cout<<Q.QFront()<<endl;     // տպել հերթի առաջին
                                // տարրը՝ 30
    if ( !QEmpty ( ) )
       cout<<Q.QDelete()<<endl; // տպել 30 և այն
                                // հեռացնել հերթից
    cout<<Q.QFront()<<endl;     // տպել 70
    Q.ClearQueue();             // դատարկել հերթը
    return 0;
}
```

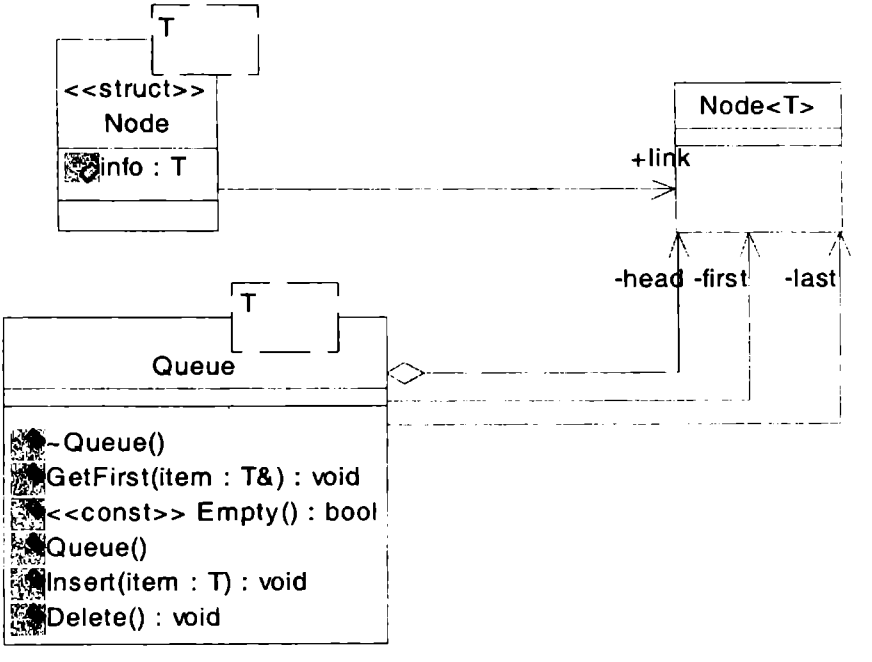
Queue դասի իրականացումը կատարենք aqueue.cpp ֆայլում:

```
// Queue() կոնստրուկտորը ստեղծում է դատարկ հերթ
Queue::Queue(void): front(0), rear(0), count(0) { }

//QInsert()-ը DataType տիպի item տարրն ավելացնում է
//հերթի վերջում
void Queue::QInsert (const DataType & item) {
    // եթե հերթը լիքն է ավարտել ծրագիրը
    if (count==MaxQSize) {
         cerr<< "Հերթը հագեցված է "<< endl;
         exit(1);
    }
    // ավելացնել count-ը, զանգվածում գրել item-ը,
    // փոխել rear-ը
    count++;
    qlist[ rear] =item;
    rear=(rear+1)%MaxQSize;
}

// QDelete()-ը հերթից հեռացնում է տարր և
// վերադարձնում է նրա արժեքը
DataType Queue::QDelete(){
    DataType temp;
    // եթե հերթը դատարկ է ավարտել ծրագիրը
    if(count == 0) {
         cerr <<"Հեռացում դատարկ հերթից" <<endl;
         exit(1);
    }
    // փոքրացնել count-ը, տեղաշարժել հերթի սկիզբը և
    // հեռացված տարրը վերադարձնել
```

```
    temp=qlist[ front] ;
    count--;
    front=(front+1)% MaxQSize;
    return temp;
}
```

Վերնագրային հանգույցով հերթի իրականացում

Այժմ կառուցենք Queue դասի շաբլոնը, որը մոդելավորում է կապակցված գծային ցուցակով իրականացված վերնագրային հանգույցով հերթ: Որպես շաբլոնի պարամետր համարենք հերթի տարրերի տիպը: Ստորև ներկայացված է Queue դասի շաբլոնի UML-դիագրամը:



Նկարագրենք հերթի հանգույցների Node տիպը:

```
template <class T>
struct Node {
    T info;
    Node<T> *link;
};
```

Queue դասի հայտարարում:

```
template <class T>
class Queue{
private:
    Node <T> head;
    Node <T> *first;
    Node <T> *last;
public:
    Queue(): first(&head), last(&head) { }
    ~ Queue ();
    void Insert (T item);
    void Delete ();
    void GetFirst (T & item);
    bool Empty() const;
};
```

Queue դասի իրականացում:

```
// Insert()-ը item-ն ավելացնում է հերթին
template <class T>
void Queue <T>::Insert (T item) {
    Node <T> *p = new Node<T>;
    if (!p) throw "Overflow in heap";
    p→info = item;
    p→link = 0;
    last→link = p;
    last = p;
}

// Delete()-ը հերթից հեռացնում է տարր
template <class T>
void Queue <T>::Delete(){
    if (Empty())
        throw "Empty Queue";
    Node <T> *p = head.link;
    head.link = p→link;
    delete p;
    if (head.link = = 0) {
        last =&head; head.link=NULL;
    }
    return;
}
```

```
// Empty()-ը ստուգում է հերթի դատարկ լինելը
template <class T>
bool Queue <T>:: Empty ()const{
    return last == &head;
}

// GetFirst()-ը վերադարձնում է հերթի առաջին տարրի
// արժեքը
template <class T>
void Queue <T>::GetFirst (T &item) {
    if (Empty())
        throw "Empty Queue";
    item = head.link→info;
    return;
}

// ~Queue() դեստրուկտորն ազատում է հերթի զբաղեցրած
// տիրույթը
template <class T>
Queue <T>:: ~ Queue(){
    Node <T> *p;
    if (first != last) {
        p = first→link;
        while (p) {
            first→link = p→link;
            delete (p);
            p = first→link;
        }
        last = first;
    }
}
```

Բերենք Queue դասի շաբլոնն օգտագործող ծրագրի օրինակ:

```
#include <iostream.h>
#include "Queue.h"
int main(){
    Queue <int> IQ;          // ստեղծվում է int տիպի
                             // տարրերից կազմված հերթ
    Queue <char*> CQ;        // ստեղծվում է ցուցիչներ
                             // պարունակող տարրերի հերթ
    IQ.Insert(30);           // 30-ը ավելացվում է հերթին
    IQ.Insert(70);           // 70-ը ավելացվում է հերթին
```

```
    CQ.Insert("Hello!");   // Hello! տողն ավելացվում
                           // է հերթին
    out << IQ.GetFirst() << endl;     // տպում է առաջինը
                                      // տարրը՝ 30
    if ( ! IQ.Empty ( ) )
         IQ.Delete();
    cout << IQ. GetFirst() << endl; // տպում է 70
    return 0;
}
```

Հերթի իրականացումը շաբլոնների ստանդարտ գրադարանում

STL գրադարանում հերթն իրականացված է queue կոնտեյների միջոցով: Հերթը, ինչպես և պահունակը, համարվում է հարմարեցված կոնտեյներ (կոնտեյների ադապտեր), և իրականացված է deque և list հաջորդական կոնտեյներների հիման վրա:

STL-ի queue դասը շաբլոնային է: Այն ունի շաբլոնի երկու պարամետրեր, որոնցով տրվում են հերթի տարրերի տիպը և այն հիմնական հաջորդական կոնտեյները, որն օգտագործվում է հերթի իրականացման համար: Այդ պարամետրի լռությամբ տրվող արժեքը deque -ն է:

Ստորև ներկայացված է STL-ի queue դասի շաբլոնը:

```
template<class T, class Cont = deque<T> >
class queue {
public:
    typedef Cont::allocator_type allocator_type;
    typedef Cont::value_type value_type;
    typedef Cont::size_type size_type;
    explicit queue(const allocator_type& al =
    allocator_type()) const;
    bool empty() const;
    size_type size() const;
    allocator_type get_allocator() const;
    value_type& top();
    const value_type& top() const;
    void push(const value_type& x);
    void pop();
    protected:
    Cont c;
};
```

4.4 ՆԱԽԱՊԱՏԿՈՒԹՅԱՄԲ ՀԵՐԹ

Ինչպես գիտենք հերթը տվյալների կառուցվածք է, որն ապահովում է տարրերի FIFO կարգ: Տարրերը հերթից հեռացվում են սկսած հերթի սկզբում դրված տարրից: Սակայն, հաճախ հարկ է լինում օգտագործել այնպիսի հերթ, որի տարրերն օժտված են նախապատվությամբ և հերթական հեռացվող տարրն ունի ամենաբարձր նախապատվությունը: Այդպիսի կառուցվածքը կոչվում է նախապատվությամբ հերթ (priority queue):

ADT Pqueue

Տվյալներ

Տարրերի ցուցակ

count – հերթի տարրերի քանակը

Գործողություններ

Կոնստրուկտոր

մուտք - չկա

նախապայման – չկա

պրոցես - հերթի սկզբնարժեքավորում

վերջնապայման – դատարկ հերթ

PQLength

մուտք – չկա

նախապայման – չկա

պրոցես – հերթի տարրերի քանակի որոշում

ելք – հերթի տարրերի քանակը

վերջնապայման – չկա

PQEmpty

մուտք – չկա

նախապայման – չկա

պրոցես – հերթի դատարկ լինելու ստուգում

ելք – true, եթե հերթը դատարկ է, false՝ հակառակ դեպքում

վերջնապայման – չկա

PQInsert

մուտք - հերթին ավելացվող տարրը

նախապայման – չկա

պրոցես – տարրի ավելացում հերթին

ելք – չկա

վերջապայման – ավելացված տարրով հերթ

բացառիկ իրավիճակ – նոր տարրի ավելացում լցված հերթին

PQDelete

մուտք – չկա

նախապայման – հերթը դատարկ չէ

պրոցես – ամենաբարձր նախապատվությամբ տարրի հեռացում հերթից

ելք – չկա

վերջապայման – հեռացված տարրով հերթ

բացառիկ իրավիճակ – տարրի հեռացում դատարկ հերթից

ClearPQ

մուտք – չկա

նախապայման - չկա

պրոցես – հերթի բոլոր տարրերի հեռացում

ելք – չկա

վերջապայման – դատարկ հերթ

վերջ ADT Pqueue

Նախապատվությամբ հերթի իրականացում զանգվածի միջոցով

Այստեղ նախապատվությամբ հերթն իրականացված է զանգվածի միջոցով, որի տարրերը Datatype տիպի են:

Pqueue դասի հայտարարում:

```
// նախապատվությամբ հերթի առավելագույն չափը
const int MaxPQSize = 50;
class Pqueue
{
private:
    // հերթը ներկայացնող զանգվածը և հերթի տարրերի
    // քանակը
    DataType pqlist [ MaxPQSize] ;
    int count;
public:
    // կոնստրուկտոր
    Pqueue ();
    // հերթի փոփոխման գործողություններ
    void PQInsert (const DataType & item );
    DataType PQDelete();
    void ClearPQ();
    //հերթի վիճակի ստուգման գործողություններ
```

```
    int PQEmpty() const;
    int PQFull() const;
    int PQLength() const;
};
```

Բերենք Pqueue դասն օգտագործող ծրագրի հատված: Ենթադրենք բարձր նախապատվություն ունեցող տարրը դա ամենափոքր արժեք ունեցող տարրն է:

```
typedef int DataType;          // հայտարարվում է հերթի
                               // տարրերի տիպը
Pqueue PQ;                     // ստեղծվում է դատարկ հերթ
PQ. PQInsert (20);             // հերթին ավելացվում է 20
PQ.PQInsert (10);              // հերթին ավելացվում է 10
cout<<PQ.PQLength()<<endl;     // տպվում է 2
N =PQ.PQDelete ();             // 10-ը հանվում է հերթից
                               // և վերագրվում N-ին
```

Pqueue դասի իրականացումը նման է Queue դասի իրականացմանը՝ բացառությամբ PQDelete գործողության: Այս մեթոդը հերթից հեռացնում է ամենաբարձր նախապատվությամբ տարրը (մեր դեպքում դա ամենափոքր արժեք ունեցող տարրն է), և վերադարձնում է նրա արժեքը:

```
DataType PQueue::PQDelete(){
    DataType min;
    int i, minindex = 0;
    if ( count > 0 ) {
         // pqlist զանգվածում գտնել փոքրագույն արժեքը
         // և դրա ինդեքսը
         min = pqlist [ 0] ;
         for( i = 1; i < count; i ++)
             if ( pqlist[  i ] < min ) {
                   min = pqlist [  i ];
                   minindex = i;
             }
         // հերթի վերջին տարրը տեղափոխել ամենափոքր
         // տարրի տեղը, հաշվիչը մեկով փոքրացնել
         pqlist[ minindex] = pqlist[ count -1] ;
         count --;
    }
```

```
    // եթե pqlist զանգվածը դատարկ է, ապա ավարտել
    // ծրագիրը
    else{
         cerr << "հեռացում դատարկ հերթից" << endl;
         exit (1);
    }
    return min;
}
```

4.5 ՀԵՐԹԻ ԿԻՐԱՌՈՒԹՅՈՒՆՆԵՐ

Խնդիր 1: Տոպոլոգիական կարգավորում:

Հաճախ հարկ է լինում կարգավորել տարրերի այնպիսի բազմություն, որի վրա սահմանված է մասնակի կարգավորվածության հարաբերություն: Այսինքն, բինար հարաբերությունը որոշված է բազմության տարրերի ոչ թե բոլոր, այլ միայն որոշ զույգերի համար: Այդպիսի կարգավորման խնդիր է տերմինաբանական բառարանի կառուցման խնդիրը, որը պահանջում է դասավորել տերմինները բառարանում այնպես, որ այն բոլոր տերմինները, որոնք օգտագործվում են տվյալ տերմինի սահմանման մեջ, տեղադրված լինեն բառարանում ավելի վաղ:

Այդպիսին է նաև ծրագրերի կատարման պլանավորման խնդիրը: Ենթադրվում է, որ տվյալ խնդիրը կարող է կատարվել միայն այն դեպքում, երբ ավարտված է որոշ խնդիրների կատարումը: Այս խնդիրներում առկա կարգավորումը անվանում են տոպոլոգիական կարգավորում:

Կասենք, որ տարրերի S բազմությունը մասնակի կարգավորված է ըստ $\prec=$ բինար հարաբերության, եթե այդ հարաբերությունը բավարարում է հետևյալ պայմաններին՝

եթե $x \prec= y, y \prec= z$, ապա $x \prec= z$,

$(\forall x, y, z \in S)$ (տրանզիտիվություն)

եթե $x \prec= y, y \prec= x$, ապա x = y,

$(\forall x, y \in S)$ (հակասիմետրիկություն)

$x \prec= x \, (\forall x \in S)$ (ռեֆլեքսիվություն)

Եթե տարրերի հավասարությունը չի դիտարկվում, ապա որպես մասնակի կարգավորվածության հատկություն համարում են $x \prec y$ հարաբերությունը ($x \prec= y$ և x ≠ y), որը բավարարում է հետևյալ պայմաններին՝

եթե $x \prec y, y \prec z$, ապա $x \prec z$, $(\forall x, y, z \in S)$

եթե $x \prec y$, ապա $y \not\prec x$, $(\forall x, y \in S)$

$x \not\prec x$ (∀ x ∈ S)

Դիցուք S-ը մասնակի կարգավորված վերջավոր բազմություն է (ըստ $\prec$ հարաբերության): Տոպոլոգիական կարգավորման խնդիրը կայանում է նրանում, որ պետք է կառուցել $a_1\ a_2. \ldots a_n$ հաջորդականությունը ($a_i \in S$, n =|S|) այնպես, որ, եթե $a_i \prec a_j$, ապա i<j:

Օրինակ՝ դիցուք S={1, 2, . . .,9} և

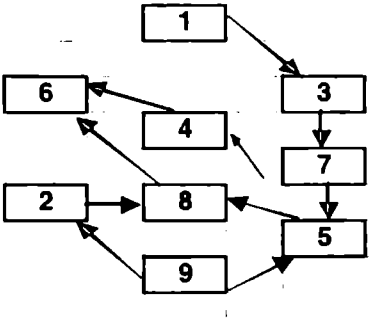
$1 \prec 3$ $5 \prec 8$

$3 \prec 7$ $4 \prec 6$ $9 \prec 2$

$7 \prec 5$ $8 \prec 6$ $2 \prec 8$

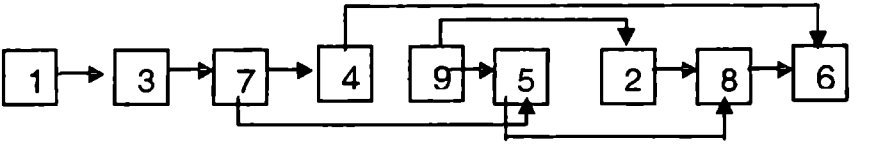
$7 \prec 4$ $9 \prec 5$

Պատկերենք S բազմությունը ուղղորդված գրաֆի տեսքով:

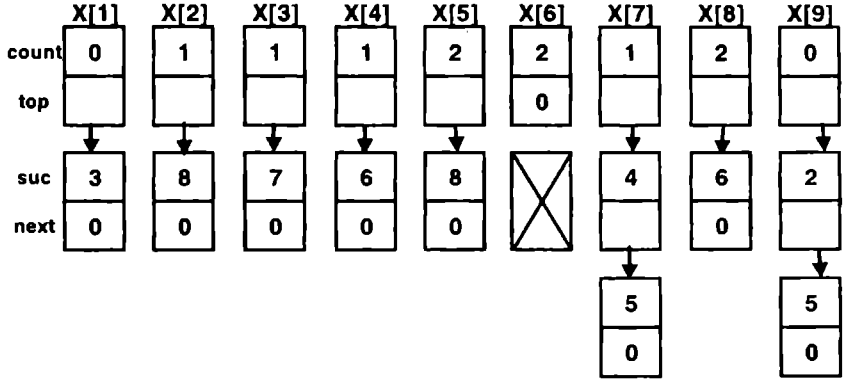


Տոպոլոգիական կարգավորման ալգորիթմը յուրաքանչյուր քայլում ելքային հաջորդականության մեջ գրանցում է բազմության այն տարրը, որը չունի նախնի: Այդ տարրը ու դրանից դուրս եկող կապերը հեռացվում են S բազմությունից: Ալգորիթմի աշխատանքը շա-

րունակվում է այնքան ժամանակ քանի դեռ S-ում տարրեր կան: Մեր օրինակում ելքային հաջորդականությունը կարող է լինել հետևյալը՝



Նկատենք, որ եթե S բազմությունը մասնակի կարգավորված բազմություն է, ապա ալգորիթմը բարեհաջող կավարտի է իր աշխատանքը, այսինքն S բազմության բոլոր տարրերը կհայտնվեն ելքային հաջորդականության մեջ: Իսկապես, ենթադրենք ալգորիթմի աշխատանքի ընթացքում հասել ենք այնպիսի իրավիճակի, որ S-ում չի մնացել նախնի չունեցող տարր: Օրինակ, ընտրենք b_1 տարրը S-ից: Դիցուք նրա նախնին b_2-ն է, որը ևս ունի նախնի՝ b_3 և այլն: Այսպիսով, կարելի է կառուցել $b_1 \succ b_2 \succ b_3 \succ \cdots$ հաջորդականությունը: Քանի որ S-ը վերջավոր բազմություն է, ապա $b_1 \succ b_2 \succ b_3 \succ \cdots \succ b_i \succ \cdots \succ b_j \succ \cdots$ հաջորդականության մեջ կլինեն կրկնվող տարրեր: Ենթադրենք $b_i = b_j$ (i<j): Քանի որ $b_i \succ b_{i+1}$, $b_{i+1} \succ = b_j$, $b_i = b_j$ ապա $b_{i+1} \succ = b_i$. Հիշեցնենք, որ $b_i \succ b_{i+1}$ նշանակում է, որ $b_i \succ = b_{i+1}$ և $b_i \neq b_{i+1}$: Համաձայն հակասիմետրիկության հատկության, եթե $b_i \succ = b_{i+1}$ և $b_{i+1} \succ = b_i$, ապա $b_i = b_{i+1}$, որը հակասում է վերը բերված $b_i \neq b_{i+1}$ պայմանին: Այս ալգորիթմը իրականացնելու համար ընտրենք S մասնակի կարգավորված բազմության հետևյալ ներկայացումը: Համարակալենք S-ի տարրերը և ներկայացնենք այն X[1],X[2], . . ., X[n] զանգվածի միջոցով (n-ը S –ի հզորությունն է): X[i]-ն (1<=i<=n) կազմված է երկու դաշտերից՝ Count[i], որտեղ գրվում է i-րդ տարրի անմիջական նախնիների քանակը, և Top[i], որը նշիչ է i-րդ տարրի անմիջական հետնորդների ցուցակի վրա: Հետնորդների ցուցակի տարրն ունի Suc և Next դաշտերը, որտեղ Suc դաշտում գրվում է հետնորդ-տարրի համարը, իսկ Next-ը՝ ցուցիչ է հետնորդների ցուցակի հաջորդ տարրի վրա:



Տոպոլոգիական կարգավորման ալգորիթմը:
Մուտք - մասնակի կարգավորված S բազմություն:
Ելք - U գծային հաջորդականություն:
Օժանդակ միջոցներ - Q հերթը, որը պարունակում է այն տարրերը, որոնք նախնի-տարրեր չունեն:

Քայլ 1. [տվյալների կառույցի սկզբնարժեքավորում]
∀ i (1≤ i ≤n) {Count[i] <-0, Top[i] <-Λ, N<-n}

Քայլ 2. [$i \prec j$ հարաբերության մուտքագրումը, մուտքի վերջի ստուգում]
եթե (մուտքի վերջն է)
ապա անցնել *Քայլ 3.*
այլապես { Count [j] <-Count[j]+1
P<=AVAIL
Suc(P)<-j
Next(P)<-Top[i]
Top[i]<-P
անցնել *Քայլ 2*}

Քայլ 3. [Q հերթի սկզբնարժեքավորում]
∀ i (1≤ i ≤n), { *եթե* (Count[i] = 0)
ապա i=>Q }

Քայլ 4. [հերթը դատարկ է]
եթե (Q-ն դատարկ է)
ապա անցնել *Քայլ 7*

Քայլ 5. [U ելքային հաջորդականության ձևավորում]
Q=> k, k-> U, P<-Top[k], N<-N-1

Քայլ 6. [փոփոխություն տվյալների կառույցի մեջ]
քանի դեռ (p ≠ Λ)
{Count [Suc(p)] <- Count [Suc(p)] – 1,
եթե (Count[Suc(p)]=0)
ապա Suc(p)-> Q,
p<- Next (p);
}
անցնել *Քայլ 4*
Քայլ 7. [ավարտ]
եթե (N=0)
ապա ավարտ
այլապես "S-ը մասնակի կարգավորված չէ"

[2]-ում տոպոլոգիական կարգավորման ալգորիթմում հերթը համադրվում է Count զանգվածի հետ: Հերթում ընդգրկվում են Count զանգվածի զրոյական տարրերը: Հերթը, որը համադրվում է Count զանգվածի հետ, անվանենք QLINK: F-ը և R-ը՝ ցուցիչներ են հերթի առաջին և վերջին տարրերիի վրա համապատասխանաբար:

Ենթադրենք Count[i_1]=0,Count[i_2]=0, . . ., Count[i_k]=0: Դա նշանակում է, որ F=i_1, R=i_k, QLINK[i_1]=i_2, QLINK[i_2]=i_3, . . ., QLINK[i_{k-1}]=i_k, QLINK[i_k]=QLINK[R]=0:

Նկարագրենք QLINK հերթի նկատմամբ կատարվող գործողությունները:

1. Տարրի ավելացում հերթին:
   եթե (Count[k]=0)
   ապա {QLINK[R] <-k, R<-k}
   Նկատենք, որ QLINK[R]=0:
2. Տարրի հեռացում հերթից և գրանցում u-ն ելքային հաջորդականության մեջ:
   F→u, F<- QLINK[F]
3. Հերթը դատարկ է, թե ոչ: Ենթադրենք, հերթը պարունակում է մեկ տարր, այսինքն QLINK[R]=0 և F=R: Այդ տարրը F<- QLINK[F] գործողությամբ հերթից հեռացնելուց հետո կունենանք դատարկ հերթ: Հետևաբար
   եթե (F= 0)
   ապա հերթը դատարկ է

4. Հերթի սկզբնարժեքավորում:

```
QLINK[0] <- 0
∀ i(1 ≤ i ≤ n) QLINK[i] <- Count[i]
R <- 0
F <- QLINK[0]
```

Նկարագրենք ալգորիթմը հերթի վերոհիշյալ իրականացմամբ:

Քայլ 1. [տվյալների կառույցի սկզբնարժեքավորում]

```
∀ (1 ≤ i ≤ n) { Count[i] <- 0,QLINK[i] <-Count[i],Top[i] <- Λ}
N <-n, QLINK[0] <- 0, R <- 0
```

Քայլ 2. [$i \prec j$ հարաբերության ներմուծում, մուտքի վերջի ստուգում]

```
եթե (մուտքի վերջ)
        ապա անցնել Քայլ 3
        այլապես {
                        Count[j] <- Count[j]+1,
                        P <= AVAIL,
                        Suc(P) <- j,
                        next(P) <- Top[i],
                        Top[i] <- P,
                        անցնել Քայլ 2
                            }
```

Քայլ 3. [QLINK հերթի սկզբնարժեքավորում]

```
∀ (1 ≤ i ≤ n) եթե (Count[i] =0 )
                        ապա { QLINK[R] <- I, R <- i}
                    F <- QLINK[0]
```

Քայլ 4. [հերթը դատարկ է]

```
եթե (F = 0)
        ապա անցնել Քայլ 7
```

Քայլ 5. [ելքային հաջորդականության ձևավորում]

```
F→u,
P <- Top[F],
N <- N – 1,
F <- QLINK[F]
```

Քայլ 6. [փոփոխություն տվյալների կառույցի մեջ]

```
քանի դեռ (P≠Λ)
        { Count[Suc(P)] <- Count[Suc(P)] – 1,
```

```
            եթե (Count[Suc(P)] = 0)
                        ապա {QLINK[R] <- Suc(P), R <- Suc(P)}
            P <- next(P)
            }
      անցնել Քայլ 4
Քայլ 7. [ավարտ]
      եթե (N = 0)
            ապա ավարտ
            այլապես "S-ը մասնակի կարգավորված չէ"
```

Խնդիր 2: Պատահարներով ղեկավարվող մոդելավորում

Դիտարկենք բանկի հաճախորդների սպասարկման խնդիրը և նրա մոդելավորումը:

Առաջարկվող մոդելում կարևորվում են հաճախորդի սպասման ժամանակը (waittime), նրա սպասարկման ժամանակը (servicetime), հաճախորդի հաճախումները, գանձապահի զբաղվածության ժամանակը, գանձապահի սպասարկած հաճախորդների թիվը և այլ պարամետրեր: Երբ հաճախորդը գալիս է բանկ, տեղի է ունենում «գալ» պատահարը (arrival event): Սպասարկումից հետո տեղի է ունենում «գնալ» պատահարը (departure event):

Օրինակ, ենթադրենք 50-րդ հաճախորդը կարող է մոտենալ գանձապահին 2:30-ին, սպասել 3 րոպե, սպասարկվել 12 րոպեի ընթացքում 2-րդ գանձապահի կողմից: Այս դեպքում պատահարների կապը արտահայտվում է հետևյալ կերպ.

time(departure) = time(arrival) + waittime + servicetime
time(departure) = 2:30 + 0:03 + 0:12 = 2:45

Ամբողջ օրվա սպասարկման իրական ընթացքը մոդելավորենք նախապատվությամբ հերթով, համարելով, որ սպասարկման բարձր նախապատվություն ունի այն հաճախորդը, որն ավելի շուտ է կանգնել հերթի:

arrival և departure պատահարները մոդելավորվում են time, etype, customerID, tellerID, waittime, servicetime դաշտեր պարունակող գրառումների միջոցով:

Օրինակ, 50-րդ հաճախորդի համար arrival պատահարի գրառումն է.

1	2	3	4	5	6
2:30	գալը	50	——	——	——
time	etype	customerID	tellerID	waittime	servicetime

4, 5, 6 դաշտերը arrival պատահարի ժամանակ չեն օգտագործվում:

Այդ դաշտերը որոշվում են այն ժամանակ, երբ գեներացվում է departure պատահարը: Նույն հաճախորդի համար departure պատահարի գրառումն է.

2:45	գնալը	50	2	3	12
time	etype	customerID	tellerID	waittime	servicetime

Event դասի հայտարարումը

```
#include <iostream.h>
#include <random.h>
enum EventType{ arrival, departure} ;

class Event{
private:
     // անդամ տվյալներ
     int time; // իրադարձության ժամանակը
     EventType etype; // իրադարձության տիպը
     int customerID; // հաճախորդի համարը
     int tellerID; // գանձապահի համարը
     int waittime; // սպասելու ժամանակը
     int servicetime; // մատակարարելու ժամանակը
public:
     Event(void);
     Event(int t, EventType et, int cn, int tn,
                                   int wt, int st);
     int GetTime(void);
     EventType GetEventType(void);
     int GetCustomerID(void) const;
     int GetTellerID(void) const;
     int GetWaitTime(void) const;
     int GetServiceTime(void) const;
};
```

Օրինակ, եթե 3-րդ հաճախորդը 10 րոպե սպասելուց հետո սպասարկվել է 5 րոպեի ընթացքում և հեռացել է 1:20-ին, ապա կարելի է գրել

```
e=Event(1:20, departure, 3, 1, 10, 5);
cout<<e.GetServiceTime();
// արտածվում է սպասարկման ժամանակը՝ 5
```

Խնդրի մոդելավորման համար անհրաժեշտ ինֆորմացիան (հաճախորդների ընդհանուր թիվը, հաճախորդի սպասման ժամանակը, բոլոր հաճախորդների սպասարկման ընդհանուր ժամանակը) պահվում է TellerStats գրառման մեջ:

FinishService-ը սպասարկման ավարտի պահն է:

TellerStats գրառում

finishService	totalCustomerCount	totalCustomerWait	totalService

```
struct TellerStats {
   int finishService;       // սպասարկման ավարտի պահ
   int totalCustomerCount; // սպասարկված հաճախորդների
                            // ընդհանուր քանակը
   int totalCustomerWait;   // սպասարկման համար սպասված
                            // ընդհանուր ժամանակը
   int totalService;        // սպասարկման ընդհանուր
                            // ժամանակը
};
```

Բանկի հաճախորդների սպասարկման մոդելում օգտագործում է պատահական թվերի գեներատոր, որի միջոցով որոշվում է հաջորդ հաճախորդի գալու պահը և ընթացիկ հաճախորդի սպասարկման ժամանակը: Եթե "գալ" պատահարը տեղի է ունեցել T ժամին, ապա հաջորդ պատահարը հնարավոր է տեղի ունենա T+ arrivalLow-ից մինչև T+arrivalHigh միջակայքի որևէ պահին: Այս խնդրի մոդելավորման տվյալները և մեթոդները պարունակվում են Simulation դասում:

Simulation դասի հայտարարում

```
class simulation {
private:
    // մոդելավորման տվյալներ
    int simulationLength;      // մոդելավորման
                               // տևողությունը
```

```
    int numTellers;         // գանձապահների քանակը
    int nextCustomer;       // հաջորդ հաճախորդի համարը
    int arrivalLow, arrivalHigh;  // ճոր հաճախորդի
                                  // գալու միջակայքը
    int serviceLow, serviceHigh;  // սպասարկման
                                  // միջակայքը
    TellerStats tstat[ 10] ; // գանձապահների
                             // առավելագույն քանակը 10 է
    PQueue pq;                 // նախապատվությամբ հերթ
    RandomNumber rnd;
    // Runsimulation ֆունկցիայի կողմից օգտագործվող
    // փակ մեթոդներ
    int NextArrivalTime ( void);
    int GetserviceTime ( void);
    int NextAvailableTeller ( void);
public:
    // կոնստրուկտոր
    Simulation ( void);
    void RunSimulation ( void);
    void PrintSimulationResults ( void);
};
```

Կոնստրուկտորն արժեքավորում է TellerStats տիպի տարրերով զանգված և դասի nextCustomer անդամը, որը սկսվում է 1-ից: Գանձապահի համարը ինդեքս է նաև tstat զանգվածում: Կոնստրուկտորը օգտվողից ստանում է տվյալներ: Դրանք են մոդելավորման տևողությունը (րոպեներով), գանձապահների քանակը, գալու- գնալու միջակայքը, սպասարկման ժամանակը: Դիտարկվում է մինչև 10 գանձապահ: Ամեն «գալ» պատահարի հետ կանչվում է NextArrivalTime մեթոդը, որպեսզի որոշվի, թե հաջորդ հաճախորդը երբ է գալու: Կանչվում է GetserviceTime –ը, որպեսզի որոշվի ընթացիկ հաճախորդի սպասարկման ժամանակը: Հաճախորդին սպասարկող գանձապահին որոշելու համար կանչվում է nextAvaiilabeleTeller մեթոդը:

Runsimulation մեթոդն իրականացնում է մոդելավորվող խնդիրը, իսկ PrintSinulationResults-ը տալիս է վերջնական վիճակագրությունը:

Simulation դասի իրականացում

