Ռուս-հայկական համալսարան
Մաթեմատիկայի, ֆիզիկայի և բարձր տեխնոլոգիաների ինստիտուտ

Մարիամ Հարությունյան, Հայկ Ասլանյան

«Տվյալների կառուցվածքներ և օբյեկտակողմնորոշված ծրագրավորում»

Գործնական պարապմունքների ձեռնարկ

Էլեկտրոնային ֆորմատ


HOMO RES SACRA

РОССИЙСКО-АРМЯНСКИЙ УНИВЕРСИТЕТ

ИНСТИТУТ МАТЕМАТИКИ, ФИЗИКИ И ВЫСОКИХ ТЕХНОЛОГИЙ

МАРИАМ АРУТЮНЯН
АЙК АСЛАНЯН

СТРУКТУРЫ ДАННЫХ И ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Руководство для практических занятий

Ереван
Издательство РАУ
2026

Электронный формат

ՌՈՒՍ-ՀԱՅԿԱԿԱՆ ՀԱՄԱԼՍԱՐԱՆ

ՄԱԹԵՄԱՏԻԿԱՅԻ, ՖԻԶԻԿԱՅԻ ԵՎ ԲԱՐՁՐ
ՏԵԽՆՈԼՈԳԻԱՆԵՐԻ ԻՆՍՏԻՏՈՒՏ

ՄԱՐԻԱՄ ՀԱՐՈՒԹՅՈՒՆՅԱՆ
ՀԱՅԿ ԱՍԼԱՆՅԱՆ

ՏՎՅԱԼՆԵՐԻ ԿԱՌՈՒՑՎԱԾՔՆԵՐ ԵՎ ՕԲՅԵԿՏԱԿՈՂՄՆՈՐՈՇՎԱԾ ԾՐԱԳՐԱՎՈՐՈՒՄ

Գործնական պարապմունքների ձեռնարկ

Երևան
ՌՀՀ հրատարակչություն
2026

Էլեկտրոնային ֆորմատ

ՀՏԴ 004(07)
ԳՄԴ 32.97գ7
Հ 422

Հրատարակման է երաշխավորվում
Ռուս-հայկական համալսարանի Գիտատեխնիկական և
Խմբագրա-Հրատարակչական խորհրդի կողմից

Պատասխանատու խմբագիր՝ տ.գ.թ. **Ջ.Ա. Հակոբյան**
Գրախոսներ՝ ՌՀՀ Համակարգային ծրագրավորման ամբիոնի վարիչ,
տեխ.գիտ.դոկտոր, դոցենտ **Ս.Ս. Սարգսյան,**
ՈԱԱԿ տնօրեն, տեխ.գիտ.թեկնածու, դոցենտ **Ռ.Վ. Թոփչյան**

Հարությունյան Մ., Ասլանյան Հ.
Տվյալների կառուցվածքներ և օբյեկտա-կողմնորոշված
Հ 422 ծրագրավորում: Գործնական պարապմունքների ձեռնարկ /
Մ. Հարությունյան, Հ. Ասլանյան. – Եր.: ՌՀՀ, 2026. – 156 էջ:

Սույն ձեռնարկը նախատեսված է «Տվյալների կառուցվածքներ և օբյեկտակողմնորոշված ծրագրավորում» դասընթացի գործնական պարապմունքների կազմակերպման և անցկացման համար: Այն ընդգրկում է տվյալների կառուցվածքների և ալգորիթմների հիմնական թեմաները՝ C++ ծրագրավորման լեզվի կիրառմամբ:

Ձեռնարկում ներկայացված նյութը կառուցված է տրամաբանական հաջորդականությամբ և ներառում է յուրաքանչյուր թեմայի նպատակները, մեթոդական ցուցումները, ինչպես նաև տարբեր բարդության աստիճանի առաջադրանքներ՝ նախատեսված ինչպես լսարանային աշխատանքի, այնպես էլ ինքնուրույն ուսումնառության համար:

ՀՏԴ 004(07)
ԳՄԴ 32.97գ7

ISBN 978-9939-67-418-6

Բովանդակություն

ԳԼՈՒԽ 3. ՈՉ ԳԾԱՅԻՆ ՏՎՅԱԼՆԵՐԻ ԿԱՌՈՒՑՎԱԾՔՆԵՐ

ՆԱԽԱԲԱՆ

Ժամանակակից ծրագրավորման ոլորտում արդյունավետ և մասշտաբավորվող համակարգերի մշակման կարևորագույն նախապայմաններից են տվյալների կառուցվածքների խորքային իմացությունը և օբյեկտա-կողմնորոշված մտածելակերպի ձևավորումը: Տվյալների կառուցվածքները և ալգորիթմները հանդիսանում են ծրագրային համակարգերի հիմքը՝ ապահովելով տվյալների արդյունավետ կազմակերպում, մշակում և պահպանություն, իսկ օբյեկտակողմնորոշված մոտեցումը հնարավորություն է տալիս կառուցել ճկուն, ընդլայնելի և վերօգտագործելի ծրագրային լուծումներ:

Սույն ձեռնարկը նպատակ ունի աջակցություն ցուցաբերել դասախոսներին՝ «Տվյալների կառուցվածքներ և օբյեկտակողմնորոշված ծրագրավորում» դասընթացի շրջանակներում ուսանողների գործնական հմտությունների համակարգված ձևավորման գործընթացում՝ C++ ծրագրավորման լեզվի կիրառմամբ: Այն ուղղված է ոչ միայն տեսական գիտելիքների փոխանցմանը, այլև դրանց նպատակային կիրառման, խնդիրների վերլուծության և արդյու-նավետ լուծումների մշակման հմտությունների ձևավորմանը:

Ձեռնարկը մշակվել է Ռուս-հայկական համալսարանի Մաթեմատիկայի, ֆիզիկայի և բարձր տեխնո-

լոգիաների ինստիտուտում իրականացվող համապատասխան դասընթացի գործնական պարապմունքների ապահովման նպատակով: Այն հիմնված է հետևյալ սկզբունքի վրա, համաձայն որի տեսական գիտելիքների արդյունավետ յուրացումը պայմանավորված է դրանց նպատակային և հետևողական գործնական կիրառմամբ:

Ձեռնարկում ներառված նյութերը կառուցված են թեմատիկ և տրամաբանական հաջորդականությամբ՝ ընդգրկելով տվյալների կառուցվածքների և ալգորիթմների հիմնական բաժինները: Յուրաքանչյուր թեմա ուղեկցվում է հստակ ձևակերպված նպատակներով, մեթոդական ցուցումներով և գործնական առաջադրանքներով, որոնք կարող են կիրառվել ինչպես լսարանային աշխատանքի, այնպես էլ ուսանողների ինքնուսուցման կազմակերպման համար:

Առանձնահատուկ ուշադրություն է դարձվում ոչ միայն ալգորիթմների և տվյալների կառուցվածքների կոռեկտ իրագործմանը, այլև դրանց արդյունավետության գնահատմանը, հիշողության կառավարման սկզբունքներին և ծրագրային կոդի որակական բնութագրիչներին: Ձեռնարկը միտված է աջակցել դասախոսներին ուսանողների մոտ համակարգային և ալգորիթմական մտածողության ձևավորմանը, ինչպես նաև նպաստել նրանց մասնագիտական կարողությունների զարգացմանը:

ԳԼՈՒԽ 1.
C++ ԼԵԶՎԻ ՀԻՄՈՒՆՔՆԵՐ՝ ՏՎՅԱԼՆԵՐԻ ԿԱՌՈՒՑՎԱԾՔՆԵՐԻ ԻՐԱԿԱՆԱՑՄԱՆ ՀԱՄԱՐ

Այս գլխում ներկայացվում են C++ լեզվի հիմնական գաղափարները, որոնք անհրաժեշտ են տվյալների կառուցվածքների ուսումնասիրման համար: Քննարկվում են հիշողության կառավարման հիմունքները, ցուցիչներն ու հղումները, ինչպես նաև ժամանակակից C++ լեզվի կաղապարային (template) մեխանիզմները:

Արդյունքում ուսանողը ձեռք է բերում հիշողության կառավարման և C++ լեզվի հիմնական աբստրակցիաների կիրառման հիմնարար հմտություններ: Այս գաղափարները կիրառվում են կապակցված ցուցակների, ծառերի, հեշ աղյուսակների և այլ կառուցվածքների իրականացման ժամանակ:

ԴԱՍ 1. ՑՈՒՑԻՉՆԵՐ, ՀՂՈՒՄՆԵՐ, ՀԻՇՈՂՈՒԹՅԱՆ ԿԱՌԱՎԱՐՈՒՄ C++ ԼԵԶՎՈՒՄ

Տևողություն՝ 90 րոպե

Թեմա՝ Ցուցիչներ (pointers), հղումներ (references), պահունակ և կույտ (stack vs heap), հիշողության հատկացում և ազատում (new/delete):

1. Դասի նպատակը

Ուսանողների մոտ ձևավորել ցուցիչների և հիշողության ձեռքով կառավարման (manual memory management) հմտություններ: Այս գիտելիքները հիմնարար նշանակություն ունեն հաջորդ դասերին բարդ տվյալների կառուցվածքներ (օրինակ՝ կապակցված ցուցակներ, ծառեր) իրականացնելու համար:

2. Մեթոդական ցուցումներ

- ✓ **Պատկերում:** Գրատախտակին գծել հիշողության բջիջներ՝ ցույց տալու համար, որ ցուցիչը պահում է հասցե (օրինակ՝ 0x7ffee...), իսկ հղումը պարզապես նույն բջջի այլ անունն է (alias):
- ✓ **Հաճախ հանդիպող սխալներ:** Հատուկ ուշադրություն դարձնել «կախված ցուցիչի» (dangling pointer) խնդրին, օրինակ, երբ ֆունկցիան վերադարձնում է լոկալ փոփոխականի հասցեն:
- ✓ **Հիմնական կանոն:** Յուրաքանչյուր new օպերատորի կանչ պետք է ունենա համապատասխան delete օպերատորի կանչ:

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Հեշտ) - Տեղափոխման ֆունկցիա

Իրականացնել swap ֆունկցիայի երեք տարբերակ և բացատրել դրանց աշխատանքի սկզբունքը.

1. void swap(int a, int b) - Ինչու՞ սա չի փոխում սկզբնական փոփոխականների արժեքները (Արժեքով փոխանցում - Pass by value):
2. void swap(int* a, int* b) - Ցուցիչով փոխանցում - Pass by pointer:
3. void swap(int& a, int& b) - Հղումով փոխանցում - Pass by reference:

Խնդիր 2. (Հեշտ) - Ցուցիչի հիմունքներ

Հայտարարել int փոփոխական, ցուցիչ դեպի այն, և կատարել հետևյալը.

- ✓ արտածել փոփոխականի արժեքը, հասցեն, ցուցիչի արժեքը և ցուցիչի հասցեն,
- ✓ ցուցիչի միջոցով փոփոխել փոփոխականի արժեքը և կրկին արտածել:

Նպատակը՝ ամրապնդել & (հասցեի) և * (ապահասցեավորման) օպերատորների հասկացողությունը:

Խնդիր 3. (Միջին) - Ցուցիչների թվաբանություն

Տրված է ամբողջ թվերի զանգված՝ int arr[] = {10, 20, 30, 40, 50}: Ստեղծել ցուցիչ ptr, որը ցույց է տալիս զանգվածի սկզբին: Առանց ինդեքսավորում (arr[i]) օգտագործելու, միայն ptr++ և *ptr (ապահասցեավորում) գործողություններով արտածել զանգվածի բոլոր տարրերը: Բացատրել, թե ինչու է ptr + 1 գործողությունը հասցեն մեծացնում է 4 բայթով (եթե տվյալ համակարգում int-ը 4 բայթ է), և ոչ թե 1-ով:

Խնդիր 4. (Միջին) - Դինամիկ զանգվածի հատկացում

Իրականացնել int* createArray(int n) ֆունկցիան, որը կույտում ստեղծում է n չափանի ամբողջ թվերի զանգված, լրացնում է այն 1-ից մինչև n թվերով և վերադարձնում է զանգվածի ցուցիչը: main-ում կանչել ֆունկցիան, արտածել ստացված զանգվածի տարրերը և ազատել հիշողությունը (delete[]):

Խնդիր 5. (Միջին) - Անվավեր ցուցիչ

Հետևյալ կոդում getNumber() ֆունկցիան վերադարձնում է լոկալ փոփոխականի հասցեն:

```
int* getNumber() {
  int x = 42;
  return &x;
}
```

- ✓ Բացատրել թե ինչու է սա վտանգավոր. ֆունկցիան ավարտվելուց հետո x-ը ջնջվում է պահունակից, բայց ցուցիչը դեռ պահում է այդ հասցեն:
- ✓ Ուղղել կոդը՝ new-ի օգնությամբ x-ը կույտում ստեղծելով, որպեսզի օբյեկտը շարունակի գոյություն ունենալ ֆունկցիայի ավարտից հետո:
- ✓ Չմոռանալ main-ում ազատել հիշողությունը (delete):

4. Տնային հանձնարարություն

Խնդիր 6. (Միջին) - Չափի կրկնապատկում

Իրականացնել ֆունկցիա, որը ստանում է դինամիկ զանգված, կրկնապատկում է դրա չափը՝ պահպանելով հին տվյալները։

Քայլեր՝ նոր հիշողության հատկացում -> պատճենում -> հին հիշողության ազատում -> վերահասցեավորում (reassign)։

Խնդիր 7. (Բարդ) - Երկչափ դինամիկ զանգված

Իրականացնել ֆունկցիաներ, որոնք՝

- ստեղծում են N x M չափի դինամիկ երկչափ զանգված (օգտագործել կրկնակի ցուցիչ՝ int**),
- լրացնում են այն արժեքներով,
- արտածում են մատրիցի ձևով,
- ճիշտ ազատում են հիշողությունը (նախ տողերը, հետո հիմնական ցուցիչը)։

Բացատրել, թե ինչու է հիշողության ազատման հերթականությունը կարևոր։

5. Լրացուցիչ առաջադրանք

Խնդիր 8. (Միջին) - Հաստատուն ցուցիչներ

Ցույց տալ հետևյալ չորս դեպքերի տարբերությունը.

```
int* ptr // սովորական ցուցիչ
const int* ptr // հաստատուն արժեքի ցուցիչ
int* const ptr // հաստատուն ցուցիչ
const int* const ptr // երկուսն էլ հաստատուն
```

Յուրաքանչյուր դեպքի համար փորձել փոփոխել և՛ ցուցիչը, և՛ արժեքը, բացատրել կոմպիլյատորի սխալները։

ԴԱՍ 2. ԸՆԴՀԱՆՐԱՑՎԱԾ ԾՐԱԳՐԱՎՈՐՈՒՄ (ԿԱՂԱՊԱՐՆԵՐ)

Տևողություն՝ 90 րոպե

Թեմա՝ Ֆունկցիաների և դասերի կաղապարներ, կաղապարների մասնագիտացում։

1. Դասի նպատակը

Ուսանողը պետք է հասկանա, թե ինչպես գրել կաղապարային (template) ֆունկցիաներ և դասեր, որոնք կարող են աշխատել տարբեր տիպերի հետ։ Պետք է հասկանա, որ կոմպիլյատորը ստեղծում է կաղապարի կոնկրետ իրականացումը կոմպիլյացիայի ժամանակ, երբ օգտագործվում է կոնկրետ տիպով։ Պետք է նաև կարողանա տարբերակել՝ երբ է օգտագործվում ընդհանուր կաղապարը, և երբ՝ հատուկ տարբերակը (մասնագիտացումը)։ Պետք է հասկանա նաև, որ յուրաքանչյուր կաղապար որոշակի պահանջներ է դնում օգտագործվող տիպի վրա։