```
Simulation:: Simulation( void) {
    int i;
    Event firstevent;
    // գանձապահների պարամետրերի սկզբնարժեքավորում
    for ( i=1; i<=10;i++)
    {
        tstat[  i ].FinishService=0;
        tstat[  i ].TotalService=0;
        tstat[  i ].TotalCustomerWait=0;
        tstat[  i ].TotalCustomerCount=0;
    }
    nextCustomer=1;
    cout<< "Ներմուծեք մոդելավորման ժամանակը
            րոպեներով";
    cin>> simulationLength;
    cout<< "Ներմուծեք բանկում գանձապահների քանակը";
    cin>> numTellers;
    cout<< "Ներմուծեք գալու ժամանակի միջակայքը
            րոպեներով";
    cin>> arrivevalLow >>arrivalHigh;
    cout << "Ներմուծեք սպասարկման ժամանակի միջակայքը
            րոպեներով";
    cin>>serviceLow>>serviceHigh;
    // գեներացնել առաջին հաճախորդի "գալ" պատահարը
    pq.PQInsert (Event(0,arrival,1,0,0,0));
}
```

simulationLength-ը մոդելավորման տևողությունն է, որը որոշում է՝ արդյոք պետք է գեներացնել հաճախորդի "գալ" պատահարը: Եթե պատահարի տեղի ունենալու պահին բանկը փակվում է՝ arrivaltime>simulationLength, ապա նոր հաճախորդներ չեն ընդունվում:

NextArrivalTime մեթոդը վերադարձնում է մինչև մյուս հաճախորդի գալու պահը, որն ընկած է arrivalLow և arrivalHigh պարամետրերով որոշվող միջակայքում: GetServiceTime մեթոդում օգտագործվում են համապատասխանաբար serviceLow և serviceHigh պարամետրերը:

```
// որոշել հաջորդ հաճախորդի գալու պատահական ժամանակը
int Simulation:: NextArrivalTime( void) {
      return arrivalLow+rnd.Random
      arrivalHigh-arrivalLow+1);
}

// որոշել հաճախորդի սպասարկման պատահական ժամանակը
int Simulation::GetServiceTime( void) {
      return serviceLow+rnd.Randow
      (serviceHigh-serviceLow+1);
}
```

Հերթական հաճախորդն ընտրում է իր գանձապահին ըստ finishService տվյալի արժեքի: NextAvailableTeller ֆունկցիան գանձապահների զանգվածից վերադարձնում է այն գանձապահի համարը, որի մոտ finishService-ի արժեքը ամենափոքրն է: Եթե մինչև բանկի փակվելը բոլոր գանձապահները զբաղված են, ապա հաճախորդին սպասարկելու համար որոշվում է որևէ գանձապահ:

```
// վերադարձնել առաջին հասանելի գանձապահի համարը
int Simulation:: NextAvailableTeller ( void) {
    //ենթադրվում է, որ բոլոր գանձապահները ազատ են
     // մինչև բանկի փակվելը
    int minfinish=simulationLength;
    int minfinishindex= rnd.Random(numTellers)+1;
    //գտնել առաջին ազատվող գանձապահին
    for (int i=1; i <= numTellers; i++)
          if (tstat[ i] .finishService < minfinish) {
               minfinish=tstat[ i] .finishService;
               minfinishindex=i;
          }
    return minfinishindex;
}
```

Մոդելավորման հիմնական ֆունկցիան RunSimulation-ն է, որը ղեկավարում է պատահարների նախապատվությամբ հերթը: RunSimulation-ն ավարտում է իր աշխատանքը դատարկ հերթի դեպքում:

Մոդելավորման արդյունքներն արտածվում են PrintSimulation Results ֆունկցիայի միջոցով:

ԽՆԴԻՐՆԵՐ

1. Իրականացնել հերթը ցիկլիկ զանգվածի միջոցով:
2. * Իրականացնել հերթի շրջանածև մոդելը առանց հերթի տարրերի քանակի փոփոխականի օգտագործման: Նախատեսել մեկ "դատարկ" տարր հերթի սկզբում: Նախասեսել բացառիկ իրավիճակների մշակում:
3. * Միանիշ և երկնիշ բնական թվերից կազմված հաջորդականությունը դասավորել չնվազման կարգով ռադիքս (radix) կարգավորման միջոցով:
4. * Ռադիքս կարգավորումը իրականացնել կամայական բնական թվերի հաջորդականության համար:
5. Տրված են Q1 և Q2 հերթերը: Կառուցել Q3 հերթը, որի առաջին մասում Q1 -ի տարրերն են, իսկ երկրորդ մասում՝ Q2-ի տարրերը շրջված կարգով:
6. Տրված է Q1 հերթը: Ստանալ Q2 հերթը, որում Q1-ի տարրերն են՝ դասավորված չնվազման կարգով:
7. Տրված են Q1 և Q2 հերթերը: Կառուցել Q3 հերթը, որի մեջ մեկումեջ դասավորված են Q1 և Q2 հերթերի տարրերը, իսկ վերջում՝ չդատարկված հերթի տարրերը:
8. Տրված է Q1 հերթը: Ստանալ Q2 հերթը, որում Q1–ի տարրերն են առանց կրկնումների:
9. Երկկողմանի հերթը սահմանվում է որպես տվյալների այնպիսի կառուցվածք, որի երկու ծայրերից էլ կարելի է կատարել տվյալների ավելացում և հեռացում (այլ կերպ նրան ասում են հերթ երկկողմանի հասանելիությամբ կամ դեկ): Իրականացնել աբստրակտ երկկողմանի հերթը՝ ժառանգելով այն սովորական հերթից:

ԳԼՈՒԽ 5.

ԾԱՌԵՐ

Հաջորդական և կապակցված ցուցակները գծային կառուցվածքներ են, որովհետև ունեն սկզբնական և վերջնական տարրեր, իսկ յուրաքանչյուր ներքին տարր ունի միայն մեկ հետնորդ (ժառանգ):

Սակայն շատ դեպքերում տարրերը կարող են ունենալ մեկից ավել հետնորդներ՝ առաջացնելով ոչ գծային կախվածություն: Այդպիսի կապը կոչվում է հիերարխիկ, իսկ կառուցվածքը՝ ոչ գծային: Ոչ գծային կառուցվածքներից են ծառերն ու գրաֆները:

T վերջավոր տարրերի բազմությունը կոչվում է ծառ (tree), եթե այն դատարկ է կամ այդ բազմության մեջ առանձնացված է մեկ տարր, որը կոչվում է ծառի արմատ (root), իսկ մնացած տարրերը բաշխված են T1, T2, T3, ..., Tm (m≥0) չհատվող բազմությունների միջև, որտեղ ցանկացած Ti-ն ծառ է: Նկատենք, որ ծառը կարող է բաղկացած լինել միայն արմատից: T1, T2, T3, ..., Tm բազմությունները կոչվում են T ծառի արմատի ենթածառեր:

Այսպիսի սահմանումից հետևում է, որ կապակցված ցուցակը նույնպես ծառ է, որտեղ յուրաքանչյուր գագաթ ունի մեկից ոչ ավել ենթածառ: Ծառով տրվում է հիերարխիկ կառուցվածք, որտեղ տարրերի միջև առկա է ծնող-որդի հարաբերություն:

Ծառի տեսքով կարելի է ներկայացնել գրքի ցանկը, ծագումնաբանական ծառերը, արտահայտությունների ծառերը և այլն:

Գոյություն ունեն ծառերի պատկերման շատ եղանակներ: Մենք ծառը կպատկերենք գագաթների (հանգույցների) և կողերի տեսքով: Ծառի պատկերման մեջ y գագաթը, որը կապված է x գագաթի հետ և գտնվում է անմիջականորեն նրանից ներքև կոչվում է x-ի որդի, այդ դեպքում x-ը y-ի ծնողն է:

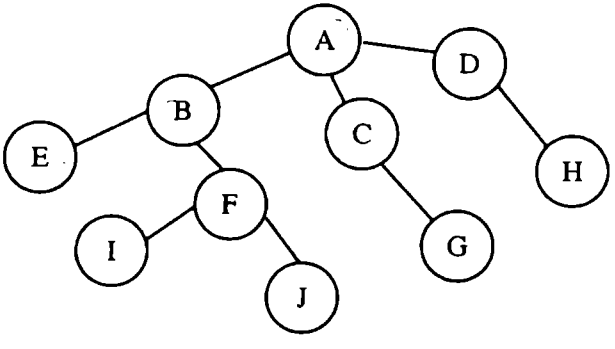
Ծառի գագաթները պատկերվում են ըստ մակարդակների (level): Արմատի մակարդակը զրո է: Յուրաքանչյուր այլ հանգույցի մակարդակ մեկով մեծ է այդ հանգույցն ընդգրկող մոտակա ծառի արմատի մակարդակից: Տվյալ հանգույցի մակարդակը մեկով մեծ է իր ծնողի մակարդակից: Ծառի գագաթների մակարդակներից ամենամեծը կոչվում է ծառի խորություն կամ բարձրություն:

Ծառի յուրաքանչյուր գագաթ (բացառությամբ արմատի) ունի միայն մեկ ծնող, իսկ յուրաքանչյուր ծնող ունի զրո կամ ավելի որդիներ: Որևէ գագաթի որդիները, նրա որդիների որդիները (և այլն) կոչվում են այդ գագաթի հետնորդներ (ժառանգներ): Որևէ գագաթի ծնողը, նրա ծնողի ծնողները (և այլն) կոչվում են այդ գագաթի նախնիներ: Այն գագաթը, որը չունի որդի կոչվում է տերև (leaf):

Տերև չհանդիսացող գագաթը կոչվում է ներքին գագաթ: Ներքին գագաթի որդիների քանակը կոչվում է նրա աստիճան: Բոլոր գագաթների մաքսիմալ աստիճանը կոչվում է ծառի աստիճան: Տերևի աստիճանը զրո է:

Ծնողից դեպի նրա հետնորդները գնացող ուղին կոչվում է ճանապարհ: Քանի որ արմատ չհանդիսացող յուրաքանչյուր գագաթ ունի մեկ ծնող, հետևաբար, յուրաքանչյուր գագաթից իր հետնորդներին կա միայն մեկ ճանապարհ: Ծառի կողերի այն քանակը, որն անհրաժեշտ է արմատից մինչև տրված գագաթը անցնելու համար կոչվում է ճանապարհի երկարություն: Ծառի բարձրությունը համընկնում է ծառի արմատից դեպի տերևներ տանող ամենաերկար ճանապարհի երկարության հետ: Ծառի որևէ գագաթի մակարդակը, դա արմատից դեպի այդ գագաթը տանող ճանապարհի երկարությունն է:

5.1 նկարում բերված օրինակում A-ն ծառի արմատն է, B-ն՝ E և F որդիների ծնողը: E, F, I, J-ն B հանգույցի սերունդներն են: F, I, J հանգույցները պարունակող ենթածառի արմատը F-ն է: G-ն սերունդներ չունեցող ենթածառի արմատ է: E, I, J, G, H-ը ծառի տերևներն են: A,B,F,J ճանապարհի երկարությունը 3 է: F հանգույցի մակարդակը 2 է: Ծառի բարձրությունն է 3:



Նկ. 5.1 Ընդհանուր տեսքի ծառ

5.1. ԲԻՆԱՐ ԾԱՌԵՐ

Դիտարկենք այնպիսի ծառեր, որտեղ ամեն ծնող ունի երկուսից ոչ ավել որդիներ (նկ. 5.2): Այդպիսի ծառերը կոչվում են բինար ծառեր (binary trees): Բինար ծառն ունի ռեկուրսիվ կառուցվածք, հետևաբար, ծառի մշակման պրոցեդուրաները ևս ռեկուրսիվ են: Բինար ծառը ռեկուրսիվ ձևով սահմանվում է հետևյալ կերպ:

Բինար ծառը հանգույցների T վերջավոր բազմություն է, որտեղ

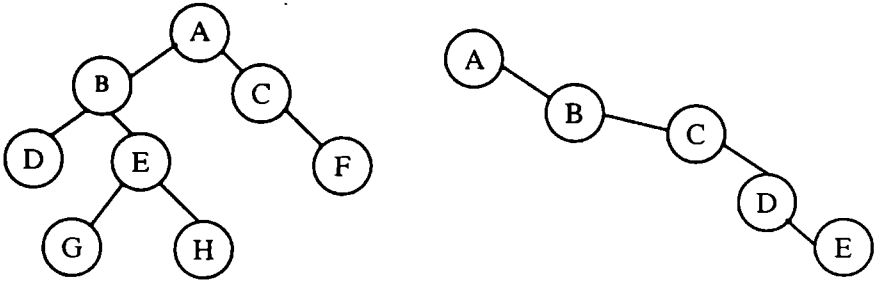
ա) T-ն ծառ է, եթե հանգույցների բազմությունը դատարկ է (դատարկ ծառ),

բ) T-ն տրոհվում է երեք չհատվող ենթաբազմությունների.

- {R} արմատային հանգույց (root),
- {L1, L2, ..., Lm} R-ի ձախ ենթածառ,
- {R1, R2, ..., Rn} R-ի աջ ենթածառ:

Նկատենք, որ բինար ծառը կարող է լինել դատարկ և ծառի կամայական հանգույց ունի ոչ ավել քան երկու որդի: Բինար ծառը կարգավորված ծառ է, այսինքն՝ նրա կամայական հանգույցի համար ենթածառերի հերթականությունը կարևոր է:

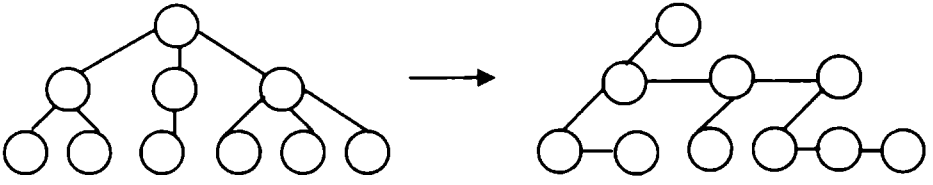
Յուրաքանչյուր n մակարդակում բինար ծառը կարող է ունենալ 1-ից մինչև 2^n հանգույց:



Նկ. 5.2 Բինար ծառեր

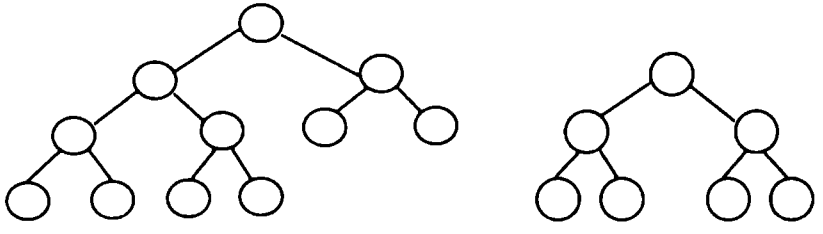
Այսուհետև կդիտարկենք միայն բինար ծառերը, որովհետև յուրաքանչյուր ընդհանուր տեսքի ծառ կարելի է ներկայացնել իրեն համարժեք բինար ծառով: Դրա համար ծառի մեջ յուրաքանչյուր ծնողի բոլոր որդիներից հարկավոր է թողնել միայն ամենաառաջինը (ձախը)՝ մաքրելով բոլոր մնացած ուղղածիգ գծերը, իսկ ծնողի բոլոր որդիները միացնել հորիզոնական գծերով (նկ.5.3):

Ավարտուն (կատարյալ) բինար ծառը (complete binary tree) դա N խորությամբ ծառ է, որտեղ յուրաքանչյուր 0...N-1 մակարդակ ունի հանգույցների լրիվ հավաքածու, իսկ N-րդ մակարդակում բոլոր հանգույցները (տերևները) տեղավորված են ձախից:



Նկ. 5.3 Ընդհանուր տեսքի ծառի բերումը բինար ծառի

Ավարտուն բինար ծառը, որը N-րդ մակարդակում պարունակում է 2^N գագաթ կոչվում է լրիվ ծառ: n հանգույցով ավարտուն ծառի խորությունը $[\log_2 n]$ է:



Ավարտուն ծառ *Լրիվ ծառ*

Ծառը կոչվում է իդեալապես բալանսավորված, եթե նրա յուրաքանչյուր գագաթի աջ և ձախ ենթածառերի գագաթների քանակները միմյանցից տարբերվում են ոչ ավել քան մեկով:

5.2 ԲԻՆԱՐ ԾԱՌ ՏԿՅԱԼՆԵՐԻ ԱԲՍՏՐԱԿՏ ՏԻՊԸ

Բինար ծառ ADT-ի նկարագրությունը հետևյալն է.

ADT BinaryTree

Տվյալներ

Բինար ծառ

Գործողություններ

Կոնստրուկտոր

մուտք – չկա

նախապայման – չկա
պրոցես – դատարկ բինար ծառի սկզբնարժեքավորում
ելք –չկա
վերջնապայման – դատարկ ծառ

Length

մուտք – չկա
նախապայման – չկա
պրոցես – բինար ծառի գագաթների քանակի որոշում
ելք – բինար ծառի տարրերի քանակ
վերջնապայման – չկա

Empty

մուտք – չկա
նախապայման – չկա
պրոցես – բինար ծառի դատարկ լինելու պայմանի ստուգում
ելք – 1(true), եթե բինար ծառը դատարկ է, 0 (false) հակառակ դեպքում
վերջնապայման – չկա

Find

մուտք – բինար ծառում փնտրվող տարրը
նախապայման – չկա
պրոցես – տարրի փնտրում բինար ծառում
ելք – փնտրվող տարրի դիրքը բինար ծառում
վերջնապայման – չկա

InsertLeft

մուտք – ցուցիչ բինար ծառի հանգույցի վրա, որին որպես ձախ ենթածառ պետք է ավելացվի տրված տարրը/ենթածառը
նախապայման – բինար ծառի հանգույցը, որին պետք է ավելացվի ենթածառ, չունի ձախ որդի
պրոցես – տարրի/ենթածառի ավելացում բինար ծառին՝ տրված դիրքում
ելք – չկա
վերջնապայման – ավելացված ենթածառով բինար ծառ

InsertRight

մուտք – ցուցիչ բինար ծառի հանգույցի վրա, որին որպես աջ ենթածառ պետք է ավելացվի տրված տարրը/ենթածառը
նախապայման – բինար ծառի հանգույցը, որին պետք է ավելացվի ենթածառ, չունի աջ որդի

պրոցես – տարրի/ենթածառի ավելացում բինար ծառին՝ տրված դիրքում

ելք – չկա

վերջնապայման – ավելացված ենթածառով բինար ծառ

DeleteLeft

մուտք – ցուցիչ բինար ծառի հանգույցի վրա, որի ձախ ենթածառը պետք է հեռացվի

նախապայման – բինար ծառի տրված հանգույցն ունի ձախ ենթածառ

պրոցես – տրված հանգույցի ձախ ենթածառի հեռացում

ելք – չկա

վերջնապայման – հեռացված ենթածառով բինար ծառ

DeleteRight

մուտք – ցուցիչ բինար ծառի հանգույցի վրա, որի աջ ենթածառը պետք է հեռացվի

նախապայման – բինար ծառի տրված հանգույցն ունի աջ ենթածառ

պրոցես – տրված հանգույցի աջ ենթածառի հեռացում

ելք – չկա

վերջնապայման – հեռացված ենթածառով բինար ծառ

InorderTraverse

մուտք – չկա

նախապայման – չկա

պրոցես – բինար ծառի ուղիղ շրջանցում

ելք – բինար ծառի ուղիղ շրջանցման արդյունք

վերջնապայման – չկա

PreorderTraverse

մուտք – չկա

նախապայման – չկա

պրոցես – բինար ծառի սիմետրիկ շրջանցում

ելք – բինար ծառի սիմետրիկ շրջանցման արդյունք

վերջնապայման – չկա

PostorderTraverse

մուտք – չկա

նախապայման – չկա

պրոցես – բինար ծառի հակադարձ շրջանցում

ելք – բինար ծառի հակադարձ շրջանցման արդյունք

վերջնապայման – չկա

ClearBinaryTree
    մուտք – չկա
    նախապայման – չկա
    պրոցես – բինար ծառի բոլոր տարրերի հեռացում
    ելք – չկա
    վերջնապայման – դատարկ բինար ծառ
CopyTree
    մուտք – բինար ծառ
    նախապայման – չկա
    պրոցես – բինար ծառի պատճենում
    ելք – չկա
    վերջնապայման – պատճենված բինար ծառ
վերջ ADT BinaryTree

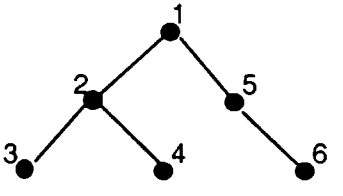
5.3 ԲԻՆԱՐ ԾԱՌԻ ԻՐԱԿԱՆԱՑՈՒՄ

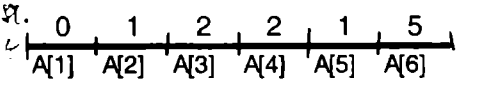
Բինար ծառը կարելի է ներկայացնել միաչափ զանգվածի կամ կապակցված ցուցակի միջոցով:

Իրականացում զանգվածի միջոցով:

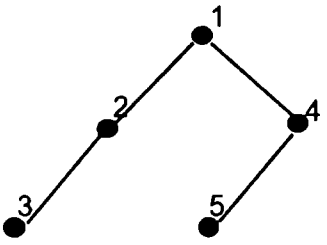
Դիցուք ծառն ունի N համարակալված հանգույց: Ծառը ներկայացնենք A[N] զանգվածի տեսքով, որտեղ A[i]-ն պարունակում է i-րդ հանգույցի ծնողի համարը, ինչպես նաև հանգույցի տվյալների դաշտը:

Ստորև բերված օրինակում զանգվածի տարրերում նշված են միայն ծնող հանգույցների համարները:





Արմատի ծնողի համարը 0 է: Ծառի այսպիսի ներկայացման դեպքում արդյունավետ են կատարվում հանգույցի ծնողի, ծառի տերևի, ծառի բարձրության որոշման գործողությունները: Ծառի մեկ այլ ներկայացման դեպքում A[N] զանգվածի յուրաքանչյուր տարր պարունակում է տվյալ հանգույցի ձախ և աջ որդիների համարները:



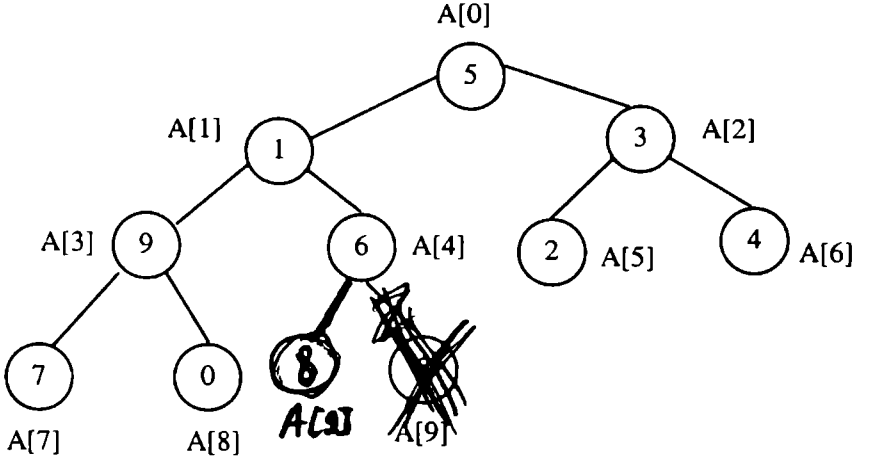
A[1]	2	4
A[2]	3	0
A[3]	0	0
A[4]	5	0
A[5]	0	0

Իրականացում զանգվածի միջոցով ըստ մակարդակների

Այսպիսի իրականացումով ծառերը կօգտագործենք բուրգային կառուցվածքների և տուրնիրային կարգավորման ժամանակ: Այս դեպքում զանգվածի տարրերում պահվում են տվյալները, իսկ հանգույցները ցույց են տրվում ինդեքսներով: Հիշենք, որ n խորությամբ ավարտուն բինար ծառը պարունակում է մինչև n-1 մակարդակի բոլոր հնարավոր հանգույցները, իսկ n մակարդակի հանգույցները տեղադրվում են ձախից աջ: A զանգվածը հաջորդական ցուցակ է, որի տարրերը ավարտուն բինար ծառի հանգույցներն են: Ընդ որում՝ A[0]-ն արմատն է, A[1], A[2]–ը՝ առաջին մակարդակի սերունդը, A[3], A[4], A[5], A[6]–ը՝ երկրորդ մակարդակի սերունդը և այլն:

Հետևյալ նկարում պատկերված է 10 տարրից բաղկացած A զանգվածով տրվող ավարտուն բինար ծառ:

```
int A[ 10 ] = { 5, 1, 3, 9, 6, 2, 4, 7, 0, 8 }
```



Վերը նշված ծառը կարելի է ներկայացնել հետևյալ աղյուսակով.

մակարդակ	ինդեքս	արժեք	ձախ որդի (ինդեքս)	աջ որդի (ինդեքս)
0	0	A[0]=5	1	2
1	1	A[1]=1	3	4
	2	A[2]=3	5	6
2	3	A[3]=9	7	8
	4	A[4]=6	9	–
	5	A[5]=2	–	–
	6	A[6]=4	–	–
3	7	A[7]=7	–	–
	8	A[8]=0	–	–
	9	A[9]=8	–	–

N տարր պարունակող A զանգվածի յուրաքանչյուր A[i] հանգույցի համար նրա որդիների ինդեքսը որոշվում է հետևյալ կերպ.

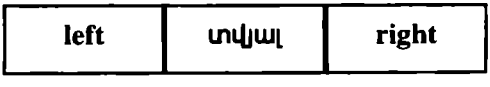
ձախ որդու ինդեքսը =2*i+1 (անորոշ է, երբ $2*i+1 \geq N$),

աջ որդու ինդեքսը =2*i+2 (անորոշ է, երբ $2*i+2 \geq N$),

A[i] հանգույցի ծնողի ինդեքսը =(i-1)/2 (անորոշ է, երբ i=0):

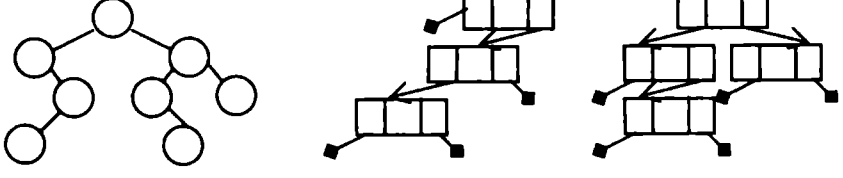
Իրականացում կապակցված ցուցակի միջոցով

Այս դեպքում ծառի յուրաքանչյուր հանգույց ունի հետևյալ կառուցվածքը.



Այստեղ left-ը ցուցիչ է հանգույցի ձախ ենթածառի արմատի վրա, իսկ right-ը՝ աջ ենթածառի արմատի վրա:

Ծառը տրվում է արմատը մատնանշող ցուցիչով:

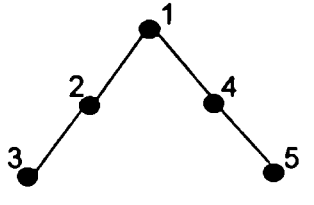


Նկարագրենք ծառի հանգույցը ներկայացնող TreeNode շաբլոն դասը:

```
template <class T>
class BinSTree;
template <class T>
class TreeNode {
private:
     TreeNode <T> *left, *right; // ձախ և աջ որդիների
                                 // ցուցիչներ
public:
     T info; // բաց անդամ-տվյալ
     // կոնստրուկտոր
     TreeNode (const T &item, TreeNode <T> *l=0,
     TreeNode <T> *r=0):
          info (item), left (l), right (r) { }
     // ցուցչային դաշտերին հասանելիության մեթոդներ
     TreeNode <T> *Left(void) const { return left;}
     TreeNode <T> *Right(void) const { return right;}
     friend class BinSTree <T>; // BinSTree դասը
                                // հայտարարել friend
};
```

Հանգույցը կառուցված է public հայտարարված տվյալից (info) և նրա ձախ և աջ հանգույցների ցուցիչների private հայտարարված դաշտերից (left,right): Կոնստրուկտորը սկզբնարժեքավորում է հանգույցի դաշտերը: Տերև հանդիսացող հանգույցների left և right դաշտերը NULL են: Left() և Right() ֆունկցիաները հասանելի են դարձնում ձախ և աջ հանգույցների ցուցիչները: BinSTree դասը հայտարարված է որպես TreeNode դասի ընկեր դաս (friend):

Ստորև բերված է ծառը և այն կառուցող ծրագիրը:




```
int main (){
    TreeNode <int> a(3);
    TreeNode <int> b(2,&a);
    TreeNode <int> c(5);
    TreeNode <int> d(4,0,&c);
    TreeNode <int>
             root(1,&b,&d);
    return 0;
}
```

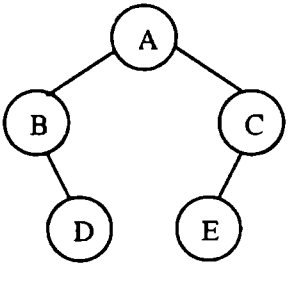
Ծառի հանգույցների ստեղծման և ոչնչացման համար նկարագրենք GetTreeNode և FreeTreeNode ֆունկցիաները: GetTreeNode ֆունկցիան որպես պարամետր ստանում է հանգույցի արժեքը և TreeNode օբյեկտների 0, 1 կամ 2 ցուցիչներ: Եթե ցուցիչներն առկա են, ապա նոր ստեղծված հանգույցը միացվում է այդ ցուցիչներով տրված որդիներին:

```
template <class T>
TreeNode<T>*GetTreeNode  (T  item,TreeNode<T>*lptr=0,
TreeNode<T>* rptr=0) {
    TreeNode<T> *p;
    P=new TreeNode<T>(item, lptr, rptr);
    if (p==NULL) {
        cerr << "հիշողության սխալ" ;
        exit(1);
    }
    return p;
}
```

FreeTreeNode ֆունկցիան ստանալով TreeNode օբյեկտի ցուցիչը, ազատում է նրա կողմից զբաղեցրած հիշողության տիրույթը:

```
template <class T>
void FreeTreeNode(TreeNode<T> *p) {
    delete p;
}
```

Այժմ GetTreeNode-ի օգնությամբ կառուցենք գագաթներում սիմվոլային արժեքներ պարունակող հետևյալ ծառը: Ծառի կառուցումը կատարվում է ներքևից վերև:



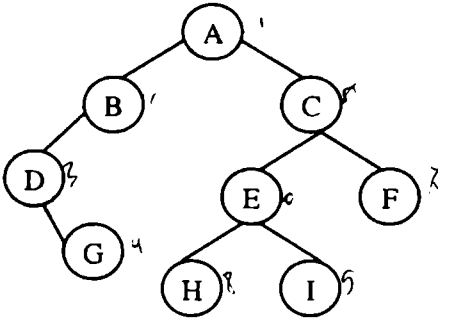

```
TreeNode <char> *a, *b, *c, *d,
                *e, *root;
d = GetTreeNode ('D');
e = GetTreeNode ('E');
b = GetTreeNode ('B', NULL, d);
c = GetTreeNode ('C', e, NULL);
a = GetTreeNode ('A', b, c);
root = a;
```

5.4 ԾԱՌԵՐԻ ՇՐՋԱՆՑՄԱՆ ԵՂԱՆԱԿՆԵՐ

Դիտարկենք ծառի գագաթների շրջանցման ուղիղ, սիմետրիկ և հակադարձ եղանակները: Նրանցից յուրաքանչյուրը հիմնված է բինար ծառի ռեկուրսիվ կառուցվածքի վրա: Շրջանցման յուրաքանչյուր ալգորիթմի դեպքում կատարվում է երեք գործողություն՝ այցելել հանգույց, ռեկուրսիվ անցնել ձախ ենթածառով, ռեկուրսիվ անցնել աջ ենթածառով: Անցումը դադարում է հասնելով դատարկ ծառին:

1. Ուղիղ շրջանցում (Preorder) - այցելել ծառի արմատը, այնուհետև ուղիղ շրջանցել ձախ ենթածառը, հետո ուղիղ շրջանցել աջ ենթածառը (NLR շրջանցում - Node, Left, Right):
2. Սիմետրիկ շրջանցում (Inorder) - սիմետրիկ շրջանցել ձախ ենթածառը, այնուհետև այցելել ծառի արմատը, հետո սիմետրիկ շրջանցել աջ ենթածառը (LNR շրջանցում):
3. Հակադարձ շրջանցում (Postorder) - հակադարձ շրջանցել ձախ ենթածառը, այնուհետև հակադարձ շրջանցել աջ ենթածառը, հետո այցելել գագաթը (LRN շրջանցում):

Ծառի հանգույցների բոլոր նշված շրջանցումները պատկերված են նկար 5.4-ում:



ուղիղ - ABDGCEHIF

սիմետրիկ - DGBAHEICF

հակադարձ - GDBHIEFCA

Նկ.5.4 Ծառի հանգույցների ուղիղ, սիմետրիկ և հակադարձ շրջանցումներ

Ուղիղ շրջանցման ռեկուրսիվ ալգորիթմը հետևյալն է.

```
Preorder(T ծառ)
{ եթե(T-ն դատարկ չէ)
     ապա { այցելել արմատը;
            Preorder(T-ի ձախ ենթածառ);
            Preorder(T-ի աջ ենթածառ);
          }
}
```

Սիմետրիկ շրջանցման Inorder շաբլոն-ֆունկցիան ունի հետևյալ տեսքը.

```
template < class T >
void Inorder(TreeNode <T> *t, void visit (T& item)) {
    // ռեկուրսիվ շրջանցումն ավարտվում է դատարկ ծառում
    if (t != NULL) {
        Inorder(t → Left(), visit);
                    // շրջանցել ձախ ենթածառը
        visit(t → info);
                    // այցելել հանգույցը
        Inorder(t → Right(), visit);
                    // շրջանցել աջ ենթածառը
    }
}
```

Հակադարձ շրջանցման Postorder շաբլոն-ֆունկցիան ունի հետևյալ տեսքը.

```
template <class T>
void Postorder (TreeNode <T> *t,
void visit (T& item)) {
    // ռեկուրսիվ շրջանցումն ավարտվում է դատարկ ծառում
    if (t !=NULL) {
        Postorder (t → Left(), visit);
                                // շրջանցել ձախ ենթածառը
        Postorder (t → Right(), visit);
                                // շրջանցել աջ ենթածառը
        visit (t → info);       // այցելել հանգույցը
    }
}
```

Եթե որպես visit ֆունկցիա ընդունենք PrintChar-ը, ապա ծառի սիմետրիկ շրջանցման ֆունկցիան կարելի է օգտագործել հետևյալ կերպ.

```
void PrintChar (char & elem) {
    cout << elem << " " ;
}
```

```
int   main (){
    TreeNode < char > *root;
    // ծառի կառուցում
    . . . . . .
    cout << " սիմետրիկ անցում ";
    Inorder (root, PrintChar);
    return 0;
}
```

Այժմ դիտարկենք ծառի գագաթների շրջանցման ոչ ռեկուրսիվ (իտերատիվ) եղանակներ:

Նկարագրենք սիմետրիկ շրջանցման իտերատիվ ալգորիթմը:

Մուտք - T ցուցիչ արմատի վրա
Ելք - սիմետրիկ շրջանցման արդյունք
Օժանդակ միջոցներ - գագաթների ցուցիչների A պահունակ,
ընթացիկ գագաթի p ցուցիչ:

քայլ 1. [սկզբնարժեքավորում]
T → p, դատարկել A պահունակը
քայլ 2. [p = Λ ?]
եթե (p=Λ) *ապա* անցնել *քայլ 4*
քայլ 3. [p ≠ Λ]
p ⇒ A
p←LLINK (p)
անցնել *քայլ 2*
քայլ 4. [A-ն դատարկ է ?]
եթե (A = Λ) *ապա* անցնել *քայլ 6*
քայլ 5. [A-ից հանել ցուցիչը և մշակել հանգույցը]
p ⇐ A
մշակել p-ով նշվող հանգույցը (օրինակ cout << p → info;)
p ← RLINK (p)
անցնել *քայլ 2*
քայլ 6. [ավարտ]

Օրինակ: Կազմել ֆունկցիա, որը սիմետրիկ շրջանցում է բինար ծառը, կատարելով յուրաքանչյուր գագաթի արժեքի մշակում: Գագաթի մշակման ֆունկցիան փոխանցվում է շրջանցման ֆունկցիային որպես պարամետր:

```
template <class T>
void Inorder (TreeNode <T> *t, void visit(T item)) {
     stack <TreeNode <T> * > A;
     TreeNode <T> *p = t;
     bool b=true;
     while (b) {
            while (p != 0) {
            A.push (p);
            p=p→Left();
            }
            if (A.IsEmpty ())
                  b=false;
            else {
                  A.gettop (p);
                  A.pop ();
                  visit (p→info);
                  p=p→Right();
            }
     }
}
template <class T>
void f1(T & x) {
      x = x+1;
}
void f2(int &y){
      cout << y + 2;
}
void  main (){
      TreeNode <int> *root;
      //կառուցել ծառը, root-ը ծառի արմատի ցուցիչն է
      //սիմետրիկ շրջանցել ծառը, f1-ը կիրառելով
      //յուրաքանչյուր հանգույցի վրա
      Inorder(root, f1);
      //սիմետրիկ շրջանցել ծառը, f2-ը կիրառելով
      //յուրաքանչյուր հանգույցի վրա
      Inorder(root, f2);
}
```

Կարելի է կատարել նաև ծառի գագաթների լայնակի (ըստ մակարդակների՝ ձախից աջ) շրջանցում: Նկ. 5.4-ում պատկերված ծառը լայնակի շրջանցելիս, նրա գագաթները կդիտարկվեն հետևյալ հաջորդականությամբ՝ ABCDEFGHI:

Նկարագրենք ծառի լայնակի շրջանցման իտերատիվ ալգորիթմը:

Մուտք - T ցուցիչ արմատի վրա
Ելք - լայնակի շրջանցման արդյունք
Օժանդակ միջոցներ - ծառի գագաթների ցուցիչներ պարունակող Q հերթ, ծառի գագաթների p ցուցիչ:

քայլ 1. [սկզբնարժեքավորում]
դատարկել Q հերթը
T→p
եթե (p ≠ Λ) *ապա* p⇒ Q

քայլ 2. [Q = Λ?]
եթե (Q = Λ)
ապա անցնել *քայլ 4*

քայլ 3. [Q ≠ Λ]
եթե (Q ≠ Λ)
ապա
{ p ⇐ Q
p -ով մատնանշվող հանգույցի մշակում
եթե (LLINK(p) ≠ Λ)
ապա LLINK(p) ⇒ Q
եթե (RLINK(p) ≠ Λ)
ապա RLINK(p) ⇒ Q
անցնել *քայլ 2*
}
քայլ 4. [ավարտ]

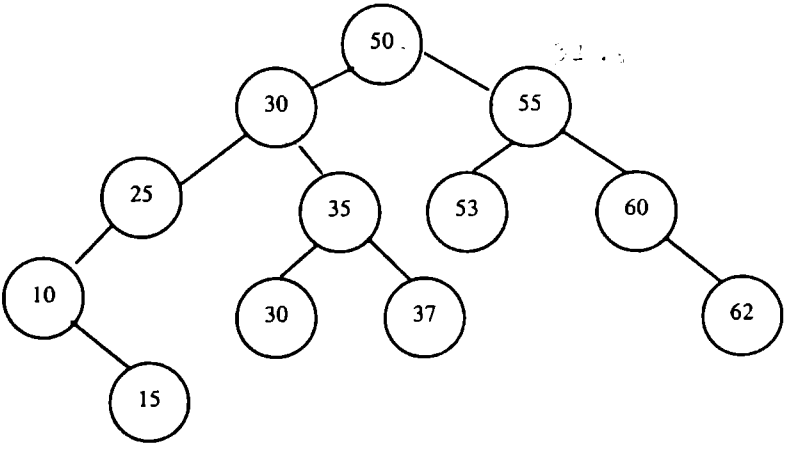
5.5 ՈՐՈՆՄԱՆ ԲԻՆԱՐ ԾԱՌԵՐ

Գծային կառուցվածքներում տարրի փնտրումը պահանջում է O(N) կարգի գործողություն (համեմատում), որը մեծ հավաքածուների դեպքում արդյունավետ չէ: Ծառանման կառուցվածքներում ցանկացած տվյալի փնտրման ճանապարհի երկարությունը չի գերազանցում ծառի խորությանը, որն ապահովում է փնտրման ավելի բարձր արդյունավետություն: Ավարտուն ծառերի դեպքում փնտրման արդյունավետությունն առավելագույնն է՝ $O(\log_2 N)$:

Օրինակ, 10000 տարր պարունակող ցուցակում հաջորդական փնտրման դեպքում համեմատումների միջին թիվը 5000 է: Ավարտուն ծառի դեպքում փնտրումը կպահանջեր 14 –ից ոչ ավել գործողություն ($O(\log_2 10000)$):

Բինար որոնման ծառում (binary search tree) տարրերը կարգավորվում են "<" հարաբերությամբ: Ծառի հանգույցները համեմատելիս տվյալների դաշտի մի մասը կամ ամբողջ տվյալները դիտարկվում են որպես բանալի և համեմատվում են "< " գործողությամբ: Որոնման բինար ծառում արմատային հանգույցի բանալին մեծ է նրա ձախ ենթածառի բոլոր հանգույցների բանալիներից և փոքր է կամ հավասար աջ ենթածառի բոլոր հանգույցների բանալիներից: Ընդորում՝ աջ և ձախ ենթածառերը որոնման բինար ծառեր են:

Օրինակ, հետևյալ ծառում 37 թվի փնտրումը պահանջում է չորս համեմատում սկսած արմատից:

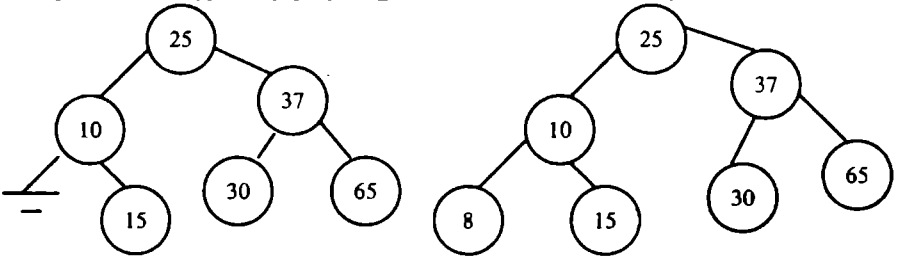


Նկ. 5.5 Որոնման բինար ծառ

Որոնման ծառում կարևոր գործողություններից են տվյալների որոնումը, ծառին նոր գագաթի ավելացումը և հեռացումը: Նշված գործողություններից հետո ստացված ծառը նույնպես պետք է լինի որոնման ծառ:

Օրինակ, դիտարկենք տրված ծառին 8 հանգույցի ավելացումը: Սկսելով 25 արմատային հանգույցից, որոշում ենք, որ 8 հանգույցը պետք է լինի 25 հանգույցի ձախ ենթածառում (8<25): 10 հանգույցում որոշում ենք, որ 8 հանգույցը պետք է լինի 10 հանգույցի ձախ ենթածառում, որը տվյալ պահին դատարկ է: 8 հանգույցն ավելացվում է ծառին որպես 10 հանգույցի ձախ որդի:

մինչև 8 հանգույցի ավելացումը *8 հանգույցն ավելացնելուց հետո*



Որոնման բինար ծառ ADT-ի նկարագրությունը հետևյալն է.

ADT BinarySearchTree

Տվյալներ

Որոնման բինար ծառ

Գործողություններ

Կոնստրուկտոր

մուտք – չկա

նախապայման – չկա

պրոցես - դատարկ որոնման բինար ծառի սկզբնարժեքավորում

ելք –չկա

վերջնապայման – դատարկ ծառ

Find

մուտք - որոնման բինար ծառում փնտրվող տարրը

նախապայման – չկա

պրոցես – տարրի փնտրում որոնման բինար ծառում

ելք – փնտրվող տարրի դիրքը բինար ծառում

վերջնապայման – չկա

Insert

մուտք - որոնման բինար ծառին ավելացվող տարրը

նախապայման – չկա

պրոցես – տարրի ավելացում որոնման բինար ծառին

ելք – չկա

վերջնապայման – ավելացված տարրով որոնման բինար ծառ

Delete

մուտք – հեռացվող տարր

նախապայման – չկա

պրոցես – տարրի հեռացում որոնման բինար ծառից

ելք – չկա

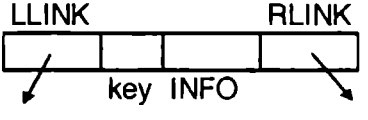
վերջնապայման - հեռացված տարրով որոնման բինար ծառ

վերջ ADT BinarySearchTree

Որոնման բինար ծառ ADT-ի նկարագրության մեջ ընդգրկված չեն բինար ծառերին բնորոշ Length, Empty, InorderTraverse, PreorderTraverse, PostorderTraverse, ClearBinaryTree, CopyTree գործողությունները:

5.6 ՈՐՈՆՄԱՆ ԲԻՆԱՐ ԾԱՌԻ ԻՐԱԿԱՆԱՑՈՒՄ

Ենթադրենք ծառը ներկայացված է կապակցված հանգույցների հավաքածուի տեսքով, որտեղ յուրաքանչյուր հանգույց ունի 5.6 նկարում բերված տեսքը:



Նկ. 5.6 Որոնման բինար ծառի հանգույցի կառուցվածքը

Նկարագրենք որոնման բինար ծառում իրականացվող գործողությունները:

Որոնում ըստ x բանալու

Մուտք – ծառի արմատի r ցուցիչ, փնտրվող տարրի x բանալի

Ելք – փնտրվող տարրի Q ցուցիչ

քայլ 1. [սկզբնարժեքավորում]
Q ← r

քայլ 2. [Q ≠ Λ]
եթե (Q ≠Λ) *ապա*
եթե (Key (Q) = x) *ապա* {"x –ը պատկանում է ծառին", ավարտ}
այլապես {
եթե (x < Key (Q)) *ապա* Q←LLINK (Q)
այլապես Q←RLINK (Q),
անցնել *քայլ 2.*
}

քայլ 3. [Q = Λ]
եթե (Q = Λ) *ապա* ավարտ

Տարրի ավելացում ըստ x բանալու

Բանալու կրկնություն չի թույլատրվում:
Մուտք – ծառի արմատի r ցուցիչ, ավելացվող տարրի x բանալի
Ելք – ավելացվող տարրով ծառ
Օժանդակ միջոցներ – ընթացիկ հանգույցի Q ցուցիչ, ընթացիկ հանգույցի ծնողի p ցուցիչ

քայլ 1. [սկզբնարժեքավորում]
Q ← r, p ← Λ

քայլ 2. [Q ≠ Λ]
եթե (Q ≠Λ) *ապա*
եթե (Key (Q) = x) *ապա* { "x –ը պատկանում է ծառին", ավարտ}
այլապես { p←Q, *եթե* (x < Key (Q))
ապա Q←LLINK (Q)
այլապես Q←RLINK (Q),
անցնել *քայլ 2*
}

քայլ 3. [Q = Λ]
եթե (Q = Λ) *ապա* {Q ⇐ AVAIL,
Key(Q) ← x,
լրացնել INFO (Q),
LLINK(Q)←Λ,
RLINK(Q)←Λ,
եթե (p = Λ) *ապա* r←Q

```
                    այլապես { եթե ( x < Key(p))
                                    ապա LLINK(p)←Q
                                    այլապես RLINK(p)←Q
                                    }
                }
```

Տարրի հեռացում ըստ x բանալու

Մուտք – ծառի արմատի r ցուցիչ, հեռացվող տարրի x բանալի
Ելք – որոնման բինար ծառ
Օժանդակ միջոցներ – Q - հեռացվող գագաթի ցուցիչ,
p - հեռացվող գագաթի ծնողի ցուցիչ,
S - փոխարինող հանգույցի ցուցիչ,
L- փոխարինող հանգույցի ծնողի ցուցիչ;

քայլ 1. [հեռացվող գագաթի որոնում]
bool Find(TreeNode < Key_T,info_T> *&r, Key_T x,
TreeNode < Key_T,info_T> * &Q,
TreeNode < Key_T,info_T> * &p)
եթե (Find() = false) *ապա* ավարտ
այլապես անցնել *քայլ 2*

քայլ 2. [հեռացվող հանգույցը տերև է]
եթե (LLINK(Q) = Λ և RLINK(Q) = Λ) *ապա*
եթե (p = Λ(ծառը կազմված է միայն արմատից))
ապա {r ←Λ, Q⇒ AVAIL, ավարտ}
այլապես { *եթե* (x < Key(p)) *ապա* LLINK(p) ←Λ
այլապես RLINK(p) ←Λ
Q⇒ AVAIL,
ավարտ}

քայլ 3. [եթե հեռացվող հանգույցը ունի միայն ձախ ենթածառ]
եթե (LLINK(Q) ≠ Λ և RLINK(Q) = Λ)
ապա եթե (p = Λ) (հեռացվում է արմատը)
ապա { r ← LLINK(Q),
Q⇒ AVAIL,
ավարտ }
այլապես
{*եթե* (x < Key(p)) *ապա* LLINK(p) ← LLINK(Q)
այլապես RLINK(p) ← LLINK(Q)
Q⇒ AVAIL, ավարտ }

քայլ 4. [եթե հեռացվող հանգույցը ունի միայն աջ ենթածառ]
եթե (LLINK(Q) = Λ և RLINK(Q) ≠ Λ)
ապա նույն քայլ 3-ը (r ← LLINK(Q)-ը փոխարինելով
r ← RLINK(Q)-ով)

քայլ 5. [եթե հեռացվող հանգույցն ունի և աջ, և ձախ ենթածառեր]
եթե (LLINK(Q) ≠ Λ և RLINK(Q) ≠ Λ)
ապա {L← Q, S ← LLINK(Q)}

քայլ 6. [շարժվել S հանգույցի աջ ենթածառերով]
քանի դեռ (RLINK(S) ≠ Λ)
{ L←S, S← RLINK(S) }

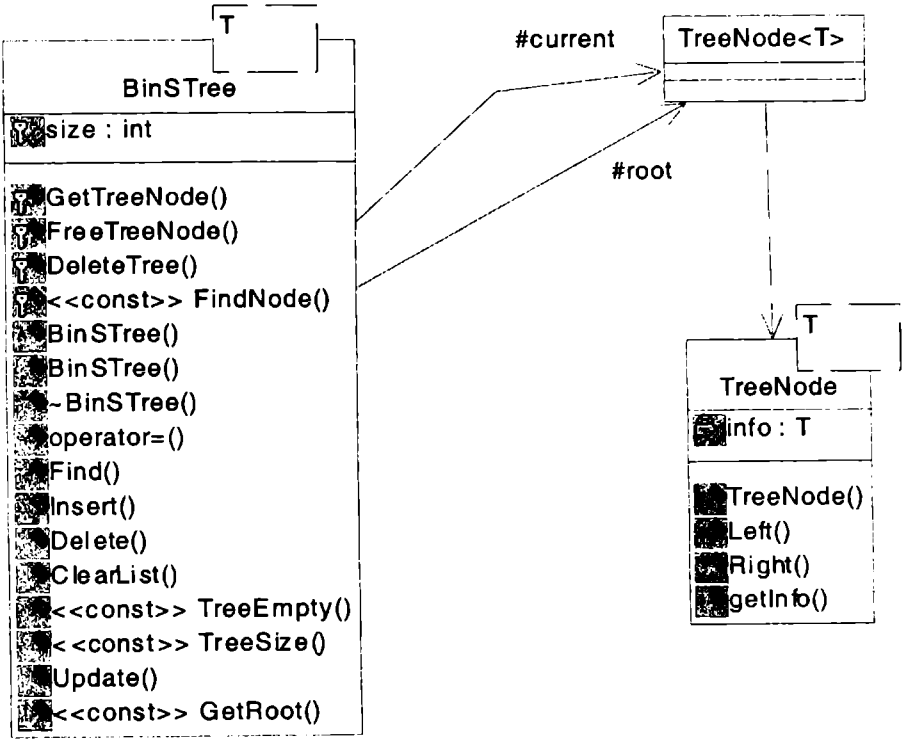
քայլ 7. [անջատել S հանգույցը և ձևավորել S հանգույցի ձախ և աջ կապերը]
եթե (L≠Q) { RLINK(L) ← LLINK(S),
LLINK(S) ← LLINK(Q),
RLINK(S) ← RLINK(Q)}
այլապես RLINK(S) ← RLINK(Q)

քայլ 8. [ձևավորել p հանգույցի համապատասխան կապը դնելով այն S-ի վրա]
եթե (p=Λ) *ապա* r← S
այլապես
եթե(Key(Q) < Key(p)) *ապա* LLINK(p) ← S
այլապես RLINK(p) ← S

քայլ 9. [հեռացնել Q հանգույցը]
Q⇒AVAIL, ավարտ

5.7 ՈՐՈՆՄԱՆ ԲԻՆԱՐ ԾԱՌԻ ԻՐԱԿԱՆԱՑՈՒՄԸ C++ ԼԵԶՎՈՎ

Որոնման բինար ծառն իրականացվում է BinSTree դասի միջոցով: Այն որոշվում է իր արմատի ցուցիչով (root), որի նախնական արժեքը NULL է: root-ին հասանելիությունը կատարվում է GetRoot-ի միջոցով: Դասում հայտարարված երկրորդ անդամ-տվյալը ընթացիկ տարրի ցուցիչն է (current): root-ը և current-ը օգտագործվում են Insert, Find և Delete գործողություններում: Դասի հաջորդ անդամ-տվյալը ցույց է տալիս ծառի ընթացիկ տարրերի քանակը (size), որի արժեքը փոխվում է Insert և Delete գործողություններով: Նկ. 5.7-ում ներկայացված է BinSTree դասի շաբլոնի UML-դիագրամը:



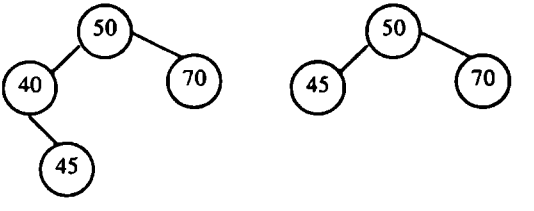
Նկ. 5.7 BinSTree շաբլոն դասի UML դիագրամը

```
template <class T>
class BinSTree {
protected:
     TreeNode <T> *root;       // արմատի ցուցիչ
                          // ընթացիկ հանգույցի ցուցիչ
     TreeNode <T> *current;
     int size ; // ծառի տարրերի քանակ
     // հիշողության տրամադրում, ազատում, տարրի փնտրում
     TreeNode<T>*GetTreeNode(const T& item,
       TreeNode<T>*lptr, TreeNode<T>*rptr);
     void FreeTreeNode (TreeNode <T> *p);
     void DeleteTree (TreeNode <T> *t);
     TreeNode <T>*FindNode (const T&item,
       TreeNode <T> *&parent) const;
```

```
public:
     // կոնստրուկտոր և դեստրուկտոր
     BinSTree(void);
     BinSTree (const BinStree <T>& tree);
     ~BinSTree(void);
     // վերագրման օպերատոր
     BinStree <T>& operator=(const BinStree <T>& rhs);
     // ցուցակների մշակման ստանդարտ մեթոդներ
     int Find (T & item);
     void Insert (const T & item);
     void Delete(const T & item);
     void ClearList (void);
     int ListEmpty (void) const;
     int ListSize (void) const;
     // ծառերին հատուկ մեթոդներ
        // ընթացիկ հանգույցի վերաարժեքավորում
     void Update (const T&item);
        // հասանելի է դարձնում արմատը
     TreeNode <T>*GetRoot (void)const;
};
```

Հետևյալ օրինակում ստեղծվում է չորս հանգույցով ծառ, այնուհետև հեռացվում է 40 արժեքով հանգույցը:

```
BinStree <int> T;
T. Insert (50);
T. Insert (40);
T. Insert (70);
T. Insert (45);
T. Delete (40);
```



Նկարագրենք BinSTree դասի որոշ ֆունկցիաներ:

Find գործողություն (տարրի փնտրում)

Find գործողությունն օգտագործում է FindNode փակ անդամ-ֆունկցիան, որը որպես պարամետր ստանալով բանալին, արմատից սկսում է շարժվել ձախ կամ աջ ենթածառով: FindNode-ը վերադարձնում է համընկնող հանգույցի ցուցիչը և նրա ծնողի ցուցիչը:

```
template <class T>
TreeNode <T>*BinSTree<T>::FindNode(const T& item,
TreeNode<T> * &parent)const {
```

```
    // անցնել ծառի հանգույցներով՝ սկսելով արմատից
    TreeNode<T> *t = root;
    parent = NULL; // արմատը ծնող չունի
    while (t! == NULL) {
        if (item == t→info)
            break;
        else { // ճորացնել ծնողի ցուցիչը և անցնել աջ
               // կամ ձախ
            parent = t;
            if (item < t → info)
                t = t → left;
            else
                t = t → right ;
        }
    }
    // վերադարձնել հանգույցի ցուցիչը,
    // NULL՝ եթե չի գտնվել
    return t;
}
```

Ծնողի մասին տեղեկությունը (parent) օգտագործվում է Delete գործողության կողմից: Find մեթոդում համընկնող հանգույցի հղումը տեղադրվում է item պարամետրում: Find-ը վերադարձնում է True(1) կամ False(0), նշելով փնտրումը հաջողվել է թե ոչ:

```
// փնտրել item-ը
template <class T>
int BinStree <T>:: Find(T & item) {
    TreeNode<T>* parent;
    // փնտրել item-ը և համընկնող հանգույցը նշել
    // որպես current
    current = FindNode (item, parent);
    // եթե գտնվել է, վերադարձնել True
    if (current ! = NULL)
        return 1;
    else // item-ը չի գտնվել, վերադարձնել False
        return 0;
}
```

Insert գործողություն (տարրի ավելացում)

Insert-ը որպես պարամետր ընդունում է նոր տարր և նրան տեղադրում ծառի մեջ իրեն համապատասխան տեղում: Սկսելով ար-

մատից և անցնելով ձախ ու աջ ենթածառերով ֆունկցիան որոշում է նոր տարրի ավելացման ճիշտ տեղը, որից հետո հանգույցն ավելացվում է այդ դիրքում որպես տերև:

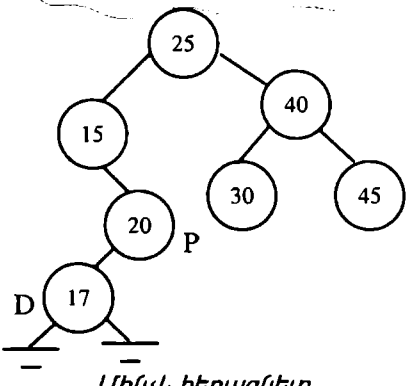
```
// item-ն ավելացնել որոնման ծառին
template <class T>
void BinSTree<T>:: Insert (const T & item) {
    // t-ն ընթացիկ հանգույցն է, parent-ը նախորդը
    TreeNode<T> *t=root, *parent= NULL, *newNode;
    while (t! = NULL) { // տեղադրել parent ցուցիչը և
                        // անցնել աջ կամ ձախ
        parent=t;
        if (item < t → info)
            t = t → left;
        else
            t = t → right;
     }
     // ստեղծել նոր հանգույց
     newNode = Gettreenode(item, NULL, NULL) ;
     // եթե ծնող չկա, գրանցել որպես արմատային հանգույց
     if (parent == NULL)
         root = newNode;
     else    if (item < parent → info)
         parent → left = newNode;
     else
         parent → right = newNode;
     // ավելացված հանգույցի հասցեն հիշել current-ում
     // և size-ը 1-ով մեծացնել
     current = newNode;
     size++;
}
```

Delete գործողություն (տարրի հեռացում):

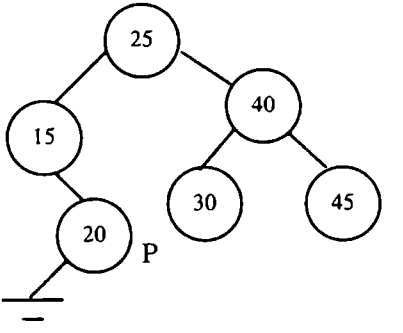
Delete գործողությունը ծառից հեռացնում է տված բանալիով հանգույցը: Սկզբում FindNode-ի միջոցով գտնվում է ծառում այդ հանգույցի տեղը: Եթե փնտրվող հանգույցը բացակայում է, ապա հեռացման գործողությունն ավարտված է: Հանգույցի հեռացման ժամանակ պետք է որոշել, թե որտեղ միացնել հեռացվող հանգույցի որդիներին, որ պահպանվի որոնման բինար ծառի կառուցվածքը:

FindNode-ը վերադարձնում է հեռացվող D հանգույցի DNodePtr ցուցիչը, իսկ PNodePtr-ը հեռացվող հանգույցի ծնողի ցուցիչն է: Պետք է գտնվի R փոխարինող հանգույց, որը կմիացվի ծնողին և կգրավի հեռացվող հանգույցի տեղը: Փոխարինող R հանգույցի ցուցիչը RnodePtr-ն է: Հեռացվող հանգույցի որդիների քանակից կախված, փոխարինող հանգույցի փնտրման ալգորիթմը պետք է դիտարկի մի քանի դեպք: Նկատենք, որ եթե ծնողի հղումը NULL է, ապա հեռացվում է արմատը:

Դեպք 1. D հանգույցը որդիներ չունի, այսինքն տերև է:



Մինչև հեռացնելը

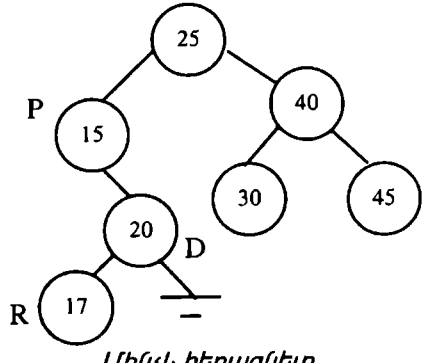


17-ը հեռացնելուց հետո

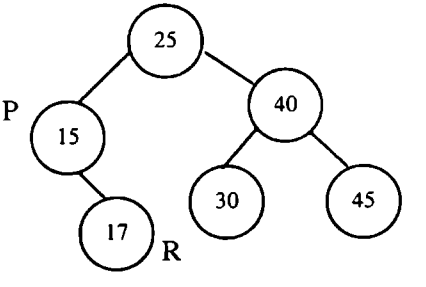
Ծառի նորացումը տեղի է ունենում հետևյալ ձևով.