2. Մեթոդական ցուցումներ

- ✓ **Կոնկրետացում (Instantiation):** Կոմպիլյատորը ստեղծում է կաղապարի կոնկրետ իրականացումը, երբ այն օգտագործվում է որոշակի տի-

պով: Դա կարող է տեղի ունենալ ինչպես ֆունկցիայի կանչի ժամանակ, այնպես էլ այն դեպքում, երբ տիպը ավտոմատ որոշվում է:

- ✓ **Ոչ տիպային պարամետրեր:** Կաղապարը կարող է ընդունել նաև արժեքներ (օրինակ՝ template <typename T, int N>), ինչը հարմար է ֆիքսված չափի զանգվածների համար:
- ✓ **Մասնագիտացում (Specialization):** Երբեմն որոշ տիպերի համար (օրինակ՝ char* կամ bool) ընդհանուր իրականացումը չի ապահովում ցանկալի վարքագիծ, և մեզ պետք է տրամադրել մասնագիտացված իրականացում տվյալ տիպի համար:
- ✓ **Պահանջներ:** Կաղապարները պահանջում են, որ տիպը ունենա որոշ գործողություններ (օրինակ՝ + կամ ==): Եթե տիպը չունի այդ գործողությունները, ծրագիրը չի կոմպիլացվի: Այս սխալները երևում են կոմպիլյացիայի ժամանակ, ոչ թե ծրագրի աշխատանքի ընթացքում:

3. Առաջադրանքներ լսարանային աշխատանքի համար

Մաս 1. Ֆունկցիաների կաղապարներ (Function Templates)

Խնդիր 1. (Հեշտ) - Բազային գործիքներ

1. Իրականացնել printElement<T>(T value) ֆունկցիան, որը էկրանին արտածում է value-ի արժեքը:

2. Իրականացնել mySwap<T>(T& a, T& b) ֆունկցիան, որը փոխում է երկու փոփոխականների արժեքները:

Խնդիր 2. (Միջին) - Որոնում և Գումարում

Իրականացնել sumArray<T>(T* arr, int size), որը վերադարձնում է զանգվածի տարրերի գումարը:

Նշում՝ այս ֆունկցիան աշխատում է միայն այն տիպերի համար, որոնց օբյեկտները կարելի է գումարել:

Մաս 2. Դասերի կաղապարներ (Class Templates)

Խնդիր 3. (Միջին) - Զույգ

Ստեղծել Pair<T1, T2> դասը, որը պահում է տարբեր տիպի երկու արժեք:

Խնդիր 4. (Բարդ) - Մատրից

Ստեղծել Matrix<T, N, M> կաղապարային դասը N տողով և M սյունակով մատրիցների հետ աշխատելու համար:

Իրականացնել հետևյալ մեթոդները՝

- ✓ set(int row, int col, T value)՝ տարրի արժեքը սահմանելու համար,
- ✓ get(int row, int col)՝ տարրը ստանալու համար,
- ✓ print()՝ մատրիցը ֆորմատավորված արտածման համար,
- ✓ fill(T value)՝ մատրիցը նույն արժեքով լցնելու համար,
- ✓ transpose()՝ վերադարձնում է նոր մատրից, որտեղ տողերն ու սյունակները տեղերով փոխված են (եթե սկզբում N×M էր, դառնում է M×N),

- ✓ isSquare()՝ ստուգելու՝ մատրիցը քառակուսի է, թե ոչ,
- ✓ trace()՝ գլխավոր անկյունագծի տարրերի գումարը։ Աշխատում է միայն քառակուսի մատրիցների համար։

Գերբեռնել հետևյալ օպերատորները՝

- ✓ operator+՝ նույն չափի երկու մատրիցների գումարման համար *(C[i][j] = A[i][j] + B[i][j])*,
- ✓ operator==՝ երկու մատրիցների հավասարությունը ստուգելու համար,
- ✓ operator*՝ մատրիցների բազմապատկման համար (համապատասխան չափերի դեպքում)։ Բազմապատկումը հնարավոր է միայն, եթե առաջին մատրիցի սյունակների քանակը հավասար է երկրորդի տողերի քանակին։ Արդյունքում ստացվում է նոր մատրից համապատասխան չափերով։

Նշումներ՝

- ✓ Մատրիցների բազմապատկման հաշվարկային բարդությունը O(N³) է։
- ✓ Անհրաժեշտ է ապահովել ինդեքսների սահմանների ստուգում (set, get ֆունկցիաներում)։

Մաս 3. Մասնագիտացում (Specialization)

Խնդիր 5. (Բարդ) - Տողերի համեմատություն

Ստեղծել isEqual<T>(T a, T b) կաղապարը։ Լռությամբ այն պետք է օգտագործի == օպերատորը։ Կա-

տարել մասնագիտացում const char* տիպի համար, որպեսզի այն համեմատի տողերի բովանդակությունը (strcmp), այլ ոչ թե հասցեները:

Նշում՝ char* տիպի դեպքում ==-ը համեմատում է հասցեները, ոչ թե տեքստը, դրա համար պետք է համեմատել բովանդակությունը:

4. Տնային հանձնարարություն

Խնդիր 6. (Հեշտ) - SmartArray և բացառություններ

Իրականացնել SmartArray<T> կաղապարային դաս, որը օգտագործում է դինամիկ հիշողություն զանգված պահելու համար: Ավելացնել at(index) մեթոդ, որը վերադարձնում է համապատասխան տարրը: Եթե index-ը դուրս է թույլատրելի սահմաններից, մեթոդը պետք է նետի std::out_of_range բացառություն: Պետք է ապահովել, որ հիշողությունը ճիշտ ազատվի (ունենա փլուզիչ): Նաև ապահովել, որ պատճենելիս խնդիրներ չլինեն (պատճենման մեխանիզմներ):

Խնդիր 7. (Միջին) - Միջակայք

Ստեղծել դաս, որը սահմանում է միջակայք [start, end] և ունի contains(value) մեթոդը: Փորձարկել int, double և char տիպերի համար:

Նշում՝ այս դասը աշխատում է այն տիպերի համար, որոնք կարելի է համեմատել (<, >):

Խնդիր 8. (Միջին) - Հատուկ արտածում

Ստեղծել ֆունկցիայի կաղապար printValue<T>, որը bool-ի դեպքում արտածում է "true" կամ "false"

(տեքստային), իսկ char*-ի դեպքում արտածում է տողը չակերտների մեջ՝ "<<string>>":

5. Լրացուցիչ առաջադրանքներ

Խնդիր 9. (Բարդ) - Զանգվածի ֆիլտրում

Ստեղծել filter ֆունկցիա-կաղապարը, որը ստանում է զանգված (T* arr) և դրա չափը: Որպես արգումենտ ընդունում է նաև պրեդիկատ (ֆունկցիա, որը վերադարձնում է bool և ստանում է const T&): Վերադարձնում է նոր դինամիկ զանգված, որը պարունակում է միայն այն տարրերը, որոնց համար պրեդիկատը true է:

Նշում՝ նոր ստեղծված զանգվածի հիշողությունը պետք է ազատի օգտագործողը:

Խնդիր 10. (Բարդ) - Խորը պատճենում

Իրականացնել copyValue<T> կաղապարային ֆունկցիա, որը սովորական տիպերի (int, double և այլն) դեպքում վերադարձնում է փոխանցված արժեքի պատճենը, իսկ ցուցիչների (T*) դեպքում՝ կատարում է խորը պատճենում. ստեղծում է նոր հիշողություն, պատճենում այնտեղ միայն մեկ արժեք (այսինքն՝ *ptr) և վերադարձնում նոր ցուցիչը:

Նշում՝ ֆունկցիան չի աշխատում զանգվածների համար, քանի որ ցուցիչից հնարավոր չէ որոշել տարրերի քանակը: Վերադարձված ցուցիչի հիշողությունը պետք է ազատվի օգտագործող կոդի կողմից:

Խնդիր 11. (Միջին) - Նույնական տիպերի զույգ

Ստեղծել Pair<T1, T2> կաղապարային դասը երկու արժեք պահելու համար։ Իրականացնել մասնակի մասնագիտացում (Partial Specialization) Pair<T, T> դեպքի համար (երբ երկու տիպերն էլ նույնն են)։ Այս դեպքում print() մեթոդը պետք է սովորական արտածելու փոխարեն տպի հատուկ հաղորդագրություն. *"Pair of identical types: [արժեքները]"*։

Նշում՝ եթե կաղապարը օգտագործվում է տիպով, որը չի բավարարում պահանջներին (օրինակ՝ չունի լռությամբ կառուցիչ կամ անհրաժեշտ օպերատորներ), կոմպիլյացիայի ժամանակ առաջանում է սխալ։ Սա կաղապարների կարևոր առանձնահատկություններից է։

ԴԱՍ 3. STD::STRING, ԿԻՐԱՌՈՒԹՅՈՒՆՆԵՐ

Տևողություն՝ 90 րոպե

Թեմա՝ std::string դասի խորացված ուսումնասիրություն։

1. Դասի նպատակը

Ուսանողը պետք է ուսումնասիրի std::string դասի ներքին կառուցվածքը և ռեսուրսների կառավարման սկզբունքները՝ որպես դինամիկ հիշողության կառավարման օրինակ։ Գործնական խնդիրների միջոցով ձեռք կբերի տողերի մշակման հիմնական հմտություն-

ներ: Ինչպես նաև հասկանա std::string-ի հիմնական գործողությունների ժամանակային բարդությունները (օրինակ՝ ինդեքսով սիմվոլի դիմում՝ O(1), կցում (append)՝ ամորտիզացված O(1), տեղադրում (insert) / հեռացում (erase)՝ O(N)):

2. Մեթոդական ցուցումներ

- ✓ **Տողի կառուցվածք:** std::string-ը սիմվոլների դինամիկ հաջորդականություն է, որը ավտոմատ կառավարում է հիշողությունը:
- ✓ **Small String Optimization (SSO)**: Կարևոր է հասկանալ SSO մեխանիզմը, որի դեպքում կարճ տողերը պահվում են օբյեկտի ներսում՝ առանց դինամիկ հիշողության հատկացման: Սա թույլ է տալիս խուսափել դինամիկ հիշողության օգտագործումից փոքր տողերի դեպքում և արագացնում է աշխատանքը: SSO-ի իրականացումը կախված է կոնկրետ կոմպիլյատորից և ստանդարտ գրադարանի իրականացումից:
- ✓ **size և capacity**: Պետք է հստակ տարբերել size և capacity պարամետրերը. size-ը ներկայացնում է իրական երկարությունը, իսկ capacity-ն՝ հատկացված հիշողությունը: capacity-ի մեծացումը սովորաբար կատարվում է բազմապատկմամբ (օրինակ՝ ×2):
- ✓ **Հիմնական մեթոդներ:** std::string-ի հետ աշխատանքում հաճախ օգտագործվում են հետևյալ մեթոդները՝ find, substr, append, replace, erase:

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Հեշտ) - Տողի վերլուծություն

Իրականացնել ֆունկցիա, որը ստանում է std::string և վերադարձնում է նրա մեջ եղած բառերի քանակը (բառերը բաժանված են բացատներով)։

Մեթոդ՝ անցնել տողի վրայով և հաշվել այն դիրքերը, որտեղ տեղի է ունենում «բացատից ոչ-բացատ սիմվոլի» անցում (նոր բառի սկիզբ)։

Խնդիր 2. (Միջին) - Տողային թվի փոխարկիչ

Իրականացնել ֆունկցիա, որը ստանում է std::string (որը պարունակում է ամբողջ թիվ, օրինակ՝ "1234") և վերադարձնում է համապատասխան int արժեքը։

Պահանջ՝ չօգտագործել std::stoi։

Մեթոդ՝ անցնել տողի վրայով, վերցնել յուրաքանչյուր char, հաշվել (ch - '0') արժեքը և կուտակել արդյունքը։