```
RnodePtr = NULL;
PNodePtr → left = RNodePtr;
```

Դեպք 2. D հանգույցն ունի ձախ որդի, բայց չունի աջ որդի: Այս դեպքում D հանգույցի ձախ ենթածառը միացվում է ծնողին:



Մինչև հեռացնելը

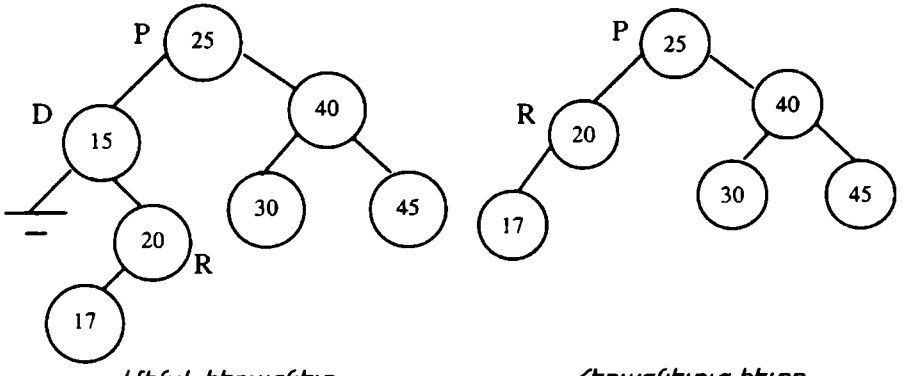


Հեռացնելուց հետո

```
RNodePtr = DNodePtr → left;
PNodePtr → right = RNodePtr;
```

Դեպք 3. D հանգույցը ունի աջ որդի, բայց չունի ձախ որդի: Միացնել D-ի աջ ենթածառը իր ծնողին:

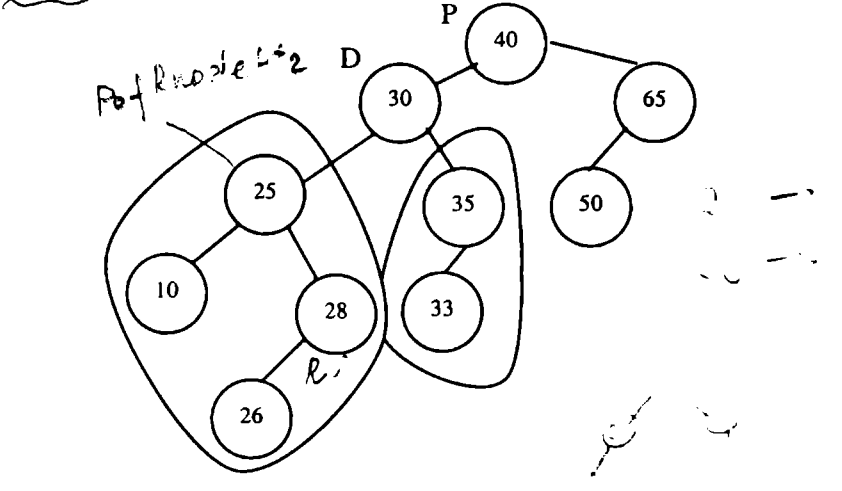
Հեռացնել 15 հանգույցը, R հանգույց է հանդիսանում D-ի աջ որդին:



Մինչև հեռացնելը *Հեռացնելուց հետո*

```
RnodePtr = DnodePtr → right;
PnodePtr → left = RnodePtr;
```

Դեպք 4: Երկու որդի ունեցող հանգույցի հեռացում:



Օրինակ, 30 հանգույցն հեռացնելիս, որպես նրան փոխարինող պետք է ընտրել ձախ ենթածառի ամենաաջ հանգույցը: Դա հեռացվողից փոքր հանգույցներից ամենամեծն է: Պետք է այդ R հանգույցն անջատել ծառից, նրա ձախ ենթածառը միացնել իր ծնողին, հետո R-ը դնել հեռացվող հանգույցի տեղում:

Բերված օրինակում փոխարինող հանգույցը 28-ն է: Իր ձախ 26 որդուն կմիացնենք 25 ծնողին և 30 հեռացվող հանգույցը կփոխարինենք 28-ով:

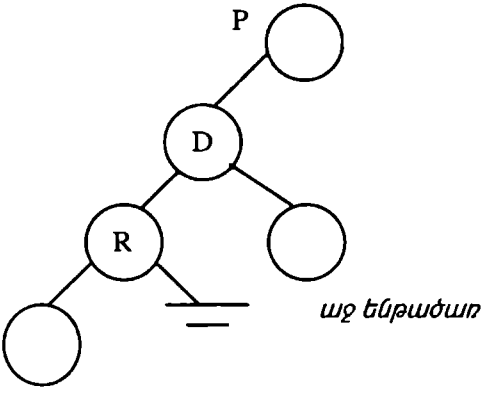
Ձախ ենթածառի ամենաաջ հանգույցը գտնում ենք հետևյալ կերպ.

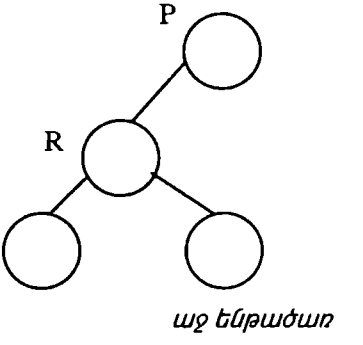
1. Քանի որ փոխարինող R հանգույցը փոքր է քան հեռացվող D-ն, պետք է անցնել D-ի ձախ ենթածառին (իջնել 25 հանգույցին):
2. Քանի որ R-ը ձախ ենթածառի մեծագույն հանգույցն է, այն գտնելու համար պետք է իջնել աջ ենթածառով՝ հիշելով նախորդը PofRNodePtr:

Մեր օրինակում իջնում ենք 28 հանգույց, իսկ PofRNodePtr-ը 25-ի ցուցիչն է:

Աջ ենթածառով իջնելիս հնարավոր է երկու դեպք:

ա) Եթե աջ ենթածառը դատարկ է, ապա D-ի աջ ենթածառը կմիացնենք R-ին որպես աջ ենթածառ, իսկ R-ը կմիացնենք հեռացվող հանգույցի P ծնողին:



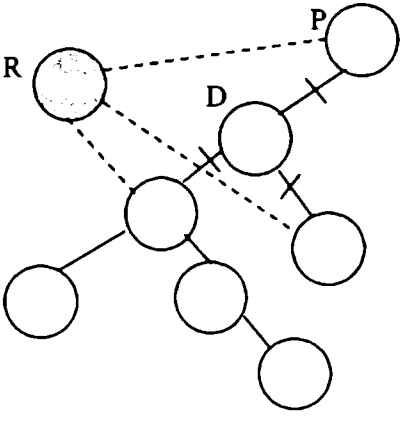



```
RNodePtr → right = DNodePtr → right;
PNodePtr → left = RNodePtr;
```

բ) Եթե աջ ենթածառը դատարկ չէ, ապա նրանով վայրէջքն ավարտվում է կամ տերևի, կամ միայն ձախ ենթածառ ունեցող հանգույցի վրա: Ամեն դեպքում պետք է R-ն անջատել ծառից և R-ի ձախ որդին միացնել PofRNodePtr ծնող հանգույցին:

```
PofRNodePtr → right = RNodePtr → left;
```

Ալգորիթմն ավարտվում է հեռացվող D հանգույցը R-ով փոխարինելով: Սկզբում D-ի որդիները միացվում են R-ին, որպես որդիներ: Այնուհետև R հանգույցը փոխարինում է D-ին՝ դառնալով D-ով սկսվող ենթածառի արմատը: Վերջում R-ը միացվում է P ծնող հանգույցին՝ եթե այն գոյություն ունի, հակառակ դեպքում R-ը դառնում է ծառի արմատը:




```
// D-ի որդիների միացում R-ին
RNodePtr → left = DNodePtr → left;
RNodePtr → right = DNodePtr → right;
// արմատային հանգույցի հեռացում և նոր արմատի տեղադրում
if (PNodePtr == NULL)
    root = RNodePtr;
    //կցել R-ը P-ին ճիշտ կողմից
    else if (DNodePtr → info < PNodePtr → info)
            PNodePtr → left = RNodePtr;
        else
            PNodePtr → right = RNodePtr;
```

5.8 ՀԱՎԱՍԱՐԱԿՇՌՎԱԾ ԾԱՌԵՐ

Ինչպես տեսանք, որոնման ծառերը շատ հարմար են տվյալների փնտրման և արագ հայտնաբերման համար: Նրանց բարձրությունը լավագույն դեպքում $O(\log_2 n)$ է: Սակայն տվյալների որոշ դասավորության դեպքում նրանց բարձրությունը կարող է լինել $O(n)$ և այս ժամանակ տվյալներին դիմումը կդանդաղի:

Այժմ դիտարկենք որոնման բինար ծառերի հավասարակշռված (բալանսավորված) տարբերակները: Նրանց բարձրությունը չի գերազանցում $O(\log_2 n)$-ը: Այս դեպքում նոր գագաթի հեռացման կամ ավելացման համար անհրաժեշտ գործողությունների քանակը $O(\log_2 n)$ է:

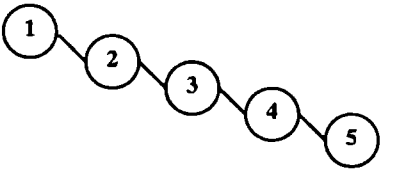
Հավասարակշռված ծառերում ապահովվում է գագաթների դասավորվածության համաչափությունը: Դիտարկենք հավասարակշռված ծառերի երկու տեսակ՝ իդեալապես հավասարակշռված ծառեր և AVL(Adel'son-Vel'skii, Landis) ծառեր:

Ծառը իդեալապես հավասարակշռված է, եթե նրա յուրաքանչյուր գագաթի աջ և ձախ ենթածառերի գագաթների քանակները միմյանցից տարբերվում են ոչ ավել քան մեկով:

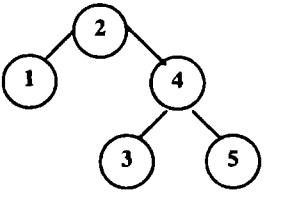
Ծառը AVL է, եթե նրա յուրաքանչյուր գագաթի աջ և ձախ ենթածառերի բարձրությունները միմյանցից տարբերվում են ոչ ավել քան մեկով:

Օր. 1. A[5] = {1, 2, 3, 4, 5 }

Որոնման ծառը կլինի

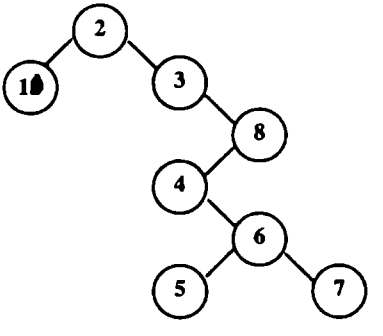


AVL ծառը կլինի

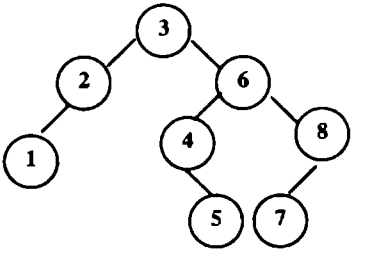


Օր. 2. B[8] = {2, 3, 8, 4, 1, 6, 5, 7}

Որոնման ծառը կլինի



AVL ծառը կլինի



{ Նոր տարր ավելացնելուց կամ հեռացնելուց հետո ծառի հավասարակշռվածությունը պահպանելու համար անհրաժեշտ է ստուգումներ կատարել ծառի յուրաքանչյուր գագաթի համար:} Այդ նպատակով, ծառի հանգույցի կառուցվածքում ավելացել է bal դաշտը: }

left	data	bal	right

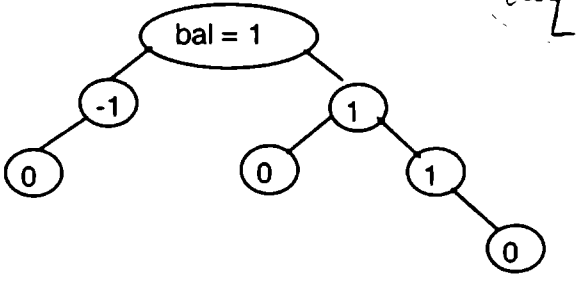
Ինչպես տեսնում ենք, ծառի գագաթները բացի տվյալից (data), ձախ և աջ ցուցիչներից (left, right) պարունակում են նաև տվյալ գագաթի հավասարակշռվածությունը ցույց տվող bal ենթադաշտը, որը տվյալ գագաթի աջ ենթածառի բարձրության (h_l) և ձախ ենթածառի բարձրության (h_R) տարբերությունն է:

Ստորև բերված է AVL ծառը իր հանգույցների հավասարակշռման ցուցանիշներով:

bal = 0, եթե $h_l = h_R$

bal = -1, եթե $h_l > h_R$

bal = 1, եթե $h_l < h_R$



իդեալապես հավասարակշռված ծառի կառուցումը

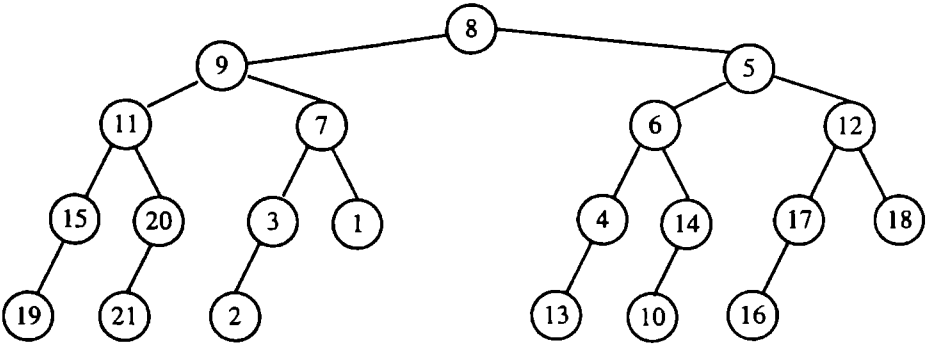
Ստանդարտ մուտքային ֆայլից ներածվում են ծառի գագաթների n քանակը և գագաթների արժեքները: Ծառի գագաթները հավասարաչափ տեղաբաշխելով ձախ և աջ ենթածառերում, յուրաքանչյուր մակարդակում, բացի վերջինից, կառուցվում են մաքսիմալ թվով գագաթներ: Ակնհայտ է, որ կկառուցվի n գագաթից կազմված մինիմալ բարձրությամբ ծառ: Դա կարելի է կատարել հետևյալ ռեկուրսիվ ալգորիթմի միջոցով.

- Վերցնել ներածվող գագաթը որպես արմատ,
- Կառուցել արմատի ձախ ենթածառը nl = n/2 գագաթներով,
- Կառուցել արմատի աջ ենթածառը nr = n-nl-1 գագաթներով:

Օրինակ, ենթադրենք, որ 21 գագաթից կազմված իդեալապես հավասարակշռված ծառ կառուցելու համար ներածվում են հետևյալ թվերը.

21 8 9 11 15 19 20 21 7 3 2 1 5 6 4 13 14 10 12 17 16 18

Ալգորիթմի կիրառման արդյունքում կկառուցվի հետևյալ ծառը:



Ալգորիթմն իրականացվում է հետևյալ ծրագրով:

```
const int indent = 6;
struct Node {
            int data;
            Node * left, * right;
};
int n;
typedef Node * PNode;
PNode root;
// n գագաթով իդեալապես բալանսավորված ծառի կառուցումը
PNode Tree(int n) {
      PNode newnode;
      int x, nl, nr;
      if (n==0) newnode = NULL;
      else {
            nl = n / 2; nr = n-nl-1;
            cin >> x;
            newnode = new Node;
            newnode->data = x;
            newnode->left = Tree(nl);
            newnode->right= Tree(nr);
      };
      return newnode;
}
```

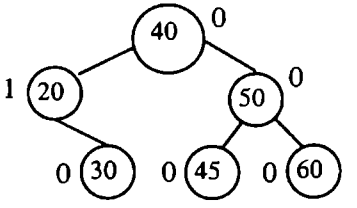
```
// տպել ծառը 90 աստիճանով շրջված դեպի աջ
Printtree (PNode t, int level) {
      int i;
      if (t != NULL) {
            Printtree (t->right, level + 1);
            for (i = 1; i < indent * level;
                  i++) cout <<' ';
            cout << t->data << endl;
            Printtree (t->left, level + 1);
      }
}

int main(){
      // ներածվող առաջին թիվը գագաթների քանակն է
      cin >> n;
      root = Tree(n);
      Printtree(root, 0);
      return 0;
}
```

AVL ծառի կառուցումը

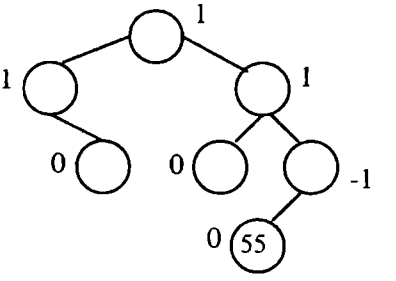
AVL ծառի կառուցումը կատարվում է դատարկ ծառին նոր գագաթներ ավելացնելով: Ենթադրենք r-ը AVL ծառի արմատն է, իսկ L-ը և R-ը նրա ձախ և աջ ենթածառերն են: Եթե որևէ ենթածառում նոր գագաթի ավելացումը չի փոխում այդ ենթածառի բարձրությունը, ապա ծառի հավասարակշռվածությունը պահպանվում է: Այժմ ենթադրենք, որ նոր գագաթի ավելացումը բերում է նրա համապատասխան ենթածառի բարձրության մեկով ավելացմանը: Հնարավոր է երեք դեպք.

1. *$h_L=h_R$. Գագաթն ավելացվում է R-ենթածառում:* Այս դեպքում L-ը և R-ը կունենան տարբեր բարձրություններ, սակայն ծառի հավասարակշռվածությունը չի խախտվի:

Օրինակ, 40-50-60 ճանապարհի բոլոր հանգույցները հավասարակշռված են: 55-ի ավելացումից հետո նշված հանգույցների հավասարակշռվածության ցուցանիշները փոխվում են, սակայն ծառի հավասարակշռվածությունը պահպանվում է:

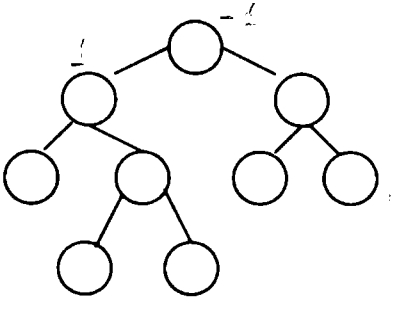


Մինչև գագաթի ավելացումը

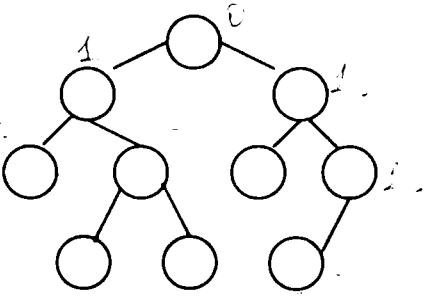


Գագաթի ավելացումից հետո

2. *$h_L > h_R$. Գագաթն ավելացվում է R-ենթածառում:* Այս դեպքում L-ի և R-ի բարձրությունները կհավասարվեն, և ծառի հավասարակշռվածությունը կրկին չի խախտվի:



Մինչև գագաթի ավելացումը



Գագաթի ավելացումից հետո

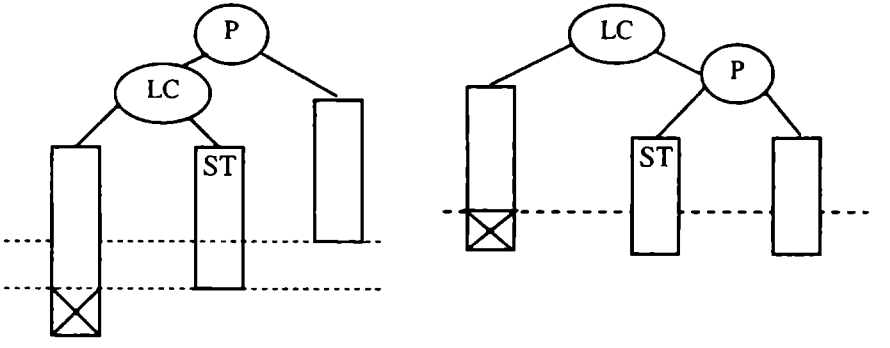
Այս երկու դեպքերում էլ գագաթի ավելացումը չի խախտում ծառի հավասարակշռվածությունը, և ծառի վերակազմավորման անհրաժեշտություն չկա, սակայն հարկավոր է լինում վերափոխել փնտրման ճանապարհին հանդիպող գագաթների հավասարակշռման ցուցանիշները:

3. *$h_L > h_R$. Գագաթն ավելացվում է L-ենթածառում:* Այս դեպքում ծառի հավասարակշռվածությունը կխախտվի, և ծառն անհրաժեշտ է վերակազմավորել՝ միաժամանակ փոփոխելով փնտրման ճանապարհին եղած ծառի գագաթների հավասարակշռվածության ցուցանիշները: Ծառի վերակազմավորումը կատարվում է նրա երկու կամ երեք գագաթների մեկ կամ երկու պտույտ կատարելով:

Դիտարկենք այս դեպքն ավելի մանրամասն:

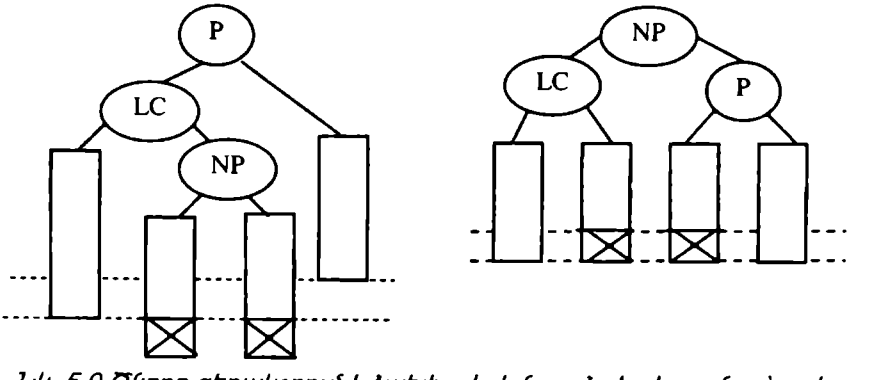
Տարբերակների ուշադիր քննարկման դեպքում պարզ է դառնում, որ այստեղ գոյություն ունեն ըստ էության միայն երկու իրարից տարբեր հիմնական դեպքեր: Մնացած դեպքերը հանգում են այս երկուսին՝ հաշվի առնելով սիմետրիկությունը:

Առաջին հիմնական դեպքն առաջանում է այն ժամանակ, երբ X-ով նշված դիրքում գագաթ ավելացնելուց հետո, ծնողն (P) ու իր ձախ ժառանգը (LC) գերակշռում են ձախից: Այս դեպքում կատարվում է պտույտ և LC-ն փոխարինում է իր ծնողին, որը դառնում է աջ որդի: LC-ի նախորդ դիրքի աջ ենթածառը (ST) միանում է P-ին որպես ձախ ենթածառ (նկ. 5.8):



Նկ. 5.8 Ծնողն ու իր ձախ ժառանգը գերակշռում են ձախից

Երկրորդ հիմնական դեպքում ծնողը (P) գերակշռում է ձախից, իսկ նրա (LC) ձախ ժառանգը՝ աջից: Այս դեպքում NP-ն փոխարինում է (P) ծնողին, որն էլ դառնում է (NP)-ի աջ ժառանգ (նկ. 5.9):

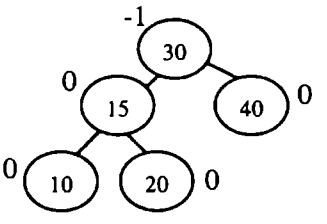


Նկ. 5.9 Ծնողը գերակշռում է ձախից, իսկ նրա ձախ ժառանգը՝ աջից (իրականում պետք է դիտարկել X-ով նշված դիրքերից միայն մեկը)

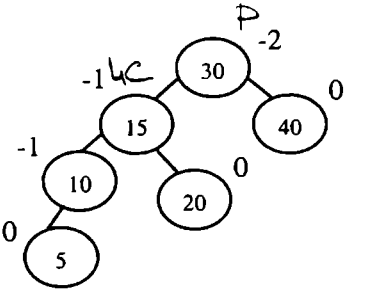
Ինչպես տեսնում ենք, երկու դեպքում էլ պարզ ձևափոխությունների շնորհիվ ծառի հավասարակշռվածությունը վերականգնվում է: Ընդ որում, ծառի գագաթները և ենթածառերը տեղաշարժվում են միայն ուղղահայաց ուղղությամբ: Հորիզոնական ուղղությամբ տեղաշարժեր չեն կատարվում:

Օրինակներ.

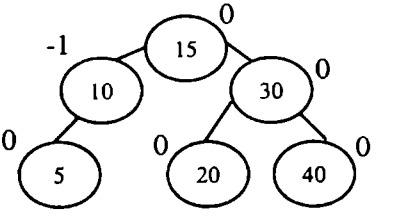
Դեպք 1. Ավելացնենք 5 հանգույցը AVL ծառին:



Նախնական ծառը

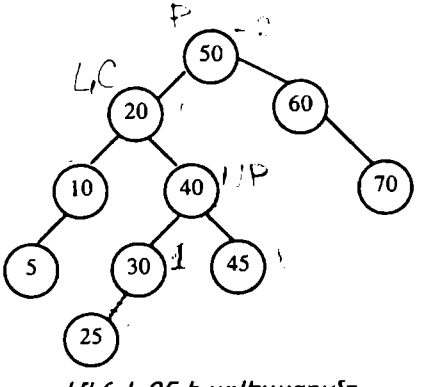


5-ը ավելացնելուց հետո

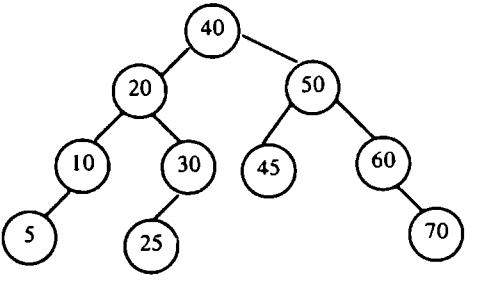


Բալանսավորելուց հետո

Դեպք 2. Ավելացնենք 25 հանգույցը AVL ծառին:



Մինչև 25-ի ավելացումը



25-ն ավելացնելուց և բալանսավորելուց հետո

Գագաթի ավելացումը կատարվում է հետևյալ երեք հաջորդական քայլերով:

1. Անցնել փնտրման ճանապարհով, որպեսզի համոզվենք որ ծառում չկա տրված արժեքով գագաթ:
2. Ավելացնել նոր գագաթը և որոշել հավասարակշռվածության ցուցանիշները:
3. Հետ գալ փնտրման ճանապարհով, յուրաքանչյուր գագաթի համար ստուգելով հավասարակշռվածության ցուցանիշը: Անհրաժեշտության դեպքում կատարել հավասարակշռում:

Ենթադրենք, որ p գագաթի ձախ ենթածառից պրոցեսը վերադառնում է p-ին և նշվում է, որ բարձրությունն ավելացել է: Հնարավոր է երեք դեպք, կախված այն բանից, թե մինչ գագաթի ավելացումը ինչպիսին էին ենթածառերի բարձրությունները.

1. $h_L < h_R$ (bal = 1) - այս դեպքում p-ում հավասարակշռվածությունը պահպանվում է.
2. $h_L == h_R$ (bal = 0) - այս դեպքում ձախ ենթածառը սկսում է գերակշռել.
3. $h_L > h_R$ (bal = -1) - անհրաժեշտ է հավասարակշռում:

Երրորդ դեպքում ձախ ենթածառի գագաթի հավասարակշռման ցուցանիշից կարելի է եզրակացնել առաջին հիմնական դեպքն է, թե երկրորդ: Եթե այդ գագաթի ձախ ենթածառը աջից բարձր է, ուրեմն առաջին հիմնական դեպքն է, հակառակ դեպքում՝ երկրորդ հիմնական դեպքն է: Ակնհայտ է, որ այդ գագաթի աջ և ձախ ենթածառերի բարձրությունները իրար հավասար լինել չեն կարող, որովհետև հայտնի է, որ նոր գագաթի ավելացումը բերել է ծառի բարձրության ավելացմանը: Ծառի հավասարակշռումը կատարվում է համապատասխան ցուցանիշների հաջորդական վերաարժեքավորմամբ (1 կամ 2 պտույտով):

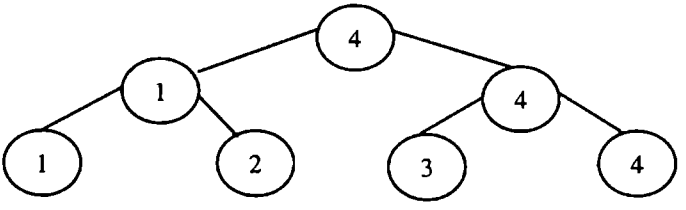
5.9 ԲԻՆԱՐ ԾԱՌԵՐԻ ԿԻՐԱՌՈՒԹՅՈՒՆՆԵՐ

Մրցույթային կարգավորում (տուրնիր)

Բինար ծառերն ունեն մեծ կիրառություններ այնպիսի խնդիրներում, որտեղ ծառի յուրաքանչյուր հանգույց ներկայացնում է երկու հնարավոր ելք ունեցող իրադրություն: Մասնավորապես, այդպիսի

խնդիր է սպորտային մրցույթը: Յուրաքանչյուր ոչ տերևային հանգույց համապատասխանում է երկու խաղացողների միջև հանդիպման հաղթողին, իսկ տերևային հանգույցները նշում են մասնակիցների սկզբնական կազմը և նրանց բաշխումն ըստ զույգերի: Օրինակ, չորս մասնակցի դեպքում հաղթողին որոշելու համար կպահանջվի ընդամենը երեք մատչ.

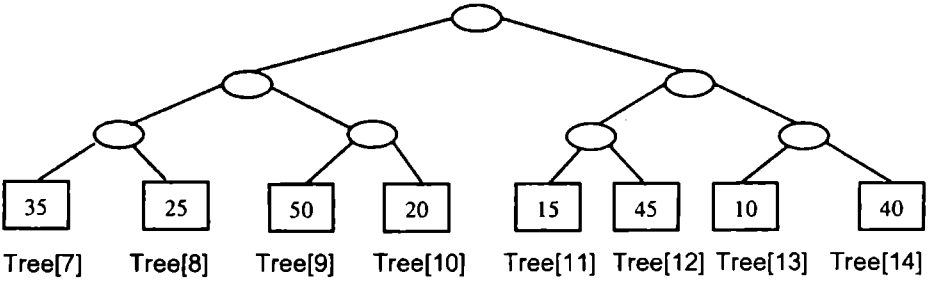
հաղթող



Մրցույթային ծառը կարող է օգտագործվել N տարրով ցուցակի կարգավորման համար:

Ենթադրենք ունենք ծառ, որն իր տերև հանգույցներում պարունակում է N տարր: Այդ տարրերը տեղավորված են k-րդ մակարդակում, որտեղ $2^k \geq N$: Օրինակ, հետևյալ ծառով տրվում է N=8 ամբողջ թվերից կազմված զանգվածի սկզբնական վիճակը: Զանգվածի տարրերը տեղավորված են 3-րդ մակարդակում՝ $2^3 = 8$:

A[8]={ 35, 25, 50, 20, 15, 45, 10, 40 }



Երկրորդ մակարդակից սկսվում է խաղը և յուրաքանչյուր զույգի փոքրագույն արժեքը տեղադրվում է ծնող հանգույցում: Օրինակ, Tree[7] և Tree[8]-ից հաղթում է փոքր արժեքով տարրը՝ 25-ը, և այն տեղադրվում է Tree[3]-ում:

Նմանատիպ համեմատումներ կատարվում են նաև երկրորդ և առաջին մակարդակներում: Վերջին համեմատման արդյունքում ամենափոքր տարրը տեղադրվում է ծառի արմատում՝ զրոյական մակարդակում: Այդ պահին այն հեռացվում է ծառի իր հին դիրքից և պատճենվում է զանգվածում: Այսպիսով, 10-ը գրվում է A[0]-ում, որից հետո ծառը նորացվում է՝ հաջորդ ամենափոքր տարրը գտնելու համար: Երկրորդ ամենափոքր տարրը գրվում է A[1]-ում, և այդպես շարունակ: Պրոցեսը շարունակվում է այնքան ժամանակ, մինչև բոլոր տերևները հեռացվեն: Մեր օրինակում ստացվում է 10, 15, 20, 25, 35, 40, 45, 50 կարգավորված ցուցակը:

Մրցույթային կարգավորման ալգորիթմի արդյունավետությունը $O(n\log_2 n)$ է:

Որոնման ծառերի և AVL ծառերի գնահատման բնութագրերի համեմատում

Համեմատվում են հավասարակշռված և որոնման բինար ծառերը, որոնցից յուրաքանչյուրը պարունակում է n հատ միևնույն պատահական թվերը: Թվերը պահվում են զանգվածում և տեղադրվում են երկու ծառերում: Այդ երկու ծառերում կատարվում է զանգվածի յուրաքանչյուր տարրի փնտրում, և գտնվում են փնտրման միջին երկարությունները: Արդյունքում, AVL ծառերի համար փնտրման բնութագրերն ստացվում են ավելի լավը: Վատագույն դեպքում որոնման բինար ծառը 1000 տարրի համար ունի միջին խորություն՝ 500, իսկ AVL ծառի միջին խորությունը 9 է:

```
// զանգվածի, որոնման բինար ծառի և AVL ծառի մեջ գրենք
// 0-999 միջակայքի միևնույն պատահական թվերը
void SetupLists(BinSTree <int> &Tree1,AVLTree<int>
                  &Tree2,int A[ ],int n) {

     int i;
     RandomNumber rnd;
     for(i=0; i<n; i++){
          A[ i ] = rnd.Random(1000);
          Tree1.Insert(A[ i ]);
          Tree2.Insert(A[ i ]);
     }
}
```

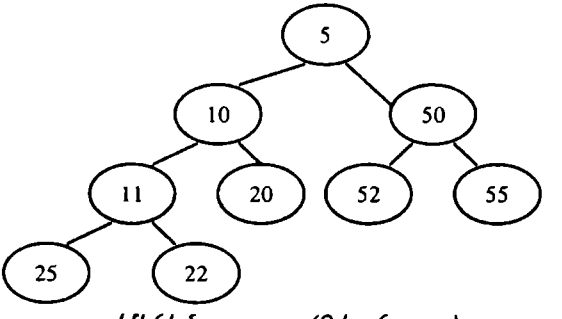
```
//t ծառում item տարրի փնտրում: Usացվում է փնտրման
// գումարային երկարությունը
template <class T>
void PathLength(TreeNode <T> *t,long &totallength,
    int item) {
    if(t==NULL || t -> data==item)
       return;
    else     {
       totallength ++;
       if(item < t ->data)
         PathLength(t ->Left(), totallength, item);
       else
         PathLength(t ->Right(),totallength, item);
    }
}
void  main(){
    BinSTree <int> binTree;
    AVLTree <int> avlTree;
    int *A;
    // փնտրման ճանապարհների գումարային
    // երկարությունները ծառերում
    long totalLengthBintree = 0,
    totalLengthAVLTree=0;
    int n, i;
    cout << "Ներմուծել ծառի գագաթների քիվը";
    cin >> n;
    A=new int[ n] ;
    // պատահական թվերով արժեքավորել զանգվածը և երկու
    // ծառերը
    SetupLists(binTree, avlTree, A, n);
    for (i=0; i<n; i++) {
        PathLength(binTree.GetRoot(),
        totalLengthBintree, A[ i ]);
        PathLength((TreeNode<int>*)avlTree.getRoot(),
        totalLengthAVLTree, A[ i ]);
    }
    cout<<float(totalLenghBintree)/n<<endl;
    cout<<float(totalLengthAVLTree)/n<<endl;
}
```

n=1000-ի դեպքում ծրագիրն աշխատելիս, որոնման բինար ծառում փնտրման միջին երկարությունն ստացվում է 10.256, իսկ հավասարակշռված ծառում՝ 7.901:

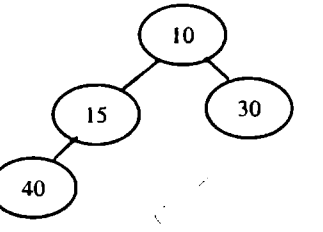
5.10 ԲՈՒՐԳԵՐ

Բուրգը ավարտուն բինար ծառ է, որն ունի հանգույցների որոշակի կարգավորվածություն: Լինում են մաքսիմալ և մինիմալ բուրգեր: Մաքսիմալ բուրգում կամայական հանգույցի արժեքը (բանալին) մեծ է կամ հավասար իր որդիների արժեքներից (եթե որդիները կան), իսկ մինիմալ բուրգում՝ փոքր է կամ հավասար իր որդիների արժեքներից:

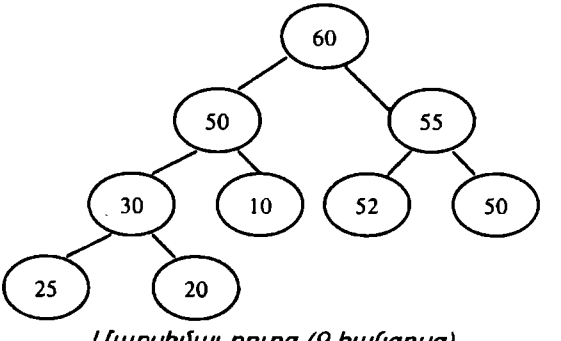
Մաքսիմալ բուրգում արմատի արժեքն ամենամեծն է, իսկ մինիմալում՝ ամենափոքրը: Օրինակներ.



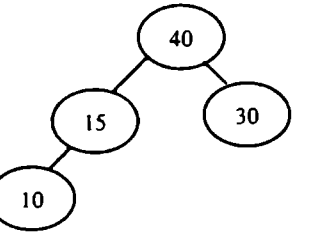
Մինիմալ բուրգ (9 հանգույց)



Մինիմալ բուրգ (4 հանգույց)



Մաքսիմալ բուրգ (9 հանգույց)



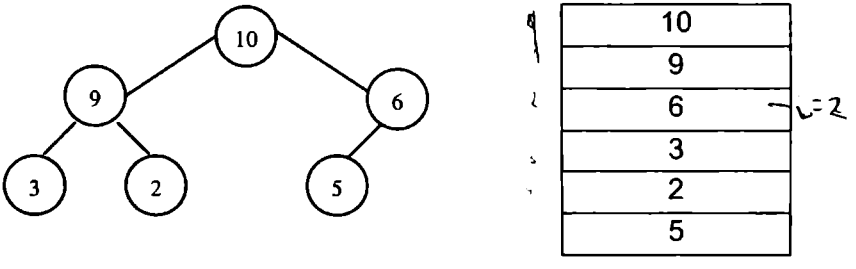
Մաքսիմալ բուրգ (4 հանգույց)

Բուրգում հիմնական գործողություններն են տարրի ավելացումը և հեռացումը: Բուրգում գործողություններ կատարելիս պահանջվում է, որ պահպանվի բուրգային կարգավորվածությունը:

5.11 ԲՈՒՐԳԻ ԻՐԱԿԱՆԱՑՈՒՄ

Դիտարկելու ենք մաքսիմալ բուրգերը: Ուշադրություն դարձնենք, որ բուրգում նախապես հայտնի չէ թե որևէ գագաթի որդիներից որի արժեքն է ավելի մեծ:

Եթե նախապես հայտնի է բուրգի տարրերի առավելագույն քանակը, ապա բուրգի իրականացման համար կարելի է կիրառել զանգված (նկ.5.15):



Նկ. 5.10 Բուրգի իրականացում զանգվածի միջոցով

Տարրի հեռացում (heapDelete)

Բուրգից տարր հեռացնելիս վարվում ենք հետևյալ կերպ.

- բուրգի արմատի արժեքը պահում ենք վերադարձի համար,
- արտագրում ենք բուրգի վերջին տարրը արմատի մեջ,
- հեռացնում ենք վերջին հանգույցը,
- ստացված ծառը վերակարգավորում ենք բուրգ ստանալու նպատակով:

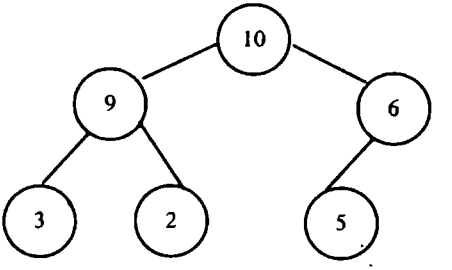
Վերակարգավորումը սկսվում է ծառի արմատից և կատարվում է հետևյալ կերպ.

ա) Եթե A գագաթի արժեքը (առաջին քայլում ծառի արմատը) փոքր է իր որդիներից մեկի արժեքից, ապա կատարվում է A գագաթի և այդ որդու արժեքների տեղափոխություն (A գագաթն իջնում է մեկ մակարդակ ներքև): Անցնել ա)–ին:

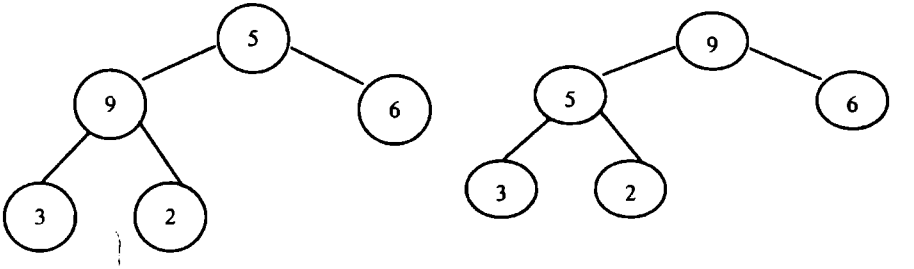
բ) Եթե A գագաթի արժեքը փոքր է իր երկու որդիների արժեքներից, ապա A գագաթի արժեքը տեղափոխվում է ավելի մեծ արժեք ունեցող որդու հետ (A գագաթն իջնում է մեկ մակարդակ ներքև): Անցնել ա)-ին:

գ) Եթե A-ն չունի որդիներ, կամ A-ի արժեքը իր որդիներից փոքր չէ՝ ավարտել:

Օրինակ, հետևյալ բուրգից հեռացվում է մեկ տարր և արդյունքը վերակարգավորվում է, բուրգ ստանալու նպատակով:



Հեռացնենք 10-ը և վերակարգավորենք.



Մեթոդի արդյունավետությունը $O(\log_2 n)$ է:

Heap դասի անդամ տվյալներն են.

items - բուրգի տարրերի զանգվածը,

size - բուրգում առկա տարրերի քանակը:

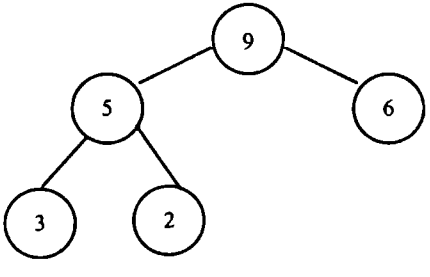
```
//տարրի հեռացում
void Heap::heapDelete(HeapItemType & rootItem) {
    if (heapIsEmpty()) cout << "դատարկ բուրգ";
    else {
          rootItem = items[ 0] ;
          items[ 0]  = items[ --size] ;
          heapRebuild (0);
    }
}
```

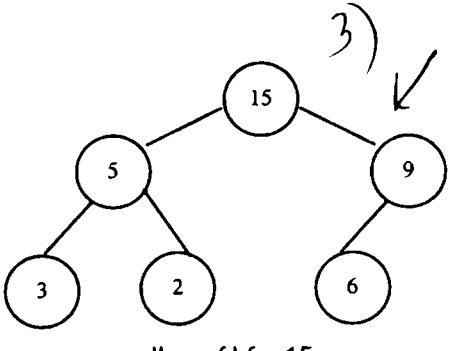
```
// վերակարգավորում
void Heap:: heapRebuild (int root) {
    int child = 2 * root + 1; // ձախ որդու ինդեքսը
    if (child < size) {
        int rightChild = child + 1; // աջ որդու
                                    // ինդեքսը
        if ((rightChild<size) && (items[ rightChild] >
                                    (items[ child))
        child = rightChild; // ավելի մեծ արժեքով
                            // որդու ինդեքսը
        if (items[ root] < items[ child] ) {
             HeapItemType temp = items[ root] ;
             items[ root] = items[ child] ;
             items[ child] = temp;
             heapRebuild (child); // ճոր ենթածառի
                                  // վերակարգավորում
        }
    }
}
```

Տարրի ավելացում (heapInsert)

Բուրգում նոր տարր ավելացնելիս այն ավելացվում է բուրգի վերջում: Այնուհետև, քանի դեռ այդ տարրն ունի ծնող, և իր արժեքը փոքր է ծնողի արժեքից՝ ծնողի և որդու արժեքները տեղափոխվում են: Սա հեշտ է իրականացնել, քանի որ i-րդ տեղում գտնվող տարրի ծնողը items[(i-1)/2] -ն է:

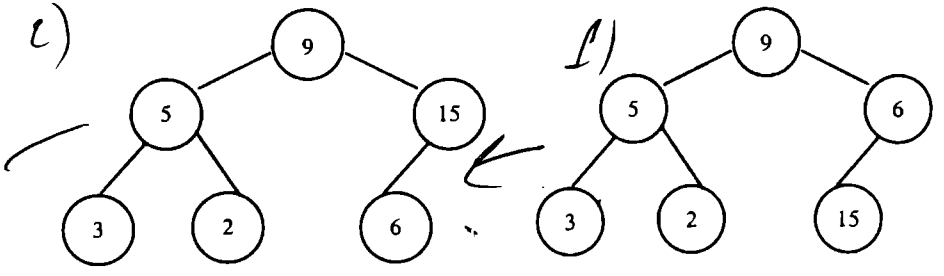
Օրինակ.





ավելացնենք 15-ը

Այժմ 15 տարրը տեղափոխենք իր ճիշտ տեղը.




```
// տարրի ավելացում
void Heap:: heapInsert (const HeapItemType & newItem)
{
    if (size > Max_Heap) cout << "կույտը լիքն է";
    else     {
        items [ size] = newItem ;
        int place = size ;
        int parent = (place - 1)/2 ;
        while(( parent >= 0 ) && (items[ place] >
             items[ parent] )) {
            HeapItemType temp = items[ parent] ;
            items[ parent] = items[ place] ;
            items[ place] = temp ;
            place = parent;
            parent = (place-1)/2;
        }
        size ++ ;
    }
}
```

Heap դասի նկարագրությունը հետևյալն է.

```
const int MaxHeap = 1000; // բույրի տարրերի
                          //  առավելագույն քանակը
typedef int HeapItemType;
class Heap {
public:
      Heap ();
      bool heapIsEmpty ();
```

```
        void heapInsert (const HeapItemType & newItem);
        void heapDelete (HeapItemType & rootItem);
private:
        HeapItemType items[ MaxHeap] ;
        int size;
        void heapRebuild(int root);
} ;
```

5.12 ԲՈՒՐԳԱՅԻՆ ԿԱՐԳԱՎՈՐՈՒՄ

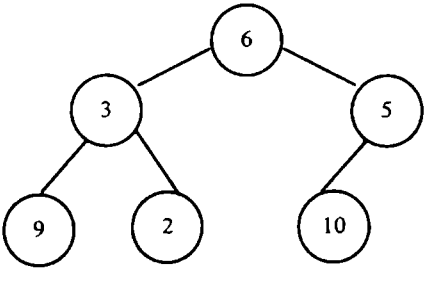
Այժմ, բուրգի միջոցով կարգավորենք տրված զանգվածի տարրերը (բուրգային կարգավորում):

Գործընթացն իրականացվում է երկու քայլով: Սկզբից զանգվածից կառուցվում է բուրգ, այնուհետև նրա միջոցով կատարվում է զանգվածի տարրերի կարգավորում:

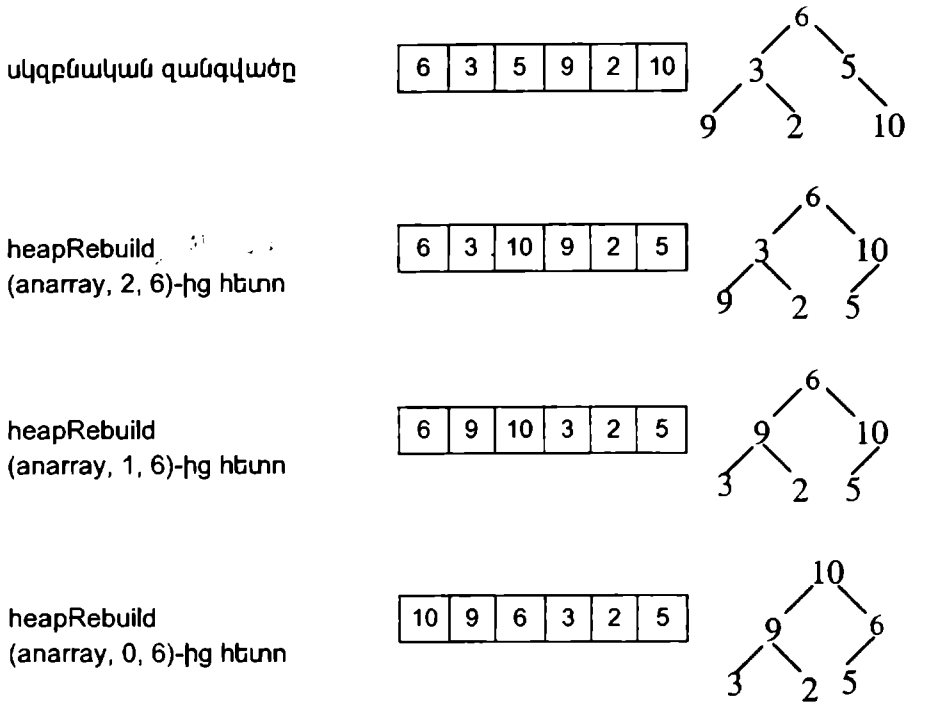
1. Զանգվածից բուրգի կառուցումը կարելի է կատարել երկու եղանակով.
   ա) զանգվածի տարրերը հերթով ավելացվում են բուրգի մեջ HeapInsert()-ի միջոցով,
   բ) զանգվածի տարրերը ներկայացվում են բինար ծառի տեսքով՝ ծառը կառուցելով վերևից ներքև և ձախից աջ:

anArray

6	3	5	9	2	10



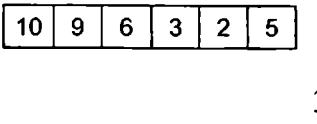
Այնուհետև ստացված ծառը վերածվում է բուրգի՝ յուրաքանչյուր տարրի վրա կիրառելով heapRebuild() ֆունկցիան (վերջից մինչև սկիզբ): heapRebuild() ֆունկցիայի ստորև օգտագործված տեսքում զանգվածի անունը, դիտարկվող տարրի ինդեքսը և զանգվածի չափը տրված են որպես պարամետրեր:

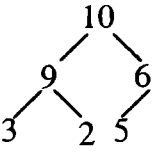


2. Զանգվածը բուրգի վերածելուց հետո, բուրգային կարգավորման ալգորիթմը զանգվածը բաժանում է երկու մասի, բուրգի տիրույթ՝ (anArray[0 .. last] - ձախ մաս) և կարգավորված տիրույթ՝ (anArray[last + 1 .. n-1] - աջ մաս): Սկզբնական վիճակում ամբողջ զանգվածը բուրգ է, այսինքն last-ի արժեքը հավասար է n-1: Ալգորիթմի յուրաքանչյուր քայլում բուրգի մեկ տարրը (արմատը) տեղափոխվում է զանգվածի վերջին տարրի հետ, որի հետևանքով զանգվածի կարգավորված մասի երկարությունը (աջ մասը) մեկով ավելանում է, իսկ զանգվածի բուրգ կազմող հատվածի երկարությունը (ձախ մասը) մեկով պակասում է: Իհարկե, նման տեղափոխությունից հետո ձախ մասը կարող է բուրգ չլինել, այդ պատճառով, ամեն անգամ տեղափոխություն կատարելուց հետո, ձախ մասի նկատմամբ պետք է կիրառել HeapRebuild ալգորիթմը:

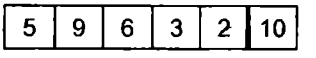
Այսպիսով, բուրգի մեծագույն տարրը հերթով տեղափոխվում է զանգվածի մեջ՝ աջից ձախ: Արդյունքում զանգվածը կարգավորվում է աճման կարգով:

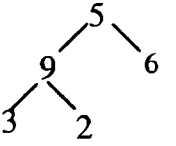
սկզբնական բուրգը



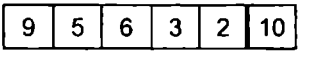


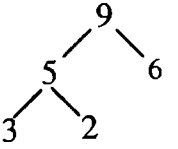
anArray[0] և anArray[last]-ի տեղափոխումից և last-ը 1-ով պակասեցնելուց հետո



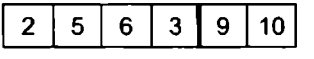


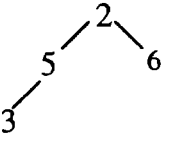
heapRebuild (anArray, 0,5)-ից հետո



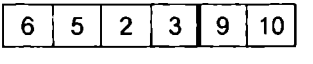


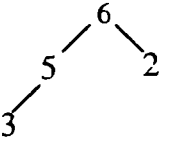
anArray[0] և anArray[last]-ի տեղափոխումից և last-ը 1-ով պակասեցնելուց հետո



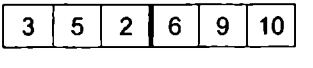


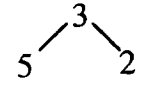
heapRebuild (anArray, 0,4)-ից հետո



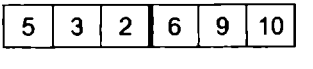


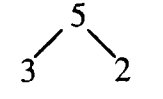
anArray[0] և anArray[last]-ի տեղափոխումից և last-ը 1-ով պակասեցնելուց հետո



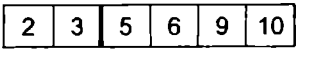


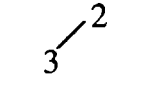
heapRebuild (anArray, 0,3) - ից հետո



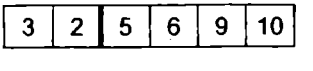


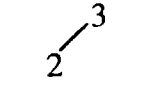
anArray[0] և anArray[last]-ի տեղափոխումից և last-ը 1-ով պակասեցնելուց հետո



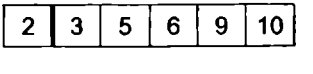


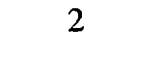
heapRebuild (anArray, 0,2) - ից հետո





anArray[0] և anArray[last]-ի տեղափոխումից և last-ը 1-ով պակասեցնելուց հետո



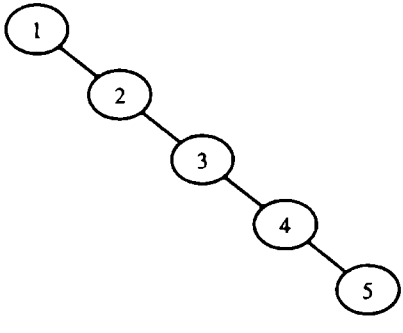


Ալգորիթմը նկարագրող ֆունկցիան հետևյալն է.