Խնդիր 3. (Բարդ) - Բառերի շրջում

Իրականացնել ֆունկցիա, որը ստանում է նախադասություն (std::string) և վերադարձնում է նոր տող, որտեղ յուրաքանչյուր բառ շրջված է, սակայն բառերի կարգը պահպանված է։

Օրինակ՝ "hello world" → "olleh dlrow"

Հուշում՝ օգտագործել find, substr մեթոդները բառերը առանձնացնելու համար, std::string-ի կառուցիչը կամ append-ը նոր տողը հավաքելու համար։

4. Տնային հանձնարարություն

Խնդիր 4. (Բարդ) - Պարզեցված MyString դաս

Իրականացնել MyString դասը, որը որպես անդամ փոփոխական պահում է char* զանգված, սակայն ինտերֆեյսով նմանվի std::string-ին։

Իրականացնել՝

- ✓ MyString(const MyString& other) - Պատճենման կառուցիչ
- ✓ MyString(MyString&& other) - Տեղափոխման կառուցիչ
- ✓ MyString& operator=(const MyString& other) - Վերագրման օպերատոր
- ✓ MyString& operator=(MyString&& other) - Տեղափոխման վերագրման օպերատոր։
- ✓ ~MyString() - Փլուզիչ։
- ✓ MyString operator+(const MyString& other) const - Երկու տողերի միացում։
- ✓ MyString& operator+=(const MyString& other) - Տողի կցում։
- ✓ const char* c_str() const - Վերադարձնում է const char*՝ C գրադարանների հետ համատեղելիության համար։
- ✓ size_t length() const - Վերադարձնում է տողի երկարությունը։
- ✓ size_t capacity() const - Վերադարձնում է հատկացված հիշողության չափը։

- ✓ bool empty() const - Ստուգում է՝ տողը արդյոք դատարկ է:
- ✓ void clear() - Մաքրում է տողը:
- ✓ char& operator[](size_t index), const char& operator[](size_t index) const - Սիմվոլների հասանելիություն:
- ✓ void push_back(char c) - Ավելացնում է սիմվոլ վերջում:
- ✓ void pop_back() - Հեռացնում է վերջին սիմվոլը:
- ✓ MyString substr(size_t pos, size_t len) const - Վերադարձնում է ենթատող:

5. Լրացուցիչ առաջադրանքներ

Խնդիր 5. (Հեշտ) - Պալինդրոմ ստուգում

Իրականացնել ֆունկցիա, որը ստուգում է արդյոք տրված std::string-ը պալինդրոմ է (նույնն է ձախից աջ և աջից ձախ):

Օրինակ՝ "racecar" → true, "hello" → false

Հուշում՝ օգտագործել երկու ինդեքս՝ մեկը սկզբից, մյուսը վերջից, համեմատել նիշերը մինչև կենտրոն հասնելը:

Խնդիր 6. (Հեշտ) - Տողի մաքրում

Իրականացնել ֆունկցիա, որը տրված տողից հեռացնում է բոլոր բացատները:

Օրինակ՝ "hel lo wo rld" → "helloworld"

Հուշում՝ օգտագործել erase և find մեթոդները կամ անցնել նիշ առ նիշ և հավաքել նոր տող:

Խնդիր 7. (Միջին) - Տողի փոխարինում

Իրականացնել ֆունկցիա, որը տրված տողում փոխարինում է բոլոր հանդիպումները մի ենթատողից մյուսին։

Օրինակ՝ "I like cats and cats", փոխարինել "cats"-ը "dogs"-ով → "I like dogs and dogs"

Նշում՝ օգտագործել find ցիկլի մեջ, ամեն անգամ գտնելուց հետո կիրառել replace։ Անհրաժեշտ է խուսափել անվերջ ցիկլից։

Խնդիր 8. (Միջին) - CSV փոխարկիչ

Իրականացնել ֆունկցիա, որը ստանում է CSV ձևաչափի տող (ստորակետով բաժանված արժեքներ) և վերադարձնում է std::vector<std::string>՝ առանձին արժեքներով։

Օրինակ՝ "Ani,