```
heapSort(arrayType *anArray, int n){
    // կառուցենք սկզբնական բուրգը
    for( int index = n-1; index > 0; index -- )
heapRebuild( anArray, index, n);
    last = n-1;
    // կարգավորենք
    for( int i = 1; i <= n; i ++ ) {
        anArrayType temp = anArray[ 0] ;
        anArray[ 0]  = anArray[ last] ;
        anArray[ last]  = temp;
        last --;
        heapRebuild(anArray, 0, last+1);
    }
}
```

5.13 B ԾԱՌԵՐ

Կառուցենք ամբողջ թվերով որոնման բինար ծառ, որի արմատը 1 է: Արմատին հաջորդաբար ավելացնենք 2, 3, 4, 5 թվերը:



Այս ձևով կառուցված ծառն ունի մեկ ճյուղի տեսք, և այստեղ փնտրում կատարելիս, հաճախ, անհրաժեշտ է լինում անցնել բոլոր հանգույցներով: Աշխատենք փոքրացնել տերևի խորությունը: Դրա համար օգտագործենք հատուկ տիպի ծառ՝ B-ծառ, որտեղ տերևները սկզբունքորեն չեն կարող շատ խորն իջնել մնացած հանգույցնե-

րի համեմատ: B-ծառերն առաջին անգամ առաջարկվեցին 1972 թ-ին Բայերի և Մակ-Կրեյտի կողմից: B-ծառերը չեն կարող համարվել որոնման բինար ծառեր, քանի որ B-ծառի հանգույցները կարող են ունենալ երկուսից ավելի որդիներ: Բացի դրանից հանգույցը կարող է ունենալ մի քանի տվյալներ: Յուրաքանչյուր B-ծառի համար տրվում է մի դրական ամբողջ հաստատուն (minimum), որը որոշում է մեկ հանգույցում գտնվող տարրերի թիվը:

Այժմ տանք B ծառերում տվյալների կազմակերպման մի քանի կանոններ.

Կանոն 1. B ծառի բոլոր հանգույցները պետք է ունենան minimum-ից ոչ պակաս տարրեր: Բացառություն է կազմում արմատը, որը կարող է ունենալ մինչև մեկ տարր (կամ ոչ մի տարր):

Կանոն 2. B-ծառի յուրաքանչյուր հանգույցի տարրերի առավելագույն քանակը չի կարող գերազանցել minimum հաստատունի կրկնապատիկին:

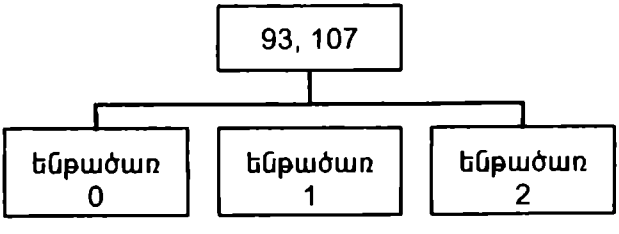
Կանոն 3. B ծառի հանգույցի տարրերը պահվում են զանգվածում և կարգավորված են ըստ աճման:

Կանոն 4. B ծառի տերև չհանդիսացող հանգույցի ենթածառերի քանակը միշտ մեկով մեծ է տվյալ հանգույցում գտնվող տարրերի քանակից:

Կանոն 5. B ծառի յուրաքանչյուր ոչ տերևային հանգույցի համար

- i-րդ տարրի արժեքը մեծ է տվյալ հանգույցի i-րդ ենթածառի ցանկացած տարրից:
- i-րդ տարրի արժեքը փոքր է տվյալ հանգույցի i+1 ենթածառի ցանկացած տարրից:

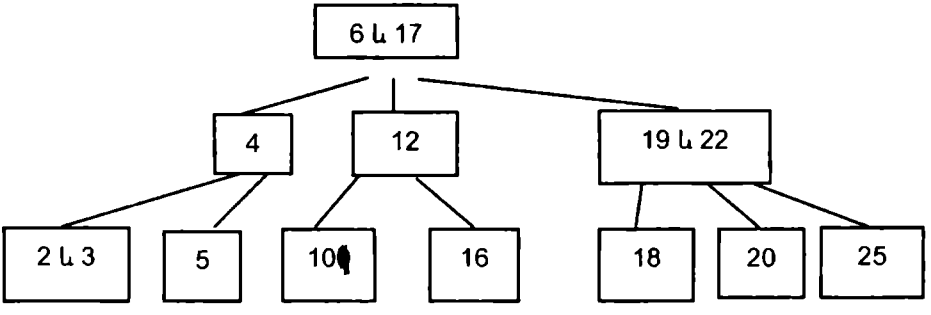
Ենթադրենք կա ոչ տերևային հանգույց, որը պարունակում է երկու ամբողջ թիվ՝ 93 և 107: Այս դեպքում հանգույցը պետք է ունենա երեք ենթածառ:



0 ենթածառի յուրաքանչյուր տարրի արժեքը փոքր է 93-ից, 1 ենթածառի յուրաքանչյուր տարրի արժեքը գտնվում է 93-ից 107 միջակայքում: 2 ենթածառի յուրաքանչյուր տարրի արժեքը մեծ է 107-ից:

Կանոն 6. B ծառի բոլոր տերևներն ունեն հավասար խորություն: B ծառը հավասարակշռված է:

Ստորև բերված B ծառում 10-ի փնտրումը սկսվում է արմատից, որտեղ պետք է գտնել այն ամենափոքր տարրը, որի արժեքը 10-ից փոքր չէ: Օրինակում դա 17-ն է: Քանի որ10-ը 6-ից 17 միջակայքից է, ապա փնտրումը շարունակվում է 12 արմատով ենթածառում: Քանի որ 10 փոքր է 12-ից, ապա փնտրումը շարունակվում է 10 արմատով ենթածառում (տերևում): Արժեքը գտնված է:



Ծրագրավորման ժամանակ B ծառերի օգտագործումը պայմանավորված է նրանով, որ B ծառը միշտ հավասարակշռված է (բոլոր տերևներն ունեն հավասար խորություն): Դա թույլատրում է ստեղծել արագ և արդյունավետ ալգորիթմներ: Եթե B ծառում minimum=1, ապա ծառի յուրաքանչյուր հանգույց պարունակում է 1 կամ 2 տարր, իսկ յուրաքանչյուր ոչ տերևային հանգույց ունի 2 կամ 3 ենթածառ: Այդպիսի ծառերը կոչվում են 2-3 ծառեր:

Մեծ քանակով տվյալների հետ աշխատելիս նրանք կարող են կազմակերպվել B ծառի միջոցով: Սովորաբար այդպիսի դեպքերում արմատային հանգույցի տվյալները գտնվում են օպերատիվ հիշողությունում, իսկ ոչ արմատային հանգույցները բեռնվում են արտաքին կրիչներից ըստ անհրաժեշտության:

ԽՆԴԻՐՆԵՐ

1. Տպել տրված բինար ծառի գագաթներում պարունակվող արժեքները, կատարելով ծառի սիմետրիկ շրջանցում (ռեկուրսիվ և ոչ ռեկուրսիվ եղանակներով):
2. Տպել տրված բինար ծառի գագաթներում պարունակվող արժեքները, կատարելով ծառի ուղիղ շրջանցում (ռեկուրսիվ և ոչ ռեկուրսիվ եղանակներով):
3. Տպել տրված բինար ծառի գագաթներում պարունակվող արժեքները, կատարելով ծառի հակադարձ շրջանցում (ռեկուրսիվ և ոչ ռեկուրսիվ եղանակներով):
4. Ճի՞շտ է արդյոք, որ բինար ծառը կարելի է միարժեքորեն վերականգնել ունենալով նրա կամայական երկու շրջանցումների արդյունքները:
5. Վերականգնել բինար ծառը ըստ ուղիղ և սիմետրիկ շրջանցումների:
6. Ստուգել, արդյո՞ք տրված բինար ծառի սիմետրիկ շրջանցման արդյունքում ծառի գագաթներում պարունակվող տվյալները դասավորվում են աճման կարգով:
7. Պատճենել տրված բինար ծառը:
8. *Գտնել տրված բինար ծառի գագաթների քանակը:
9. Գտնել տրված բինար ծառի տերևների քանակը:
10. *Գտնել տրված բինար ծառի բարձրությունը:
11. Գտնել տրված բինար ծառում X տարրի հանդիպումների քանակը:
12. *Բինար ծառը կարելի է ներկայացնել զանգվածի միջոցով հետևյալ կերպ: Զանգվածի յուրաքանչյուր տարրը երկու դաշտից բաղկացած կառուցվածք է և համապատասխանում է ծառի գագաթին: Առաջին դաշտում դրվում է գագաթում պարունակվող արժեքը (եթե ծառում այդ գագաթը չկա, ապա դրվում է ինչ-որ ֆիկտիվ արժեք): Երկրորդ դաշտում դրվում է մեկ, եթե ծառն ըստ մակարդակների ձախից աջ շրջանցելիս հերթական գագաթն առկա է, և զրո հակառակ դեպքում: Ըստ այս ներկայացման կառուցել ծառը որպես ոչ գծային կապակցված ցուցակ:
13. Բինար ծառը ներկայացված է զանգվածի միջոցով: Զանգվածի յուրաքանչյուր տարր երեք դաշտից բաղկացած կառուցվածք է և համապատասխանում է ծառի գագաթին: Առաջին դաշտում դրվում է հերթական գագաթում պարունակվող արժեքը, երկրորդ դաշտում նշվում է այդ գագաթի ձախ որդուն ներկայացնող զանգվածի տարրի ինդեքսը, երրորդ դաշտում՝ աջ որդուն ներկայացնող զանգվածի տարրի ինդեքսը: Ըստ այս ներկայացման կառուցել ծառը որպես ոչ գծային կապակցված ցուցակ:

14. *Տալել տրված ծառի յուրաքանչյուր մակարդակում պարունակվող հանգույցների քանակը:

15. Տալել տրված ծառի գագաթներում պարունակվող արժեքները ըստ մակարդակների՝ վերևից ներքև, ձախից աջ շրջանցումով:

16. Գտնել տրված բինար ծառի այն մակարդակների համարները, որոնք պարունակում են ամենաշատ թվով գագաթներ:

17. Որոշել տրված բինար ծառի լայնությունը: Ծառի լայնությունը՝ դա այն մակարդակի գագաթների քանակն է, որում այդ քանակն ընդունում է մեծագույն արժեք:

18. Գտնել տրված բինար ծառի այն մակարդակների համարները, որոնցում գագաթների քանակը նվազագույնն է (բացառությամբ զրոյական մակարդակի):

19. Գտնել տրված բինար ծառի մեկ որդի ունեցող հանգույցների քանակը:

20. *Գտնել տրված բինար ծառի երկու որդի ունեցող հանգույցների քանակը:

21. Իրական թվեր պարունակող բինար ծառի համար գտնել այդ արժեքներից փոքրագույնը:

22. Իրական թվեր պարունակող բինար ծառի համար գտնել այդ արժեքների միջին թվաբանականը:

23. Իրական թվեր պարունակող բինար ծառի համար գտնել զրոյական արժեքով հանգույցների քանակը:

24. Գտնել բինար ծառի այն մակարդակների համարները, որոնք պարունակում են զրոյական արժեքով հանգույցներ:

25. Գտնել իրական թվեր պարունակող բինար ծառի այն մակարդակների համարները, որոնց հանգույցները պարունակում են ծառում առկա փոքրագույն արժեքը:

26. Բինար ծառը կոչվում է խիստ բինար, եթե կամայական ոչ տերև հանդիսացող գագաթ ունի ճիշտ երկու որդի: Ստուգել ծառը խիստ բինա՞ր է, թե՞ ոչ:

27. Ստուգել տրված բինար ծառը լրի՞վ է, թե՞ ոչ:

28. Ստուգել տրված բինար ծառը իդեալապես բալանսավորվա՞ծ է, թե՞ ոչ:

29. Տրված զանգվածի տարրերից կառուցել իդեալապես բալանսավորված ծառ:

30. Ստուգել տրված բինար ծառը կատարյա՞լ է, թե՞ ոչ:

31. Կարված է կոչվում այն բինար ծառը, որի հանգույցների զրոյական կապերը փոխարինված են ոչ զրոյական կապերով (թել-կապերով): Եթե տվյալ հանգույցի ձախ կապը զրոյական է, ապա կարված ծառում այն փոխարինվում է ոչ զրոյական թել-կապով՝ ուղղված այն հանգույցին, ո-

րը սկզբնական ծառի սիմետրիկ շրջանցման արդյունքում հանդիպում է տվյալ հանգույցից անմիջապես առաջ: Եթե զրոյական է տվյալ հանգույցի աջ կապը, ապա կարված ծառում այն փոխարինվում է թել-կապով այն հանգույցի վրա, որը սկզբնական ծառի սիմետրիկ շրջանցման արդյունքում հանդիպում է տվյալ հանգույցից անմիջապես հետո: Բացառություն են կազմում ծառի ամենածախ և ամենաաջ հանգույցները: Այս հանգույցների համար պահպանվում են համապատասխանաբար ձախ և աջ զրոյական կապերը: Կառուցել կարված ծառ ըստ տրված ծառի:

32. Իրականացնել կարված ծառի սիմետրիկ շրջանցում:
33. Թվաբանական արտահայտությունը ներկայացվում է բինար ծառի միջոցով հետևյալ կերպ. ծառի տերևները պարունակում են արտահայտության օպերանդները, իսկ ոչ տերև հանդիսացող հանգույցները՝ գործողությունների նշանները: Գործողության նշան պարունակող հանգույցի ձախ և աջ ենթածառերը համապատասխանաբար տվյալ գործողության ձախ և աջ օպերանդները ներկայացնող ծառ-արտահայտություններն են: Ըստ փոփոխականներ չպարունակող արտահայտության կառուցել այն ներկայացնող բինար ծառը և հաշվել արտահայտության արժեքը:
34. Տպել ծառ-արտահայտությունը բանաձևի տեսքով: Ստուգել, արդյո՞ք տրված բինար ծառը ներկայացնում է արտահայտություն, թե՞ ոչ:
35. Պարզեցնել ծառ-արտահայտությունը հաշվի առնելով հետևյալ առնչությունները՝ f+0=f, 0+f=f, f-0=0, f*1=1, 1*f=f:
36. Ձևափոխել ծառ-արտահայտությունը հաշվի առնելով հետևյալ առնչությունները՝ (f1+f2)*f3=(f1*f3)+(f2*f3), f3*(f1+f2)=(f3*f1)+(f3*f2):
37. Կասենք, որ երկու բինար ծառեր նման են, եթե դրանք համընկնում են ըստ իրենց կառուցվածքի: Որոշել տրված երկու ծառերը նմա՞ն են, թե՞ ոչ:
38. Որոշել տրված երկու ծառերը նույնն են, թե՞ ոչ, այսինքն նման են և համապատասխան հանգույցներում պարունակում են միևնույն արժեքները:
39. Բինար ծառի յուրաքանչյուր հանգույցում գրանցել նրա մակարդակի համարը:
40. Գտնել տրված բինար ծառի այն մակարդակների համարները, որոնցում կամայական գագաթ ունի ճիշտ երկու որդի:
41. Գտնել իրական թվեր պարունակող տրված բինար ծառի այն մակարդակների համարները, որոնցում հանգույցների արժեքները ձախից աջ դասավորված են չնվազման կարգով:
42. Բինար ծառի յուրաքանչյուր գագաթում գրանցել իրենով որոշվող ենթածառի գագաթների քանակը:

43. Բինար ծառի յուրաքանչյուր գագաթում գրանցել իրենով որոշվող ենթածառի տերևների քանակը:
44. Բինար ծառի յուրաքանչյուր գագաթում գրանցել իրենով որոշվող ենթածառի ոչ տերև հանդիսացող գագաթների քանակը:
45. Բինար ծառի յուրաքանչյուր գագաթում գրանցել իրենով որոշվող ենթածառի բարձրությունը:
46. Բինար ծառի յուրաքանչյուր գագաթում գրանցել մեկ, եթե իրենով որոշվող ենթածառում առկա է ծառի փոքրագույն տարրը, և զրո՝ հակառակ դեպքում:
47. Որոշել տրված բինար ծառի մեծագույն տարրը պարունակող առաջին մակարդակի հանգույցների քանակը:
48. Բինար ծառի յուրաքանչյուր գագաթում գրանցել, թե որերորդն է նա իր մակարդակում ձախից աջ դիտարկման ժամանակ:
49. Բինար ծառի յուրաքանչյուր գագաթում գրանցել, թե լայնակի շրջանցելիս քանի բաց թողած գագաթ կա մինչ այդ գագաթը ծառի արմատից սկսած:
50. Բինար ծառի յուրաքանչյուր գագաթում գրանցել, թե քանի բաց թողած գագաթ կա իր մակարդակում մինչ այդ գագաթը:
51. Որոշել ամբողջ թվեր պարունակող բինար ծառի մինչև տրված i-րդ մակարդակը եղած տարրերի գումարը:
52. Որոշել բինար ծառի տրված i-րդ մակարդակից հետո եղած հանգույցների քանակը:
53. Իրական թվեր պարունակող բինար ծառի յուրաքանչյուր գագաթում գրանցել իրենով որոշվող ենթածառի տարրերի միջին թվաբանականը:
54. Իրական թվեր պարունակող բինար ծառի տրված i-րդ մակարդակի յուրաքանչյուր գագաթում գրանցել իրենով որոշվող ենթածառի տարրերի միջին թվաբանականը՝ հեռացնելով i-րդ մակարդակից հետո եկող բոլոր գագաթները:
55. Բինար ծառից հեռացնել տրված գագաթը՝ իրենով որոշվող ենթածառով հանդերձ:
56. Բինար ծառից հեռացնել նրա բոլոր տերևները:
57. Բինար ծառից փոքրագույն քանակությամբ գագաթներ հեռացնելով, այն դարձնել լրիվ ծառ:
58. Բինար ծառից հեռացնել նրա բոլոր հանգույցները:
59. Տպել բինար ծառի արմատից մինչև տերևներ տանող բոլոր ճանապարհների գագաթներում պարունակվող արժեքները՝ ըստ ճանապարհների:

60. Տպել բինար ծառի արմատից մինչև տերև տանող ամենաաջ (ամենաձախ) և ամենաերկար ճանապարհների գագաթներում պարունակվող արժեքները:
61. Տպել բինար ծառի արմատից մինչև տերևներ տանող բոլոր ամենաերկար ճանապարհները:
62. Տպել բինար ծառի արմատից մինչև տերև տանող ամենակարճ ճանապարհներից մեկի գագաթներում պարունակվող արժեքները:
63. Կառուցել որոնման բինար ծառ ամբողջ թվեր պարունակող զանգվածի տարրերից: Տվյալների բանալին համընկնում է զանգվածի տարրի հետ:
64. Տպել որոնման բինար ծառի այն բոլոր գագաթների արժեքները, որոնք փոքր են տրված արժեքից:
65. Որոնման բինար ծառից հեռացնել i–րդ մակարդակի բոլոր գագաթները պահպանելով բինար ծառ լինելու հատկությունը:
66. *Որոնման բինար ծառը կարող է ունենալ կրկնվող արժեքներով գագաթներ: Այդ դեպքում ծառի յուրաքանչյուր գագաթում կարելի է պահել նաև տվյալ արժեքի կրկնումների քանակը՝ չավելացնելով նոր գագաթներ: Ստեղնաշարից կարդալ n հատ թիվ (թվերը կարող են կրկնվել), ստեղծել որոնման բինար ծառ, որից հետո ծառը տպել 90 աստիճանով շրջված:
67. Կառուցել որոնման բինար ծառն իրականացնող BinSearchTree դասը: Ծառի հանգույցները բաղկացած են չորս դաշտերից: Առաջին դաշտը պարունակում է տվյալների բանալին, երկրորդը՝ տվյալները, երրորդը և չորրորդը՝ ցուցիչներ ձախ և աջ որդիների վրա համապատասխանաբար:
68. Տպել որոնման բինար ծառի հանգույցներում պարունակվող տվյալները բանալու չնվազման կարգով:
69. Իրականացնել որոնման բինար ծառի RNL-շրջանցումը:
70. Տպել որոնման բինար ծառի հանգույցներում պարունակվող տվյալները բանալու չաճման կարգով:
71. Իրականացնել տրված հաջորդականության կարգավորումը որոնման բինար ծառի միջոցով:
72. Գտնել տրված որոնման բինար ծառի գագաթներում պարունակվող տվյալների բանալիներից փոքրագույնը:
73. √ Ստուգել արդյո՞ք տրված բինար ծառը որոնման ծառ է, թե՞ ոչ:
74. Ճի՞շտ է արդյոք, որ եթե տվյալ բինար ծառի սիմետրիկ շրջանցման արդյունքում ծառի գագաթներում պարունակվող տվյալների բանալիները դասավորվում են աճման կարգով, ապա այդ ծառը որոնման բինար ծառ է:

75. Կառուցել մաքսիմալ բուրգն իրականացնող Heap դասը:
76. Օգտվելով Heap դասից կառուցել մաքսիմալ բուրգ տրված զանգվածից:
77. Իրականացնել զանգվածի տարրերի կարգավորում բուրգի միջոցով:
78. Տրված կատարյալ բինար ծառը դարձնել բուրգ:
79. Տրված մաքսիմալ բուրգից կառուցել նույն տարրերը պարունակող մինիմալ բուրգ:
80. Որոշել արդյո՞ք տրված բինար ծառը AVL ծառ է:
81. Իրականացնել տարրի ավելացում AVL ծառում:
82. Իրականացնել տարրի հեռացում AVL ծառից:
83. *Կառուցել Phonebook դասը՝ հեռախոսահամարների գրքույկ մոդելավորելու համար: Գրքույկը կազմված է Person դասի օբյեկտներից, որոնք պարունակում են ազգանուն, անուն, հեռախոսահամար: Գրքույկը ներկայացնել որոնման բինար ծառի տեսքով՝ կարգավորված ըստ ազգանունների: Նախատեսել ըստ ազգանվան հեռախոսահամարի որոնման, ավելացման և հեռացման գործողությունները: Ենթադրել, որ գրքույկը պահվում է տեքստային ֆայլում: Օգտագործելուց առաջ այն պետք է բեռնվի հիշողություն, աշխատանքի ավարտից հետո՝ կրկին գրվի ֆայլում:
84. Տեքստային ֆայլում գրված է առանց քերականական սխալների ծրագիր C++ լեզվով: Այբբենական կարգով տպել այդ ծրագրում հանդիպող բոլոր իդենտիֆիկատորները և նրանց հանդիպումների քանակը: Իդենտիֆիկատորների բազմության հետ աշխատելու համար այն պահել որոնման բինար ծառի տեսքով:
85. Իրականացնել տարրի ավելացում և հեռացում B-ծառերում:

ԳԼՈՒԽ 6.

ԱՂՅՈՒՍԱԿՆԵՐ

6.1. ԱՂՅՈՒՍԱԿ ՏՎՅԱԼՆԵՐԻ ԱԲՍՏՐԱԿՏ ՏԻՊԸ

Աղյուսակ (բառարան, dictionary) տվյալների աբստրակտ տիպը հիմնված է տարրերի բազմության մոդելի վրա: Ինչպես հայտնի է, բազմություն տվյալների աբստրակտ տիպի համար ընդհանուր դեպքում սահմանված են միավորման, հատման, տարբերության, համեմատման, ինչպես նաև տարրի պատկանելիության ստուգման, տարրի ավելացման և հեռացման գործողությունները:

Բազմություն տվյալների աբստրակտ տիպը, որի համար սահմանված են տարրի որոնման, ավելացման և հեռացման գործողությունները, կոչվում է աղյուսակ (բառարան):

Աղյուսակի տարրերը իրենցից ներկայացնում են մի քանի դաշտերից կազմված գրառումներ: Դաշտերից մեկը համարվում է բանալիային դաշտ, ըստ որի աղյուսակում կատարվում է գրառման որոնում: Պարզության համար համարում ենք, որ աղյուսակում գրառումներն ունեն տարբեր բանալիներ:

Նկարագրենք Table տվյալների աբստրակտ տիպը:

ADT Table

Տվյալներ

Տարրերի բազմություն

Գործողություններ

Կոնստրուկտոր

նախապայման - չկա

պրոցես - աղյուսակի սկզբնարժեքավորում

Empty

մուտք - չկա

նախապայման - չկա

պրոցես - ստուգում, դատարկ է արդյոք աղյուսակը

ելք - true, եթե աղյուսակը դատարկ է, false՝ հակառակ դեպքում

վերջնապայման - չկա

Insert
մուտք – աղյուսակին ավելացվող տարրը
նախապայման – չկա
պրոցես – տարրի ավելացում աղյուսակին
ելք – չկա
վերջնապայման – ավելացված տարրով աղյուսակ
բացառիկ իրավիճակ – նոր տարրի ավելացում լցված աղյուսակին
Delete
մուտք - չկա
նախապայման – չկա
պրոցես – տարրի հեռացում աղյուսակից
ելք – չկա
վերջնապայման – հեռացված տարրով աղյուսակ
բացառիկ իրավիճակ – տարրի հեռացում դատարկ աղյուսակից
Find
մուտք – որոնվող տարր
նախապայման – աղյուսակը դատարկ չէ
պրոցես – տարրի որոնում աղյուսակում
ելք - true, եթե աղյուսակը պարունակում է որոնվող տարրը, false՝ հակառակ դեպքում
վերջնապայման – չկա
ClearTable
մուտք - չկա
նախապայման - չկա
պրոցես - աղյուսակի բոլոր տարրերի հեռացում
ելք – չկա
վերջնապայման – դատարկ աղյուսակ
վերջ ADT Table

6.2. ԱՂՅՈՒՍԱԿԻ ԻՐԱԿԱՆԱՑՈՒՄ

Աղյուսակի իրականացման համար կարելի է օգտագործել տվյալների գծային և ոչ գծային կառուցվածքներ: Գծային կառուցվածքներից նշենք

- բուլյան վեկտորը,
- չկարգավորված զանգվածը,

- չկարգավորված կապակցված գծային ցուցակը,
- ըստ գրառման բանալու կարգավորված զանգվածը,
- ըստ գրառման բանալու կարգավորված կապակցված գծային ցուցակը:

Որպես ոչ գծային կառուցվածքներ կարող ենք ընտրել ըստ գրառման բանալու կարգավորված որոնման հավասարակշռված բինար ծառերը՝ սև-կարմիր, AVL-ծառեր և այլն, որոնք ապահովում են աղյուսակի հետ կատարվող գործողությունների արդյունավետությունը:

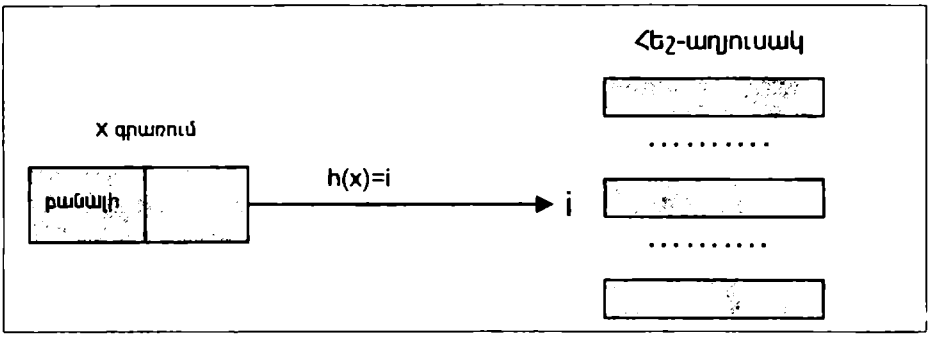
Նկատենք, որ աղյուսակի իրականացումը վերը նշված կառուցվածքների միջոցով ապահովում է գործողությունների կատարման տարբեր ժամանակային բարդություններ. O(n)՝ զանգվածի և ցուցակի, O(log n)՝ հավասարակշռված բինար ծառի, O(1)՝ բուլյան վեկտորի դեպքում: Ակնհայտ է, որ բուլյան վեկտորի միջոցով աղյուսակի իրականացումն առավել արդյունավետն է, սակայն այս դեպքում սահմանափակված ենք միայն այն բազմությունների դիտարկմամբ, որոնք որևէ վերջավոր բազմության ենթաբազմություններ են: Գործողությունների կատարումը ֆիքսած ժամանակում կարելի է ապահովել՝ ընտրելով աղյուսակի իրականացման եղանակ, որը հիմնված է հեշավորման մեթոդի վրա:

Աղյուսակի իրականացում հեշ-աղյուսակի միջոցով

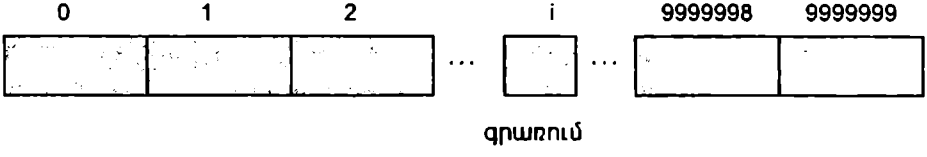
Այս մեթոդի էությունն այն է, որ մեծ քանակով տարրեր պարունակող բազմության հետ կատարվող աշխատանքը փոխարինվում է ավելի քիչ քանակով տարրեր պարունակող բազմությունների հետ աշխատանքով: Օրինակ, ազգանուններ պարունակող ծոցատետրը բաժանվում է էջերի ըստ ազգանվան առաջին տառի, տրոհելով ազգանունների բազմությունը ենթաբազմությունների: Սա ապահովում է տրված ազգանվան որոնումը ոչ թե ամբողջ բազմության մեջ, այլ ընդամենը մեկ էջում:

Դիցուք M բազմությունը ներկայացնող աղյուսակի գրառումների պահպանման համար օգտագործվում է N չափանի զանգված և H(x)-ը ֆունկցիա է, որը M բազմության յուրաքանչյուր գրառմանը համապատասխանեցնում է ոչ բացասական ամբողջ թիվ, օրինակ գրառման բանալի: Որպես հեշ-ֆունկցիա կարելի է օգտագործել h(x)= H(x)%N ֆունկցիան, որը M բազմությունը տրոհում է N չհատ-

վող և 0,1,..., N-1 թվերով համարակալված M_i (0<=i<=N-1) ենթաբազմությունների: M_i բազմությունը պարունակում է M բազմության այն տարրերը, որոնց համար h(x)=i: M_i բազմությանը համապատասխանեցնենք աղյուսակի i-րդ տարրը:



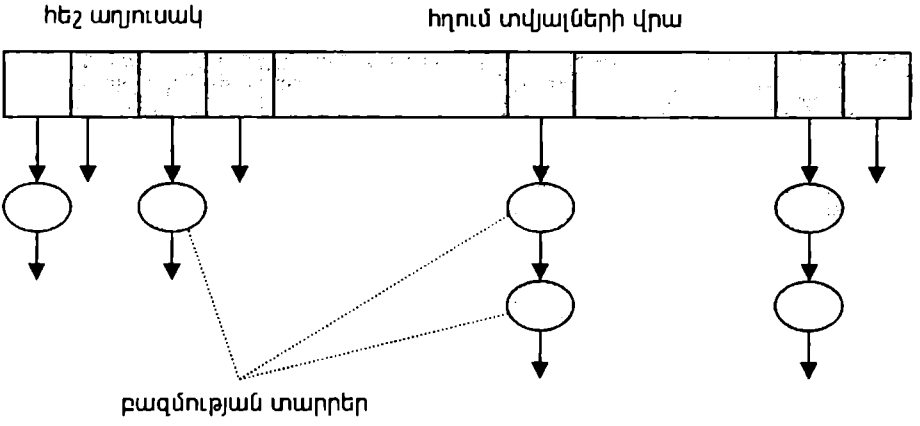
Այսպիսով հեշ-ֆունկցիան իրականացնում է մի մեխանիզմ (հասցեի որոշման մեխանիզմ), որն ըստ աղյուսակի գրառման բանալու որոշում է տվյալ գրառմանը համապատասխանող տարրը զանգվածում: Այս դեպքում ցանկացած նոր գրառման ավելացումն ու հեռացումը կատարվում է հաստատուն ժամանակում: N չափանի զանգվածը կոչվում է հեշ-աղյուսակ (hash table): Հեշ-աղյուսակի i–րդ տարրը պարունակում է տվյալներ (նկ.6.1) կամ հղում տվյալների վրա (նկ.6.2): Հեշ-ֆունկցիայի միևնույն արժեքով տվյալների խումբն անվանում են սեգմենտ:



Նկ.6.1. Փակ հեշավորում

Դիցուք աղյուսակի տվյալներն իրենցից ներկայացնում են ինֆորմացիա քաղաքի տվյալ շրջանի բնակիչների մասին և որպես բանալի ընտրված է բնակչի վեցանիշ հեռախոսահամարը, որի առաջին երկու թվանշանները կապված են տվյալ շրջանի հետ: Այս դեպքում որպես հեշ-աղյուսակ կարելի է դիտարկել 10000 տարր պարունակող զանգվածը (տարրերը համարակալված են 0 մինչև 9999) h(x)=x%10000 հեշ-ֆունկցիայով, որտեղ x–ը գրառման բանալին է՝ վեցանիշ հեռա-

խոսահամարը: h(x) ֆունկցիան իրականացնում է բանալիների բազմության փոխմիարժեք արտապատկերում [0..9999] բազմության վրա: Այստեղ հեշ-աղյուսակի h(x)-րդ տարրը կարելի է օգտագործել համապատասխան գրառումը պահելու համար:



Նկ.6.2. Բաց հեշավորում

Եթե որպես հեշ-աղյուսակ դիտարկենք 100 տարր պարունակող զանգվածը h(x)=x%100 հեշ-ֆունկցիայով, ապա h(x)-րդ տարրը համապատասխանում է 100 քառանիշ հեռախոսահամարներից կազմված սեգմենտի, որի մեջ ընդգրկված գրառումների բանալիների վերջին երկու նիշերը համընկնում են: Այս դեպքում հեշ-ֆունկցիան իրականացնում է "շատերը մեկին" տիպի արտապատկերում և, հետևաբար, երկու և ավելի գրառումների բանալիներ կարող են հեշավորվել միևնույն, սակայն հեշ-աղյուսակում չեն կարող զբաղեցնել միևնույն տեղը: Այս իրավիճակը կոչվում է բախում (collision): Նշենք, որ ոչ միշտ է հնարավոր ընտրել այնպիսի հեշ-ֆունկցիա, որն ապահովի գրառումների համաչափ բաշխում ըստ սեգմենտների: Ընդհանուր դեպքում հեշավորման ֆունկցիան պետք է լինի արագ և հեշտ հաշվարկելի, հավասարաչափ բաշխի գրառումները հեշ-աղյուսակում՝ ընդունելով տարբեր արժեքներ մոտավորապես նույն հավանականությամբ, սահմանափակի բախումների քանակը: Ցանկալի է, որ արգումենտի մոտ արժեքների համար հեշ-ֆունկցիան ընդունի հեռու արժեքներ: Բացի դրանից, տվյալների բաշխումը հեշ-աղյուսակում կլինի առավել համաչափ, եթե N-ը պարզ թիվ է:

Նկարագրենք հեշավորման հետևյալ մեթոդները.

- Բաժանման մեթոդ
- Քառակուսու կենտրոնի մեթոդ
- Բազմապատկման մեթոդ
- Փաթեթավորման մեթոդ

Բաժանման մեթոդ (division method): Այս մեթոդը ենթադրում է, որ գրառման բանալին որևէ եղանակով արտապատկերվում է ամբողջ թվի, որն այնուհետև մնացորդի ստացման գործողության օգնությամբ արտապատկերվում է 0..N-1 միջակայքի վրա, որտեղ N-ը հեշ-աղյուսակի չափն է: Օրինակ, եթե բանալին հնգանիշ թիվ է և N=100, ապա HF հեշ-ֆունկցիան վերադարձնում է բանալու վերջին երկու թվանշանները:

```
int HF(int key) {
    return key % 100; // հարյուրի վրա բաժանման մեթոդ
}
```

Եթե գրառման բանալին սիմվոլային տող է C++-ում, ապա HF հեշ-ֆունկցիան նախ որոշում է այդ տողի առաջին և վերջին սիմվոլների ASCII-կոդերի գումարը, այնուհետև հաշվում է այդ գումարի 101-ի վրա բաժանման մնացորդը և վերադարձնում այն որպես գրառման դիրքի արժեքը աղյուսակում (N=101): Այս ֆունկցիան հանգեցնում է բախման՝ տողերի առաջին և վերջին սիմվոլների համընկման դեպքում: Այսպես օրինակ, "start" և "slant" տողերը կարտապատկերվեն միևնույն արժեքի՝ HF("start")=HF("start")=29:

```
// հեշ ֆունկցիա սիմվոլային տողի համար
// վերադարձնում է արժեք 0-ից 100 միջակայքում
int HF(char *key) {
    int len = strlen(key), hashf = 0;
    // եթե բանալու արժեքը հավասար է 0 կամ 1
    // վերադարձնել key[0].
    // հակառակ դեպքում գումարել առաջին և վերջին
    // սիմվոլները
    if (len <= 1)
        hashf = key[0];
    else
        hashf = key[0] + key[len-1];
    return hashf % 101;
}
```

Նմանատիպ արդյունք է ստացվում այն հեշ-ֆունկցիայի դեպքում, որը հաշվում է տողի բոլոր սիմվոլների կոդերի գումարը: Այս դեպքում "bad" և "dab" տողերը կարտապատկերվեն միևնույն արժեքի:

```
int HF(char *key) {
    int hashf = 0;
    // գումարել տողի բոլոր սիմվոլները և բաժանել 101-ի
    while (*key)
        hashf += *key++;
    return hashf % 101;
}
```

Քառակուսու կենտրոնի մեթոդ (midsquare technique): Այս մեթոդում հեշ-ֆունկցիան գրառման բանալին արտապատկերում է ամբողջ թվի, հաշվում է այդ թվի քառակուսին և որպես արժեք վերադարձնում է այդ թվի մեջտեղից ընտրված թվանշաններով կազմված ամբողջ դրական թիվը: Օրինակ, եթե գրառման բանալին 32-բիթանի թիվ է, ապա HF հեշ-ֆունկցիայի արժեքը հավասար է բանալու քառակուսու 11-20-րդ բիթերով կազմած թվին:

```
// վերադարձնել key*key արտադրյալի միջին 10 բիթերը
int HF(int key) {
    key *= key;    // բանալին քառակուսի բարձրացնել
    key >>= 11;    // դեն նետել 11 ցածր կարգի բիթերը
    // վերադարձնել 10 ցածր կարգի բիթերը
    return key % 1024
}
```

Բազմապատկման մեթոդ (multiplicative method): Այս մեթոդն օգտագործում է պատահական թիվ [0,1] միջակայքից: Այդ թվի և գրառման բանալու արտադրյալի կոտորակային մասը նույնպես ընկած է [0,1] միջակայքում: Այն բազմապատկվում է աղյուսակի չափով և ստացված արտադրյալի ամբողջ մասը վերադարձվում է որպես հեշ-ֆունկցիայի արժեք:

```
// բազմապատկման մեթոդ օգտագործող հեշ ֆունկցիա
// վերադարձնում է արժեք 0..700 միջակայքում
int HF(int key){
```

```
    static RandomNumber rnd;
    float f;
    //բանալին բազմապատկել 0..1 միջակայքի պատահական թվով
    f = key * rnd.fRandom();
    f = f - int(f);      // վերցնել կոտորակային մասը
    // վերադարձնել թիվ 0..n-1 միջակայքից
    return 701*f;
}
```

Փաթեթավորման մեթոդ (rolling up method): Այս մեթոդը ենթադրում է, որ գրառման բանալին դրական ամբողջ թիվ է: Հեշ-ֆունկցիան հաշվում է բանալու բոլոր թվանշանների գումարը, որն էլ համարվում է գրառման դիրքը հեշ-աղյուսակում: Օրինակ, եթե բանալին ներկայացվում է իննանիշ դրական ամբողջ թվով, ապա 0<=h(x)<=81 և, հետևաբար հեշ-աղյուսակը պետք է պարունակի առնվազն 82 տարր: Կարելի է նաև որոշակի ձևով խմբավորել բանալու թվանշանները այնուհետև կիրառել փաթեթավորման մեթոդը: Օրինակ, եթե x = 102365821, ապա խմբավորելով x-ի թվանշանները եռյակներով և հաշվելով եռյակներով կազմված թվերի գումարը, կստանանք 102+365+821=1288: Խմբավորելով այս թվի թվանշանները զույգերով կստանանք 12+88=100 և կիրառելով փաթեթավորման մեթոդը կստանանք h(x) = 1+0+0=1:

Բախումների վերացում

Ինչպես տեսանք, մեկից ավելի բանալիներ կարող են հեշավորվել միանման, այսինքն արտապատկերվել միևնույն արժեքի: Սակայն, այդ բանալիներով գրառումները չեն կարող աղյուսակում զբաղեցնել միևնույն դիրքը: Առաջանում է բախում:

Հայտնի է բախումների վերացման երկու եղանակ.

- հեշ-ֆունկցիայի միևնույն արժեք ունեցող բանալիներով գրառումների (սեգմենտի) համար հեշ-աղյուսակում գտնել նոր տեղ (գծային հատարկմամբ բաց հասցեավորման մեթոդ),
- հեշ-ֆունկցիայի յուրաքանչյուր արժեքի համար ստեղծել առանձին ցուցակ, որում կընդգրկվեն այն գրառումները, որոնց բանալիները արտապատկերվել են այդ արժեքին (շղթաների մեթոդ):

Բաց հասցեավորում գծային հատարկմամբ: Այս մեթոդը ենթադրում է, որ հեշ-աղյուսակի յուրաքանչյուր տարր կարող է լինել ազատ կամ զբաղված: Եթե x բանալիով գրառումը պետք է ավելացնել աղյուսակին, ապա հաշվարկվում է h(x) հեշ-ֆունկցիայի արժեքը, որը մատնանշում է ավելացվող գրառման դիրքը հեշ-աղյուսակում: Եթե աղյուսակի h(x) համարի տարրը զբաղված է, ապա կատարվում է կրկնակի հեշավորում $h_1(x)$, $h_2(x)$, $h_3(x)$,... ֆունկցիաների միջոցով, որոնցով որոշվում է աղյուսակի այն ազատ դիրքը, որտեղ կարելի է տեղադրել x բանալիով գրառումը: Եթե աղյուսակում ազատ տեղ չկա, ապա գրանցվում է աղյուսակի գերհագեցման իրավիճակ: Այս սխեման կոչվում է բաց հասցեավորում:

Բաց հասցեավորման սխեմաները միմյանցից տարբերվում են աղյուսակի ազատ տարրի փնտրման եղանակով, այսինքն $h_i(x)$ (i=1,2,3,...) ֆունկցիաներով: Գծային հատարկման մեթոդում $h_i(x)$= (h(x)+i)%N (i=1,2,3,...), որտեղ N-ը հեշ-աղյուսակի չափն է:

Օրինակ, ենթադրենք տվյալները DataRecord տիպի են և պահվում են N=11 տարրերի համար նախատեսված աղյուսակում:

```
struct DataRecord {
    int key;
    int data;
}
```

Դիտարկենք h(item)=item.key%11 հեշ-ֆունկցիան և DataRecord տիպի հետևյալ տվյալները {54,1}, {77,3}, {94,5}, {89,7}, {14,8}, {45,2}, {76,9}: Նկատենք, որ h({54,1})=10, h({77,3})=0, h({94,5})=6, h({89,7})=1, h({14,8})=3, h({45,2})=1, h({76,9})=10:

Առաջին հինգ գրառումների բանալիների հեշավորումը տալիս է հինգ տարբեր արժեքներ, ըստ որոնց գրառումները տեղադրվում են Table աղյուսակում:

Table[0]	{77,3}	Table[6]	{94,5}
Table[1]	{89,7}	Table[7]	
Table[2]	{45,2}	Table[8]	
Table[3]	{14,8}	Table[9]	
Table[4]	{76,9}	Table[10]	{54,1}
Table[5]			

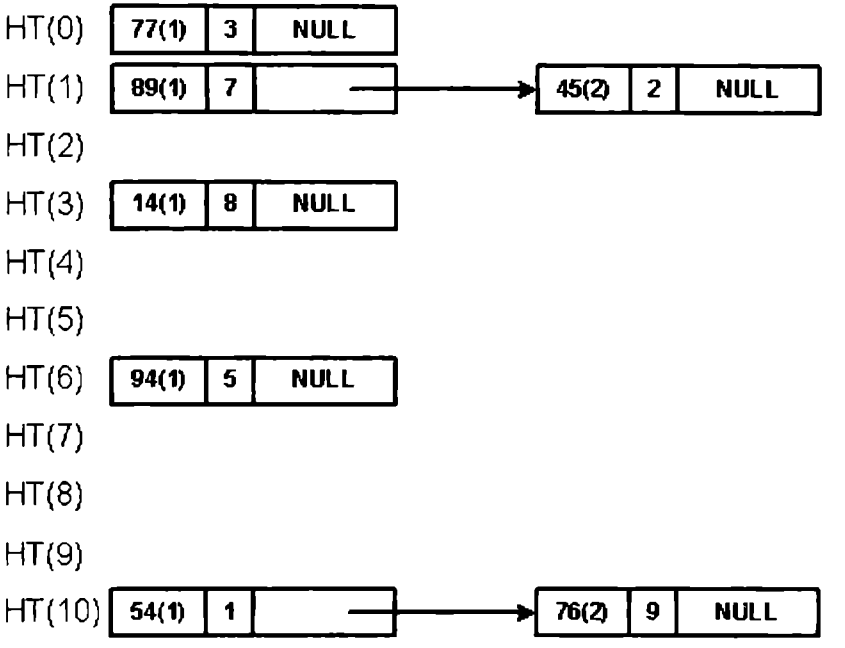
Առաջին բախումը առաջանում է 89 և 45 բանալիների միջև, քանի որ h({89,7})=1 և h({45,2})=1: Տվյալների {89,7} տարրը զբաղեցնում է Table[1]-ը: Եթե փորձենք տեղադրել {45,2} տարրը, ապա պարզվում է,որ Table[1]-ը արդեն զբաղված է: Այս դեպքում սկսվում է աղյուսակի տարրերի գծային հատարկում՝ աղյուսակում ազատ տարր հայտնաբերելու նպատակով: Տվյալ դեպքում դա Table[2]-ն է: Տվյալների {76,9} տարրն ավելացնելիս ալգորիթմի արդյունավետությունը զգալիորեն նվազում է: Հատարկման ընթացքում դիտարկվում է աղյուսակի 5 տարր միչև որ կգտնվի ազատ տեղ Table[4]-ում: Աղյուսակում բոլոր տվյալների տեղադրման համար կկատարվի 13 փորձ, այսինքն միջինում 1.9 փորձ մեկ տարրի համար:

Աղյուսակում կարելի է կատարել տարրերի քառակուսային հատարկում: Այս դեպքում $h_i(x)= (h(x)+i^2)\%N$ (i=1,2,3,...), որտեղ N-ը հեշ-աղյուսակի չափն է: Ընդունված է նաև կրկնակի հեշավորման մեթոդը, որի դեպքում հատարկման քայլը կախված է տեղադրվող գրառման բանալուց և տրվում է h'(x) ֆունկցիայի միջոցով ($h_i(x) = =(h_{i-1}(x)+ h'(x))\%N$) (i=1,2,3,...), որտեղ $h_0(x)= h(x)$, h(x)-ը սկզբնական հեշ-ֆունկցիան է, N-ը հեշ-աղյուսակի չափն է:

Շղթաների մեթոդ: Ինչպես արդեն նշվել է, հեշավորման մեկ այլ մոտեցման դեպքում հեշ-աղյուսակը դիտարկվում է որպես կապակցված գծային ցուցակների զանգված: Ամեն մի ցուցակը կոչվում է սեգմենտ (բլոկ, bucket) և պարունակում է այն գրառումները, որոնց բանալիները հեշ-ֆունկցիայով արտապատկերվում են զանգվածի միևնույն հասցեին: Այս դեպքում x գրառումը, որի համար h(x)=i, ավելացվում է i-րդ կապակցված ցուցակին՝ նոր հանգույցի տեսքով:

Դիտարկենք այս մեթոդի աշխատանքը նախորդ բաժնում բերված օրինակի համար: Այս դեպքում ամեն մի գրառում ավելացվում է համապատասխան ցուցակի վերջից: Նշենք, որ գրառումների բանալիների հետ մեկտեղ փակագծերում տրված է HT աղյուսակում համապատասխան գրառումն ավելացնելու համար կատարվող փորձերի թիվը:

Աղյուսակում բոլոր տվյալների տեղադրման համար կկատարվի 9 փորձ, այսինքն միջինում 1.3 փորձ մեկ տարրի համար:

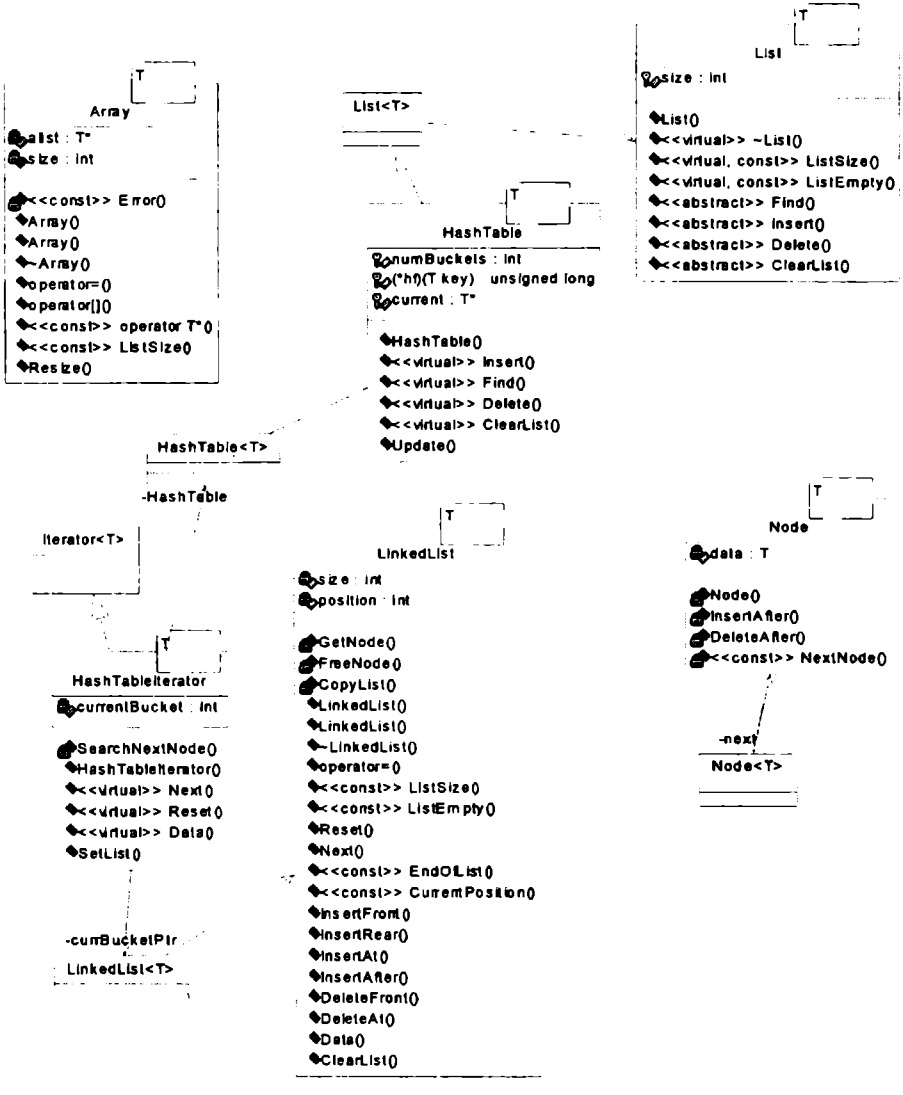


Ընդհանուր դեպքում շղթաների մեթոդը ավելի արագ է աշխատում բաց հասցեավորման մեթոդի համեմատ, քանի որ դիտարկվում են միայն այն բանալիները, որոնք հեշավորվում են միևնույն աղյուսակային հասցեով: Բացի դրանից, բաց հասցեավորումը ենթադրում է աղյուսակի ֆիքսած չափ, այն դեպքում, երբ շղթաների մեթոդում աղյուսակի տարրերը ստեղծվում են դինամիկ եղանակով: Այս մեթոդի հիմնական թերությունն է ցուցիչների համար լրացուցիչ հիշողության անհրաժեշտությունը: Բախումների վերացման համար շղթաների մեթոդը համարվում է ավելի նախընտրելի:

6.3. ԱՂՅՈՒՍԱԿԻ ԻՐԱԿԱՆԱՑՈՒՄԸ C++ ԼԵԶՎՈՎ

Աղյուսակ տվյալների աբստրակտ տիպի բերված իրականացումը հիմնված է բաց հեշավորման սկզբունքի վրա: Ստորև ներկայացված է դասերի UML-դիագրամը: Բոլոր դասերը շաբլոնային են, որոնց համար որպես շաբլոնի պարամետր հանդես է գալիս հեշ-աղյուսակի տվյալների տիպը: Դասերի համախմբի հիմնական դասը HashTable-ն է, որն ածանցված է List աբստրակտ բազային դասից

և որտեղ իրականացված են List դասից ժառանգված վիրտուալ ֆունկցիաները: HashTable-ն օգտագործում է Array դասը՝ հեշ-աղյուսակը որպես կապակցված գծային ցուցակների զանգված պահելու համար: Գծային ցուցակներն ներկայացված են LinkedList դասի շաբլոնի, իսկ ցուցակի հանգույցները՝ Node դասի շաբլոնի օգնությամբ:



HashTable դասը որպես փակ անդամներ պարունակում է ցուցիչ հեշ-ֆունկցիայի վրա, ինչպես նաև ցուցիչ աղյուսակի վերջին դիտարկվող տարրի վրա: Insert մեթոդը որոշում է հեշ-ֆունկցիայի արժեքը և որոնում է LinkedList տիպի համապատասխան օբյեկտը: Եթե այն առկա է աղյուսակում, ապա Update մեթոդը նորացնում է այդ բանալին ունեցող տվյալները, հակառակ դեպքում ցուցակին ավելացվում է նոր տարր: Find և Delete մեթոդները համապատասխանաբար որոնում և հեռացնում են տվյալ բանալիով տվյալները:

HashTable դասի շաբլոնի հայտարարում:

```
template <class T>
class HashTable: public List<T>{
protected:
    int numBuckets; // հեշ-աղյուսակի բլոկների քանակը
                    //(աղյուսակի չափը)
    // հեշ-աղյուսակը որպես կապակցված գծային
    // ցուցակների զանգված
    Array< LinkedList<T> > buckets;
    unsigned long (*hf)(T key); // հեշ-ֆունկցիա
    T *current;   // ցուցիչ աղյուսակի այն տարրի վրա,
                  // որին դիմել եմ վերջին անգամ
public:
    // պարամետրերով կոնստրուկտոր, որին փոխանցում եմ
    // աղյուսակի չափը և հեշ-ֆունկցիան
    HashTable(int nbuckets,
    unsigned long hashf(T key));

    // գծային ցուցակի մշակման մեթոդներ
    virtual void Insert(const T& key);
    virtual int Find(T& key);
    virtual void Delete(const T& key);
    virtual void ClearList(void);
    void Update(const T& key);
};
```

HashTable դասի շաբլոնի իրականացում:

```
template <class T>
void HashTable<T>::Insert(const T& key) {
```

```
    // հաշվել հեշ-ֆունկցիայի արժեքը և lst փոփոխականը
    // տեղադրել համապատասխան գծային ցուցակի առաջին
    // հանգույցի վրա
    int hashval = int(hf(key) % numBuckets);
    LinkedList<T>& lst = buckets[ hashval] ;
    for (lst.Reset();!lst.EndOfList();lst.Next())
        // եթե որոնվող բանալիով տվյալներն առկա են
        // աղյուսակում, ապա վորացնել այդ տվյալները
        if (lst.Data() == key) {
            lst.Data() = key;
            current = &lst.Data();
            return;
        }
    // եթե որոնվող բանալիով տվյալներ չկա աղյուսակում,
    // ապա այդ տվյալներն ավելացնել աղյուսակին
    lst.InsertRear(key);
    current = &lst.Data();
    size++;
}

template <class T>
int HashTable<T>::Find(T& key) {

    // հաշվել հեշ-ֆունկցիայի արժեքը և lst փոփոխականը
    // տեղադրել համապատասխան գծային ցուցակի առաջին
    // հանգույցի վրա
    int hashval = int(hf(key) % NumBuckets);
    LinkedList<T>& lst = buckets[ hashval] ;
    // դիտարկել գծային ցուցակի հանգույցները բանալին
    // գտնելու նպատակով
    for (lst.Reset(); !lst.EndOfList(); lst.Next())
        // եթե որոնվող բանալիով տվյալներն առկա են
        // աղյուսակում, ապա արժեքավորել current
        // փոփոխականը և վերադարձնել TRUE, այլապես
        // վերադարձնել FALSE
        if (lst.Data() == key){
            key = lst.Data();
            current = &lst.Data();
            return 1;
        }
    return 0;
}
```

Հետևյալ օրինակում ստեղծվում է 101 տարր պարունակող հեշ-աղյուսակ, որը լրացվում է NameRecord տիպի երկու գրառումներով: Գրառումն աղյուսակին ավելացնելու համար կիրառվում է List աբստրակտ դասից ժառանգված Insert ֆունկցիան: Այնուհետև Find ֆունկցիայով փնտրվում է գրառումներից մեկը և նորացվում է Update ֆունկցիայով :

```
int main(){
    struct NameRecord {
        String name;
        int count;
    };
    // 101 տարր պարունակող NameRecord տիպի տվյալներով
    // աղյուսակ hash հեշ-ֆունկցիայով
    HashTable<NameRecord> HF(101,hash);
    // {"Betsy",1} գրառումն ավելացնել աղյուսակին
    NameRecord rec;
    rec.name = "Betsy";
    rec.count = 1;
    HF.Insert(rec);
    // {"Harry",2} գրառումն ավելացնել աղյուսակին
    rec.name = "Harry";
    rec.count = 2;
    HF.Insert(rec);
    cout << HF.ListSize(); //տպել աղյուսակի չափը

    // գտնել "Betsy" բանալուն համապատասխանող գրառումը,
    // հաշվիչի դաշտը ավելացնել մեկով և փոփոխել գրառումը
    rec.name = "Betsy";
    //գտնել "Betsy" բանալիով գրառումը
    if (HF.Find(rec){
          rec.count += 1; // փոփոխել տվյալների դաշտը
    // փոփոխել գրառումը աղյուսակում
          HF.Update(rec);
    }
    else
        cerr<<"Սխալ:Betsy բանալիով գրառում
             աղյուսակում չկա:\n";
    return 0;
}
```

ԽՆԴԻՐՆԵՐ

1. Հեշ-աղյուսակի միջոցով իրականացնել «Տողերի բազմություն» տվյալների աբստրակտ տիպը: Տողն իրենից ներկայացնում է ոչ դատարկ, և 10-ից ոչ ավել լատինական փոքրատառեր պարունակող հաջորդականություն: Աղյուսակի չափը հավասար է 2048-ի: Դիտարկել հետևյալ հեշ-ֆունկցիան h(x)= (լատինական այբուբենում տողի տառերի դիրքերի գումար) mod 2048: Նախատեսել բազմությանը տողի ավելացման և տվյալ տողի բազմությանը պատկանելիության ստուգման գործողությունները: Բախումների վերացման համար կիրառել բաց հասցեավորման դեպքում կիրառվող հատարկման մեթոդը:

2. Հեշավորմամբ և շղթաների մեթոդով իրականացնել Table աբստրակտ աղյուսակը: Հեշավորման աղյուսակի չափը հավասար է 10000-ի (HASH_TABLE_SIZE=10000): Փնտրման բանալիներն ամբողջ թվեր են, որոնք փոփոխվում են 0-ից մինչև 9999: Հեշավորման ֆունկցիան տրված է C++-ի հետևյալ ֆունկցիայով՝

```
int hashIndex(int x)
{
   for (int i=0; i<=10000; ++i)
   key=x*x% HASH_TABLE_SIZE;
   return key;
}
```

Գնահատել հեշավորման ֆունկցիան: Ո՞ր տվյալները կբաշխվեն միևնույն բջիջներում: Նախատեսել աղյուսակին տարրի ավելացման և աղյուսակից տվյալ տարրի հեռացման գործողությունները:

3. Նկարագրել անունների symbolTable աղյուսակը, որի օգնությամբ կոմպիլյատորը հետևում է ծրագրում օգտագործված իդենտիֆիկատորներին: Հայտնաբերելով իդենտիֆիկատոր, կոմպիլյատորն իրականացնում է փնտրում անունների աղյուսակում: Եթե իդենտիֆիկատորն առաջին անգամ է հանդիպել, այն ավելացվում է աղյուսակին: Աղյուսակն իրականացնել հեշավոր-

ման միջոցով: Օգտագործել հեշավորման H(x)=x mod tableSize ֆունկցիան և Հորների սխեմայով իդենտիֆիկատորը ամբողջ թվի վերածելու ալգորիթմը: Բախումների վերացման համար կիրառել քառակուսային հատարկման մեթոդը:

4. Լուծել նախորդ խնդիրը՝ աղյուսակը գրառելով դինամիկ զանգվածում: Եթե հեշ-աղյուսակը լցվում է կեսից ավել, ավելացնել նրա չափը մինչև մոտակա պարզ թիվը, որը գերազանցում է 2*tableSize-ը:

5. Մշակել HashTableIterator դասը, որը պարունակում է մեթոդներ HashTable տիպի աղյուսակի հետ կապված գծային ցուցակներում ընդգրկված տարրերը հատարկելու համար:

ՀԱՎԵԼՎԱԾ

Լուծված խնդիրներ

Խնդիր 2.25. *Բազմանդամը ներկայացվում է միակապ գծային ցուցակի տեսքով: Յուրաքանչյուր հանգույցում նշվում է հերթական միանդամի գործակիցը, աստիճանը և կապը հաջորդ հանգույցի հետ: Աստիճանները դասավորված են ըստ միանդամների աստիճանների նվազման կարգի: Օգտագործվում է վերնագրային հանգույցով ցուցակ: Իրականացնել երկու բազմանդամների գումարման գործողությունը:*

```
#include<iostream>
using namespace std;
// միանդամի կառուցվածքի նկարագրությունը
struct node {
    float    coef;        // միանդամի գործակիցը
    int      exp;         // միանդամի աստիճանը
    node     *link;       // կապը հաջորդ հանգույցի հետ
    // կոնստրուկտոր
    node(float c=1,int e=0,node *ptr=NULL):
    coef(c),exp(e),link( ptr){}
};
// իրական գործակիցներով բազմանդամների դաս
class polinom {
    // բազմանդամի արտածում
    friend polinom operator+(polinom&, polinom&);
    // երկու բազմանդամների գումարում
    friend ostream& operator<<(ostream&, polinom&);
private:
    node* first;          // ցուցակի սկզբի հասցեն
public:
    polinom();            // լռելիությամբ կոնստրուկտոր
    polinom(polinom&);    // պատճենահանման կոնստրուկտոր
    polinom(float*,int); // բազմանդամի կառուցում
                          // գործակիցների
                          // հաջորդականությունից
    ~polinom();           // դեստրուկտոր
};
```

```
// polinom դասի մեթոդների իրականացումը
// կոնստրուկտոր,որն արժեքավորում է դատարկ բազմանդամ
polinom::polinom(){
    first = new node;
}
// պատճենահանման կոնստրուկտոր
polinom::polinom(polinom& p){
    node *t1, *t2=p.first->link;
    first = new node;
    t1 = first;
    while(t2){
        t1->link=new node (t2->coef,t2->exp);
        t1 = t1->link;
        t2 = t2->link;
    }
}
// կոնստրուկտոր,որն արժեքավորում է բազմանդամը
// գործակիցների հաջորդականությունից
polinom::polinom(float *a,int n){
    node *t;
    first = new node;
    t = first;
    for(int i = 0;i < n; i++)
        // ոչ զրոյական գործակցով միանդամը դրվում է
        // ցուցակում
        if(a[ i] != 0){
            t->link = new node(a[ i] ,n-i-1);
            t= t->link;
        }
}
// դեստրուկտոր
polinom::~polinom(){
    node *t = first->link;
    while(first->link) {
        delete first;
        first = t;
        t = t->link;
    }
    delete first;
}
```

```
// երկու բազմանդամների գումարման գործողություն
polinom operator+(polinom& p1,polinom& p2){
    polinom p3;
    // p3-ը բազմանդամների գումարը usանալու համար է
    // ցուցիչներ՝ p1,p2,p3 ցուցակների հանգույցներով
    // շարժվելու համար
    Node *t1=p1.first->link;
    Node *t2=p2.first->link;
    Node *t3=p3.first;
    // քանի դեռ չենք հասել բազմանդամներից որևէ մեկի
    // վերջին
    while(t1 && t2) {
        // եթե առաջին բազմանդամի հերթական միանդամի
        // աստիճանը ավելի մեծ է, քան երկրորդ
        // բազմանդամի հերթական միանդամի աստիճանը
        if(t1->exp > t2->exp) {
            t3->link=new node(t1->coef, t1->exp);
            t1=t1->link;
            t3=t3->link;
        }
        else if(t1->exp < t2->exp) {
            // եթե առաջին բազմանդամի հերթական
            // միանդամի աստիճանը ավելի փոքր է,
            // քան երկրորդ բազմանդամի հերթական
            // միանդամի աստիճանը
            t3->link=new node(t2->coef, t2->exp);
            t2 = t2->link ;
            t3=t3->link ;
        }
        // եթե առաջին և երկրորդ բազմանդամների հերթական
        // միանդամների աստիճաններն իրար հավասար են
        else {
            if(t1->coef + t2->coef != 0) {
                // եթե գործակիցների գումարը զրո չէ
                t3->link=new
                node(t2->coef+t1->coef,t2->exp);
                t3=t3->link;
            }
            t1=t1->link;
            t2=t2->link;
        }
    }
```

```
    // t1 ցուցիչն ուղղում ենք չավարտված ցուցակի
    // շարունակության վրա
    if(t2)
        t1=t2;
    // չավարտված ցուցակի շարունակությունն ավելացնում ենք
    // գումար բազմանդամի վերջից
    while(t1) {
        t3->link= new node (t1->coef,t1->exp);
        t1=t1->link;
        t3=t3->link;
    }
    return p3;
}

// բազմանդամի արտածում a0x^n+a1x^n-1+...+an-1x+an տեսքով
ostream& operator <<(ostream& out, polinom& p) {
    out<<endl;
    node *t;
    t = p.first->link;
    while(t) {
        // դրական գործակիցներից առաջ դնել '+' նշան,
        // բացի առաջինից
        if((t->coef>0)&& (t!=p.first->link))
            out<<'+';
        if(t->coef!=1) // մեկի հավասար գործակիցները
                       // չտպել
            out<<t->coef;
        if (t->exp!=0) { // 0-ի հավասար x-ի աստիճանը
                         // չտպել
            out<<"x";
            if(t->exp!=1 && t->exp!= 0)
                // 0-ի և 1-ի հավասար
                // աստիճանները չտպել
            out<<"^"<<t->exp;
        }
        t=t->link;
    }
    return out;
}
```

```
// main() ֆունկցիայում  ըստ գործակիցների զանգվածների
// կառուցվում են երկու բազմանդամներ և արտածվում
// այնուհետև ստացվում է նրանց գումար բազմանդամը և
// արտածվում
int main() {
    const int     m=4, n=3;
    float         p1[ n] ={ 3,0,7} ;
    float         p2[ m] ={ 8,-3,1,0} ;
    polinom       P1(p1,n);
    polinom       P2(p2,m);
    cout<<P1;
    cout<<P2;
    cout<<P1+P2;
    return 0;
}
```

Խնդիր3.1. Իրականացնել պահունակը դինամիկ զանգվածի միջոցով:

```
#include <iostream.h>
template <class T>
class stack {
    int top;   // պահունակի գագաթի տարրի ինդեքսը
    int size;  // պահունակին հատկացվող զանգվածի չափը
    T *S;      // պահունակի համար նախատեսված
               // զանգվածի ցուցիչը
public:
    // դատարկ պահունակի ստեղծում
    stack(const int=100);
    ~stack();      // դեստրուկտոր
    int getsize();  // պահունակի տարրերի քանակը
    int getmaxsize();   // պահունակի զանգվածի չափը
    void push(const T&); // ավելացնել տարր պահունակին
    void pop();    // հեռացնել տարր պահունակից
    T& getTop();   // կարդալ պահունակի գագաթի տարրը
    bool isEmpty(); // ստուգել պահունակի դատարկ լինելը
    bool isFull(); // ստուգել պահունակի լցված լինելը
};
```

```
// stack դասի մեթոդների իրականացումը
// դասարկ պահուճակի ստեղծման կոնստրուկտոր
template <class T>
stack<T>::stack(const int n) {
    S=new T[ n] ;      // դինամիկ զանգվածի հատկացում
    size=n;
    top=-1;
}

//դեստրուկտոր
template <class T>
stack<T>::~stack(){
    delete[ ] S;
}

// սահմանափակման տարրերի քանակը
template <class T>
int stack<T>::getsize() {
    return top+1;
}

// սահմանափակման համար նախատեսված զանգվածի չափը
template <class T>
int stack<T>::getmaxsize() {
    return size;
}

// ավելացնել տարր պահուճակին
template <class T>
void stack<T>::push(const T& elem) {
    S[ ++top] =elem;
}

// հեռացնել տարր պահուճակից
template <class T>
void stack<T>::pop() {
    top--;
}

// կարդալ պահուճակի գագաթի տարրը
template <class T>
T& getTop() {
    return S[ top] ;
}
```

```
// usուգել պահուճակի դատարկ լինելը
template <class T>
bool stack<T>::isEmpty() {
    return top==-1;
}

// usուգել պահուճակի լցված լինելը
template <class T>
bool stack<T>::isFull() {
    return top+1==size;
}
```

✓ Խնդիր 3.4. Տրված են ամբողջ թվերի S1, նշանների S2 և իրական թվերի S3 պահուճակները: Մեկուսել այդ պահուճակների տարրերը մինչև բոլոր պահուճակների դատարկվելը:

```
#include <iostream>
#include <stack>    // օգտագործված է STL-ի stack դասը
using namespace std;
int main() {
    stack<int>      s1;        // ամբողջ թվերի պահուճակ
    stack <char>    s2;        // նշանների պահուճակ
    stack <float>   s3;        // իրական թվերի պահուճակ
    // օժանդակ փոփոխականներ
    int             intVar;
    char            charVar;
    float           floatVar;
    int   n1;   // s1 պահուճակում դրվող տարրերի քանակը
    int   n2;   // s2 պահուճակում դրվող տարրերի քանակը
    int   n3;   // s3 պահուճակում դրվող տարրերի քանակը
    // ներմուծել պահուճակների չափերը
    cout<<endl<<"enter the s1 stack's size ";
    cin>>n1;
    cout<<endl<<"enter the s2 stack's size ";
    cin>>n2;
    cout<<endl<<"enter the s3 stack's size ";
    cin>>n3;
```

```
cout<<endl<<"enter the s1 stack's values ";
for(int i=0;i<n1;i++){
    cin>> intVar;
    s1.push(intVar);
}
cout<<endl<<"enter the s2 stack's values ";
//s2 պահուճակի ստեղծում
for( i=0;i<n2;i++){
    cin>> charVar;
    s2.push(charVar);
}
cout<<endl<<"enter the s3 stack's values ";
for( i=0;i<n3;i++) {
    cin>>floatVar;
    s3.push(floatVar);
}
int n;              // պահուճակներից ամենամեծի չափը
n1= s1.size();
n2= s2.size();
n3= s3.size();
n=(n1<n2)?n2:n1;
n=(n<n3)?n3:n;
// պահուճակների տարրերի մեկումեջ արտածում
for(i=1;i<=n;i++) {
    if(!s1.empty()){
    // արտածել s1 պահուճակի գագաթի տարրը
        cout<<s1.top()<<" ";
        s1.pop();     // հեռացնել տարրը պահուճակից
    }
    if(!s2.empty()) {
    // արտածել s2 պահուճակի գագաթի տարրը
        cout<<s2.top()<<" ";
        s2.pop();     // հեռացնել տարրը պահուճակից
    }
```

```
        if(!s3.empty()) {
        // արտածել s3 պահունակի գագաթի տարրը
            cout<<s3.top()<<" ";
            s3.pop();     // հեռացնել տարրը պահունակից
        }
        cout<<endl;
    }
    return 0;
}
```

Խնդիր 3.11. Թվաբանական արտահայտությունը ներկայացվում է նիշերի հաջորդականության միջոցով: Ստուգել նրանում եղեք տեսակի { }, [ ], ( ) փակագծերի ներդրվածության ճշտությունը:

Խնդրի լուծման համար առաջարկվում է հետևյալ ալգորիթմը:

Եթե հաջորդականության հերթական նիշը բացվող փակագիծ է, ապա այն դրվում է պահունակում: Եթե հերթական նիշը փակվող փակագիծ է, ապա այն համեմատվում է պահունակի գագաթի տարրի հետ: Եթե փակագծերի տիպերը համընկնում են, ապա պահունակից հանվում է այդ փակագիծը, այլապես, փակագծերի բալանսի խախտման պատճառով ալգորիթմն ավարտում է իր աշխատանքը: Եթե փակվող փակագիծ հանդիպելու պահին պահունակը դատարկ է, ապա դարձյալ արձանագրվում է փակագծերի բալանսի խախտում (ավելորդ փակվող փակագծի հանդիպման դեպք): Եթե արտահայտության մեջ մտնող բոլոր նիշերը դիտարկելուց հետո պահունակը չի դատարկվել, ապա դա նույնպես փակագծերի բալանսի խախտում է (ավելորդ բացվող փակագծի հանդիպման դեպք):

```
#include <iostream>
#include <stack>
using namespace std;
int main() {
// թվաբանական արտահայտության մեջ պարունակվող
// նիշերի առավելագույն քանակը
   const int n=100;
   char a[ n] ;       // թվաբանական արտահայտությունը
   cin>>a;            // թվաբանական արտահայտության մուտք
```

```
stack<char> S; // բացվող փակագծերի պահուստակ
int i=0;  // օժանդակ փոփոխական
// քանի դեռ չեն դիտարկվել թվաբանական
// արտահայտության բոլոր նիշերը
while(a[ i] ) {
   switch(a[ i] ) {
      // բացվող փակագիծը դրվում է պահուստակի մեջ
      case '{ ':
      case '[ ':
      case '(':
         S.push(a[ i] );
         break;
      case '} ':
         if(S.empty()) {
         // հանդիպել է ավելորդ "}" փակագիծ
            cout<<"Not balanced";
            return 0;
         }
         else if(S.top()!= '{ '){ // բացվող և փակվող
                // փակագծերի անհամապատասխանություն
            cout<<"Not balanced";
            return 0;
         }
         else
         // բալանսի խախտում չի հայտնաբերվել,
         // համապատասխան բացվող փակագիծը
         // հանվում է պահուստակից
            S.pop();
            break;
      case '] ':
         if(S.empty()) {
                // հանդիպել է ավելորդ "]" փակագիծ
            cout<<"Not balanced";
            return 0;
         }
         else if(S.top()!='[ ') { // բացվող և փակվող
                // փակագծերի անհամապատասխանություն
```

```
            cout<<"Not balanced";
            return 0;
          }
          else
          // բալանսի խախտում չի հայտնաբերվել,
          // համապատասխան բացվող փակագիծը
          // հանվում է պահուցակից
            S.pop();
          break;
        case ')':
          if(S.empty()) {
          //հանդիպել է ավելորդ ")" փակագիծ
            cout<<"Not balanced";
            return 0;
          }
          else if(S.top()!='(') { //բացվող և փակվող
                  // փակագծերի անհամապատասխանություն
            cout<<"Not balanced";
            return 0;
          }
          else
          // բալանսի խախտում չի հայտնաբերվել,
          // համապատասխան բացվող փակագիծը
          // հանվում է պահուցակից
            S.pop();
          break;
      }
      i++;
    }
    // բացվող փակագծերի քանակը գերազանցում է
    // փակվող փակագծերի քանակից
    if(!S.empty())
      cout<<"Not balanced";
    else
      cout<<"Is balanced";
    return 0;
}
```

? ***Խնդիր 4.2.*** *Իրականացնել հերթի շրջանաձև մոդելը առանց հերթի տարրերի քանակի փոփոխականի օգտագործման:*

Այս իրականացման մեջ նախատեսված է մեկ "դատարկ" տարրի առկայություն հերթի սկզբում: Այն ապահովում է դատարկ և լցված հերթերի տարբերումը միմյանցից՝ առանց հերթի տարրերի քանակի ստուգման:

```
#include <iostream>
#include <string>
#include <exception>
using namespace std;
// բացառիկ իրավիճակների Queue_Exception դաս
class Queue_Exception:public exception {
private:
    char *message;
public:
   Queue_Exception();  // լռելիությամբ կոնստրուկտոր
    Queue_Exception(char *s) {
    // բացառիկ իրավիճակի արժեքավորում
        message =new char[ strlen(s)+1] ;
        strcpy(message, s);
    }
    void show() {
        cout<< message;
    }
} ;
//Queue դաս
template <class T, int n>
class Queue {
private:
    int front;  // հերթի առաջին տարրի դիրքը
    int rear;   // հերթի վերջին տարրին հաջորդող դիրքը
    T Q[ n] ;      // հերթն իրականացնող զանգված
public:
    Queue() { //դատարկ հերթի արժեքավորում
        front=0;
        rear=1;
    }
```

```
    void push(int x) { //տարրի ավելացում հերթին
        if(full())
            throw Queue_Exception("Queue is full");
        Q[ rear] =x;
        rear=(++rear)%n;
    }
    // հերթի լցված լինելու պայմանի ստուգում
    bool full() {
        return rear==front;
    }
    // հերթի դատարկ լինելու պայմանի ստուգում
    bool empty() {
        return (rear-front)%n==1;
    }
    void pop() { //տարրի հեռացում հերթից
        if (empty())
            throw Queue_Exception("Queue is empty");
        front=(front++)%n;
    }
    T peek() { //հերթի առաջին տարրի ստացում
        if (empty())
            throw Queue_Exception("Queue is empty");
        return Q[ (front+1)%n] ;
    }
    void clear() { //դատարկել հերթը
        front=0;
        rear=1;
    }
    int size() { //հերթի տարրերի քանակի հաշվում
        return n-(rear-front)%n;
    }
};

// ծրագրում ստեղծվում է m տարր պարունակող ամբողջ
// թվերի հերթ,
// որից այնուհետև կարդացվում և հեռացվում են n տարրեր
// m-ի և n-ի որոշակի ընտրությունների դեպքում
// կարող են առաջանալ բացառիկ իրավիճակներ
int main() {
```

```
    try {
        Queue<int,10> Q;
        int  x;
        int  m;
        int  n;
        cout<<"Enter m";
        cin>>m;
        for(int i=1;i<=m;i++) {
            cin>>x;
            Q.push(x);
        }
        cout<<endl<<"Enter n";
        cin>>n;
        cout<<endl;
        for(i=1;i<=n;i++) {
            cout<<Q.peek()<<" ";
            Q.pop();
        }
    } catch(Queue_Exception & qe) {
        qe.show();
    }
    return 0;
}
```

Խնդիր4.3. Տրված է բնական միանիշ և երկնիշ թվերի $a_0,a_1,...a_{n-1}$ հաջորդականությունը: Դասավորել այն չնվազման կարգով:

Խնդիրը լուծելու համար օգտվենք բնական թվերի կարգավորման ռադիքս(radix) ալգորիթմից: Համաձայն այդ ալգորիթմի նախ թվերը վերադասավորվում են ըստ միավորների կարգի, ապա ըստ տասնավորների կարգի: Ալգորիթմի իրականացման համար օգտագործվում են է հերթերի երկու հաջորդականություն՝ $Q1_0$, $Q1_1$,...$Q1_9$ և $Q2_0$, $Q2_1$,...$Q2_9$: Առաջին փուլում ամեն մի տարր դրվում է այն $Q1_i$ հերթում, որի ինդեքսի հետ համընկնում է նրա միավորը: Երկրորդ փուլում $Q1_0$, $Q1_1$,...$Q1_9$ հերթերից տարրերը հաջորդաբար հանվում են և դրվում հերթերի երկրորդ՝ $Q2_0$, $Q2_1$,...$Q2_9$ տասնյակի մեջ: Յուրաքանչյուր հանված տարր դրվում է այն $Q2_i$ հերթում, որի ինդեքսի հետ համընկնում է նրա տասնավորը: Երրորդ փուլում տարրերը հանվում են $Q2_0$, $Q2_1$,...$Q2_9$ հերթերից և դրվում հաջորդականության մեջ: Արդյունքում ստացվում է կարգավորված հաջորդականություն:

```
#include<iostream>
#include <queue>
using namespace std;
// օգտագործված է STL-ի queue դասը
int main() {
    const n=5;
    // հաջորդականությունը, որը պետք է կարգավորել
    int a[ n] ;
    int x;
    // հերթերի հաջորդականությունները
    queue<int> Q1[ 10] ,Q2[ 10] ;
    for(int i=0;i<n;i++)
        cin>>a[ i] ;     // հաջորդականության ներմուծում
    // հաջորդականության տարրի փոխանցում հերթերի
    // առաջին տասնյակի մեջ
    for(i=0;i<n;i++)
        Q1[ a[ i] %10] .push(a[ i] );
    // տարրերի հանում հերթերի առաջին տասնյակից և
    // դրանց փոխանցում հերթերի երկրորդ տասնյակի մեջ
    for(i=0;i<10;i++)
        while(!Q1[ i] .empty()) {
            x=Q1[ i] .front();
            Q1[ i] .pop();
            Q2[ x/10] .push(x);
        }
    int j=0;
    // տարրերի փոխանցում հաջորդականության մեջ
    for(i=0;i<10;i++)
        while(!Q2[ i] .empty()) {
            a[ j++] =Q2[ i] .front();
            Q2[ i] .pop();
        }
    // կարգավորված հաջորդականության արտածում
    for(i=0;i<n;i++)
        cout<<a[ i] <<" ";
    cout<<endl;
    return 0;
}
```

Խնդիր 4.4. Տրված է բնական թվերի $a_0,a_1,...a_{n-1}$ հաջորդականությունը: Դասավորել այն չնվազման կարգով:

```
#include <iostream>
#include <queue>     //օգտագործված է STL-ի queue դասը
using namespace std;
// ֆունկցիան հաշվում է բնական թվի թվանշանների քանակը
int length(int number) {
    int ret=0;
    while(number) {
        ret++;
        number/=10;
    }
    return ret;
}

// ծրագրում իրականացվում է բնական թվերի կարգավորում
// ռադիքս(radix) ալգորիթմով
int main() {
    const int MAXSIZE=10000;
    int a[ MAXSIZE] ;
    int i,j,k;
    int n;
    cin>>n;
    for(i=0;i<n;i++)  // հաջորդականության ներածում
        cin>>a[ i] ;
    queue<int> Q[ 10] ; // հերթերի զանգված
    int degrees[ 10] ;  // հերթերի չափերը մինչ տարրերի
                        // նոր վերադասավորությունը
                        // ըստ հերթական դիրքի թվանշանի
    int deg10=1;        // տասի հերթական աստիճանը
    int maxlen=0;       // զանգվածի տարրերի թվանշանների
                        // առավելագույն քանակությունը
    int currentlen;     // ընթացիկ տարրի թվանշանների
                        // քանակը
    int elem;           // օժանդակ փոփոխական
                        // զանգվածի տարրերի թվանշանների
                        // առավելագույն քանակի որոշում
```

```
    for (i=0;i<n;i++) {
        currentlen=length(a[ i] );
        maxlen=maxlen>currentlen?maxlen:currentlen;
    }
    // զանգվածի տարրերի փոխանցում համապատասխան
    // հերթերի մեջ
    for (i=0;i<n;i++)
        Q[ a[ i] %10] .push(a[ i] );
    for(k=2;k<=maxlen;k++) {
        deg10*=10;
        for (i=0;i<10;i++)
        degrees[ i] =Q[ i] .size();
        // տարրերի վերադասավորում համապատասխան
        // հերթերի մեջ
        for (i=0;i<10;i++)
            for (j=0;j<degrees[ i] ;j++) {
                elem=Q[ i] .front();
                Q[ i] .pop();
                Q[ (elem/deg10)%10] .push(elem);
            }
    }
    n=0;
    // տարրերի փոխանցում հերթերից զանգվածի մեջ
    for (i=0;i<10;i++)
        while(!Q[ i] .empty()) {
            a[ n++] =Q[ i] .front();
            Q[ i] .pop();
        }
    // կարգավորված հաջորդականության արտածում
    for (i=0;i<n;i++)
        cout << a[ i] << ' ';
    cout << endl;
    return 0;
}
```

Խնդիր 5.8. Գտնել տրված բինար ծառի գագաթների քանակը:

```
// իտերատիվ եղանակ
#include <iostream>
#include <stack>
using namespace std;
int countOfNodes =0;
```

```
//գլոբալ փոփոխական ծառի գագաթների քանակի համար
// TreeNode դասի հայտարարում
struct TreeNode {
    TreeNode *left;
    TreeNode *right;
    int      data;
    //կոնստրուկտոր
    TreeNode(int d=1,TreeNode *l=NULL,
    TreeNode *r=NULL) {
        data=d;
        left=l;
        right=r;
    }
    // դեստրուկտոր
    ~TreeNode(){}
};
// բինար ծառի ուղիղ շրջանցման ֆունկցիա,
// որի visit պարամետր-ֆունկցիայով տրվում է
// ծառի գագաթի մշակման եղանակը
void preorder(TreeNode *t,void visit(TreeNode *)) {
    stack<TreeNode *> st;
    TreeNode*p=t;
    bool b=true;
    while(b) {
        while(p) {
            visit(p);
            st.push(p);
            p=p->left;
        }
        if(st.empty())
            b=false;
        else {
            p=st.top();
            st.pop();
            p=p->right;
        }
    }
}

// ֆունկցիան մշակում է ծառի գագաթը
void countOfNodesCalculation(TreeNode *p) {
            countOfNodes ++;
}
```

```
// բինար ծառի գագաթների քանակի որոշման օրինակ
int main() {
    //բինար ծառի կառուցում
    TreeNode *d=new TreeNode(4);
    TreeNode *e=new TreeNode(5);
    TreeNode *g=new TreeNode(6);
    TreeNode *b=new TreeNode(2,d,e);
    TreeNode *c=new TreeNode(3,NULL,g);
    TreeNode *a=new TreeNode(1,b,c);
          // կառուցված ծառի գագաթների քանակի
          // հաշվում և արտածում
    preorder(a,countOfNodesCalculation);
    cout<< countOfNodes;
    return 0;
}
// ռեկուրսիվ եղանակ
#include <iostream>
using namespace std;
// TreeNode տիպի սահմանումը բերված է սույն խնդրի
// իտերատիվ լուծման տարբերակում
class TreeNode;
// ֆունկցիան հաշվում է t ցուցիչով որոշվող
// ծառի հանգույցների քանակը
int size_R(TreeNode* t) {
    if(!t)
        return 0;
        return 1+size_R(t->left)+size_R(t->right);
}
// size_R() ֆունկցիայի կիրառման օրինակ
int main() {
    TreeNode *d=new TreeNode(4);
    TreeNode *e=new TreeNode(5);
    TreeNode *g=new TreeNode(6);
    TreeNode *b=new TreeNode(2,d,e);
    TreeNode *c=new TreeNode(3,NULL,g);
    TreeNode *a=new TreeNode(1,b,c);
    cout<<size_R(a);   // size_R() ֆունկցիայի կանչ
    return 0;
}
```

Խնդիր 5.10. Գտնել տրված բինար ծառի բարձրությունը:

```
// ռեկուրսիվ եղանակ
// TreeNode տիպի սահմանումը տես խնդիր 5.8-ում
// երկու թվերից մեծագույնը որոշող օժանդակ ֆունկցիա
int max(int x, int y) {
    return (x>y)?x:y;
}

// high() ռեկուրսիվ ֆունկցիան վերադարձնում է
// t ցուցիչով որոշվող ծառի բարձրությունը
int high(TreeNode *t) {
    if (t==NULL)
        return -1;
    return max(high(t->left),high(t->right))+1;
}
```

Խնդիր 5.12. Կառուցել բինար ծառ ըստ տրված զանգվածի:

Ծառի վերաբերյալ ինֆորմացիան զանգվածում պահվում է հետևյալ կերպ: Նախապես ծառը լրացվում է պայմանական գագաթներով՝ մինչև նույն բարձրության լրիվ ծառ ստանալը: Զանգվածի յուրաքանչյուր տարր համապատասխանում է ստացված լրիվ ծառի ըստ մակարդակների ձախից-աջ շրջանցման հերթական գագաթին: Այն երկու դաշտից բաղկացած կառուցվածք է: Առաջին դաշտում դրվում է գագաթում պարունակվող արժեքը (եթե ծառում այդ գագաթը չկա, ապա դրվում է ինչ-որ պայմանական արժեք): Երկրորդ դաշտում դրվում է մեկ, եթե ծառի ըստ մակարդակների ձախից աջ շրջանցման հերթական գագաթն առկա է նախնական ծառում, և զրո՝ հակառակ դեպքում:

```
// ստորև բերվող ծրագիրը զանգվածից կառուցում է ծառ
// ըստ վերը բերված ներկայացման
// որպես ոչ գծային կապակցված ցուցակ
#include <iostream>
#include <queue>   // օգտագործված է STL-ի queue դասը
using namespace std;
// բինար ծառի հանգույցի հայտարարում
template <class T>
struct TreeNode {
```

```
    T info;
    TreeNode<T>*  right;    // գագաթի աջ որդու ցուցիչ
    TreeNode<T>*  left;     // գագաթի ձախ որդու ցուցիչ
    TreeNode<T>(T x,TreeNode<T> *l = NULL,
    TreeNode<T> *r = NULL):
        info(x),left(l),right(r){}
};

// գաճգվածի տարրի հայտարարում
template <class T>
struct el {
    T       data;      // ծառի գագաթի արժեքը
    int     mark;      // ցշում գագաթի պայմանական կամ
                       // ոչ պայմանական լինելու
                       // վերաբերյալ
    el():data(){}      // լռելիությամբ կոնստրուկտոր
    el(T d,int m):data(d),mark(m){} // կոնստռուկտոր
};

// tree դասի հայտարարում
template <class T>
class tree {
private:
    TreeNode<T> *root;      // ծառի արմատի ցուցիչ
public:
    tree(el<T> *,int);      // կոնստրուկտոր, որն
                            // արժեքավորում է ծառը
                            // գաճգվածից
};

// արտածում է ծառի գագաթների արժեքները ձախից աջ
// լայնակի շրջանցումով բացակայող գագաթների փոխարեն
// արտածվում է "-" նիշը
template <class T>
tree<T>::tree(el<T> *d,int n) {
    // ծառի գագաթների ցուցիչների հերթ
    queue<TreeNode<T> *> q;
    root=new TreeNode<T>(d[ 0] .data);
    TreeNode<T>* t=root;
    cout<<t->info<<" ";
    q.push(t);
```

```
    for(int i=1;i<n;i+=2){
        t=q.front();
        if(d[ i] .mark==1){
                // ստեղծվում է ծառի հերթական գագաթը
                // որպես t-ի ձախ որդի
            t->left=new TreeNode<T>(d[ i] .data);
            if((2*i+1<n)&&d[ 2*i+1] .mark==1||
                    (2*i+2)<n&&d[ 2*i+2] .mark==1)
                    // եթե հերթական գագաթը տերև չէ,
                    // ապա այն դրվում է հերթում
                q.push(t->left);
            // գագաթի արտածում
            cout<<t->left->info<<" ";
        }
        else   // եթե գագաթը պայմանական է, արտածվում է
               // "-" նիշը
            cout<<"-";
        if(d[ i+1] .mark==1){
                // ստեղծվում է ծառի հերթական գագաթը
                // որպես t-ի աջ որդի
            t->right=new TreeNode<T>(d[ i+1] .data);
            if((2*i+3)<n&&d[ 2*i+3] .mark==1||
                    (2*i+4)<n&&d[ 2*i+4] .mark==1)
                    // եթե հերթական գագաթը տերև չէ,
                    // ապա այն դրվում է հերթում
                q.push(t->right);
            // գագաթի արտածում
            cout<<t->right->info<<" ";
        }
        else // եթե գագաթը պայմանական է, արտածվում է
             // "-" նիշը
            cout<<"-";
        if(t->left||t->right)
                  // եթե դիտարկված t գագաթի համար
                  // ցիկլի այս քայլում ստեղծվեց գոնե
                  // մեկ որդի, ապա t-ն հանվում է հերթից
                q.pop();
    }
}
```

```
// բինար ծառի կառուցում ըստ ստեղնաշարից ներմուծվող
// տվյալների
int main(){
    const int n=15;
    el<char> a[ n] ;
    for(int i=0;i<n;i++) { // ծառի գագաթի
                           // ներմուծում
        cin>>a[ i] .data;
        cin>>a[ i] .mark;
    }
    tree <char> t(a,n); // ծառի ստեղծում և արտածում
    return 0;
}
```

Խնդիր 5.14. *Գտնել և տպել տրված ոչ դատարկ ծառի յուրաքանչյուր մակարդակում պարունակվող հանգույցների քանակը:*

```
// իտերատիվ եղանակ
#include <iostream>
#include <queue>    // օգտագործվում է STL-ի queue դասը
using namespace std;
#include "TreeNode.h"
//TreeNode տիպի սահմանումը տես խնդիր5.8-ում
// ֆունկցիան իտերատիվ եղանակով արտածում է root
// ցուցիչով որոշվող ծառի
// հանգույցների քանակը ըստ մակարդակների
void nodeCountOnLevels(TreeNode *root){
    TreeNode *t=root;
    // ծառի հանգույցների ցուցիչների հերթ
    queue<TreeNode *> x;
    int level=0;  // մակարդակը նշող փոփոխական

    int k=1; // ընթացիկ մակարդակը ոասգույցների քանակը
    int k1; // հաջորդ մակարդակի հանգույցների քանակը
    x.push(t);
    while(!x.empty()) {
        k1=0;
        cout<<"Level "<<level<<"
        has- "<<k<<" nodes"<<endl;
        for(int i=0;i<k;i++) {
            t=x.front();
            x.pop();
            if(t->left) {
```

```
                x.push(t->left);
                k1++;
            }
            if(t->right) {
                x.push(t->right);
                k1++;
            }
        }
        level++;
        k=k1;
    }
}

// nodeCountOnLevels() ֆունկցիայի կիրառման օրինակ
int main() {
    TreeNode *d=new TreeNode(4);
    TreeNode *e=new TreeNode(5);
    TreeNode *g=new TreeNode(6);
    TreeNode *b=new TreeNode(2,d,e);
    TreeNode *c=new TreeNode(3,NULL,g);
    TreeNode *a=new TreeNode(1,b,c);
    nodeCountOnLevels(a);
    return 0;
}

// ռեկուրսիվ եղանակ
#include <iostream>
using namespace std;
class TreeNode;
//TreeNode տիպի սահմանումը բերված է ենթիր 5.8-ում:
// ֆունկցիան ռեկուրսիվ եղանակով որոշում է root
// ցուցիչով ծառի հանգույցների քանակը k-րդ մակարդակում
int nodeCountOnLevel(TreeNode* t,int k) {
    if(!t)
        return 0;
    if(!k)
        return 1;
    else
        return nodeCountOnLevel(t->left,k-1)+
             nodeCountOnLevel(t->right,k-1);
}
```

```
// nodeCountOnLevel() ֆունկցիայի կիրառման օրինակ
int main() {
    //ծառի կառուցում
    TreeNode *f=new TreeNode(4);
    TreeNode *d=new TreeNode(4,f);
    TreeNode *e=new TreeNode(5);
    TreeNode *g=new TreeNode(6);
    TreeNode *b=new TreeNode(2,d,e);
    TreeNode *c=new TreeNode(3,NULL,g);
    TreeNode *a=new TreeNode(1,b,c);
// ծառի 0-ից մինչև 5-րդ մակարդակների գագաթների
// քանակների արտածում
    for(int k=0;k<5;k++)
    cout<<"Level"<<k<<"has"<<nodeCountOnLevel(a,k)
                           <<"nodes"<<endl;
return 0;
}
```

Խնդիր 5.20. *Գտնել տրված բինար ծառի երկու որդի ունեցող հանգույցների քանակը:*

```
// իտերատիվ եղանակ
#include <iostream>
#include <stack>  // օգտագործված է STL-ի պահուցակը
using namespace std;
// ծառի երկու որդի ունեցող գագաթների քանակը
int twoSonsNodes=0;
// TreeNode տիպի սահմանումը բերված է խնդիր 5.8-ում
class TreeNode;
void twoSons(TreeNode *p);
// ֆունկցիայի սահմանումը բերված է խնդիր 5.8-ում
// ֆունկցիան մշակում է ծառի գագաթը
void preorder(TreeNode *t,void visit(TreeNode *));
void twoSons(TreeNode *p) {
    if (p->left&&p->right)
        twoSonsNodes++;
}
```

```
// իրականացնելով ծառի ուղիղ շրջանցում,
// հաշվվում է երկու որդի ունեցող գագաթների քանակը
int main() {
    //ծառի կառուցում
    TreeNode *d=new TreeNode(4);
    TreeNode *e=new TreeNode(5);
    TreeNode *g=new TreeNode(6);
    TreeNode *b=new TreeNode(2,d,e);
    TreeNode *c=new TreeNode(3,NULL,g);
    TreeNode *a=new TreeNode(1,b,c);
    preorder(a,twoSons);
    //երկու որդի ունեցող գագաթների քանակի հաշվում
    cout<<twoSonsNodes;
    return 0;
}
// ռեկուրսիվ եղանակ
int twoSonsRec(TreeNode *p) {
    if(!p)
        return 0;
    if (!p->left&&!p->right)
        return 0;
    if (p->left&&!p->right)
        return twoSonsRec(p->left);
    if (!p->left&&p->right)
        return twoSonsRec(p->right);
    if (p->left&&p->right)
        return 1+twoSonsRec(p->left)+
                +twoSonsRec(p->right);
}
```

Խնդիր 5.66. *Ստեղնաշարից կարդալ n հատ ամբողջ թիվ (թվերը կարող են կրկնվել), ստեղծել որոնման բինար ծառ՝ յուրաքանչյուր գագաթում հիշելով ներմուծված իրարից տարբեր արժեքներից մեկը և նրա կրկնումների քանակը: Ստացված ծառը տպել 90 աստիճանով շրջված դեպի ձախ:*

Ծրագրում օգտագործված է տարրի փնտրման և ավելացման search(int x, Node *p) ֆունկցիան, որը տված p արմատով ծառում x թիվը պարունակող գագաթ հանդիպելիս այդ գագաթում ավելացնում

է x թվի կրկնումների քանակը: Եթե այդպիսի գագաթ չի հայտնաբերվում, ապա ծառում ավելացվում է նշված արժեքով նոր գագաթ:

```
#include <iostream>
using namespace std;
const int indent = 6;
struct Node {
    int      data;
    int      count;
    Node     *left;
    Node     *right;
};

// Տպում է ծառը 90^0 շրջված դեպի ձախ
void Printtree(Node *t, int level) {
    int i;
    if (t!=0) {
        Printtree(t->right, level + 1);
        for (i=1; i<indent*level; i++)
            cout <<' ';
        cout <<t->data<<endl;
        Printtree(t->left, level + 1);
    }
}

// ֆունկցիան p ցուցիչով ծառում որոնում է տրված արժեքը
// արժեքը չգտնելու դեպքում այն ավելացնում է ծառին,
// գտնելու դեպքում ավելացվում է այդ արժեքի
// կրկնումների քանակը
void search(int x, Node *&p){
    if (p == NULL) {
          p = new Node;
          p->data=x;
          p->count=1;
          p->left=NULL;
          p->right=NULL;
    }
    else  if(x<p->data)
          search(x, p->left);
    else  if(x>p->data)
          search(x, p->right);
    else
          p->count++;
}
```

```
// ծրագրում իրականացվում է
// որոնման ծառի արժեքների ներածում ստեղնաշարից,
// որոնման ծառի կառուցում և արտածում
int main() {
    int n;
    int key;
    Node *root = NULL;
    cin >> n;
    while(n > 0) {
        cin>>key;
       // ներմուծված արժեքի ավելացում ծառից
       search(key, root);
        n--;
    }
    Printtree(root, 0); // ծառի արտածում
    return 0;
}
```

Խնդիր 5.83. Կառուցել Phonebook դասը՝ հեռախոսահամարների գրքույկ մոդելավորելու համար: Գրքույկը կազմված է Person դասի օբյեկտներից, որոնք պարունակում են ազգանուն, անուն, հայրանուն, հեռախոսահամար: Գրքույկը ներկայացնել ըստ ազգանունների կարգավորված որոնման բինար ծառի տեսքով: Նախատեսել ըստ ազգանվան հեռախոսահամարի որոնման, ավելացման և հեռացման գործողությունները:

```
// Person տիպի հայտարարում
class Person {
private:
    char* firstname;
    char* lastname;
    int phoneNumber;
public:
    Person(char*,char*,int);
    ~Person();};

//ծառի հանգույցի պարամետրացված տիպի հայտարարում
template <class K_type,class KeyedItem >
class TreeNode{
```

```
public:
        // հանգույցի արժեքը՝ բանալի և արժեք
        KeyedItem data;
        // ձախ որդու ցուցիչ
        TreeNode<K_type, KeyedItem >* left;
        // աջ որդու ցուցիչ
        TreeNode<K_type, KeyedItem >* right;
};

// ֆունկցիայի ցուցիչի հայտարարում: ֆունկցիան
// կիրառվում է ծառի հանգույցը մշակելու համար
    template <class T>
    typedef void (*func)(T);

// բինար ծառի պարամետրացված տիպի հայտարարում
template <class K_type,class class KeyedItem >
class BinSearchTree {
public:
    BinSearchTree();
    BinSearchTree(const BinSearchTree&);
    ~BinSearchTree();
    bool empty();
    void preorderTraverse(func); // ուղիղ շրջանցում
    void inorderTraverse(func); //սիմետրիկ շրջանցում
    void postorderTraverse(func);//հակադարձ շրջանցում
    bool find(K_type); //փնտրում տրված բանալիով
    void insert(K_type,KeyedItem);
    void remove(K_type);
private:
    TreeNode<K_type, KeyedItem>* root;
    // դասի փակ անդամ-ֆունկցիաներ, որոնք
    // օգտագործվում են դասի բաց ֆունկցիաների
    // ռեկուրսիվ տարբերակները իրականացնելու համար
    void preorder(TreeNode<K_type,
                  KeyedItem >*,func);
    void inorder(TreeNode<K_type,
                  KeyedItem >*,func);
    void postorder(TreeNode<K_type,
                  KeyedItem >*,func);
```

```
    bool find(TreeNode<K_type, KeyedItem >*,K_type);
    void insert(TreeNode<K_type, KeyedItem >*,
                    K_type, KeyedItem);
    void remove(TreeNode<K_type,
                    KeyedItem >*,K_type);

};
// Phonebook պարամետրացված տիպի հայտարարում
template <class K_type,class KeyedItem >
class Phonebook {
public:
    BinSearchTree<K_type, KeyedItem > book;
};
// հեռախոսահամարների գրքույկի կառուցման և օգտագործման
// օրինակ
int main() {
    Phonebook<char*,Person> myPhonebook;
      // հեռախոսահամարների գրքույկի կառուցում
      Phonebook<char*,Person> myPhonebook;
      Person P("Petrosyan","Petros",254876);
      myPhonebook.book.insert("Petrosyan",P);
      // փնտրում ըստ տրված բանալու
      myPhonebook.book.find("Petrosyan");
      //տրված բանալիով արժեքի հեռացում
      myPhonebook.book.remove("Petrosyan");
      return 0;}
```

ԳՐԱԿԱՆՈՒԹՅՈՒՆ

1. **Топп У, Форд У.** "Структуры данных в C++, М., Бином, 2000 г.
2. **Кнут Д.** "Искусство программирования" том 1, "Основные алгоритмы", Изд-во "Вильямс", М., 2000г.
3. **Ахо А., Хопкрофт Д., Ульман Дж.** "Структуры данных и алгоритмы". Изд-во "Вильямс", М., 2000г.
4. **Вирт Н.** "Структура данных + алгоритмы = программы", Изд-во "Мир", М., 1988 г.
5. **Каррано Ф., Причард Дж.** "Абстракция данных и решение задач на C++. Стены и зеркала." Изд-во "Вильямс", М., 2003г.
6. **Лэнгсам Й., Огенстайн М., Тенненбаум А.** "Структуры данных для персональных ЭВМ". Изд-во "Мир", М., 1989г.
7. **Майкл Мейн, Уолтер Савитч** "Структуры данных и другие обьекты в C++" Изд-во "Вильямс", М., 2002 г.

ԲՈՎԱՆԴԱԿՈՒԹՅՈՒՆ

Ս.Գ. ՍԱՐԳՍՅԱՆ, Ա.Ս. ՀՈՎԱԿԻՄՅԱՆ, Կ.Ս. ԴԱՐԲԻՆՅԱՆ,
Ս.Ա. ԱՎԵՏԻՍՅԱՆ, Ն.Հ. ԻՍՊԻՐՅԱՆ

ՏՎՅԱԼՆԵՐԻ ԿԱՌՈՒՑՎԱԾՔՆԵՐ

ՈՒՍՈՒՄՆԱԿԱՆ ՁԵՌՆԱՐԿ

Տեխ. խմբագիր՝ Վ.Զ. ԲԴՈՅԱՆ
Համակարգչային ձևավորող՝ Ն.Օ. ԽՆԿԻԿՅԱՆ

Ստորագրված է տպագրության 02.06.2010 թ.:
Չափսը՝ 60x84$^{1}/_{16}$: Թուղթը՝ օֆսեթ: Հրատ. 9.7 մամուլ,
տպագր. 11.75 մամուլ= 10.9 պայմ. մամուլի:
Տպաքանակ՝ 200: Պատվեր՝ 61:

ԵՊՀ հրատարակչություն, Երևան, Ալ. Մանուկյան 1:

Երևանի պետական համալսարանի
օպերատիվ պոլիգրաֆիայի ստորաբաժանում
Երևան, Ալ. Մանուկյան 1: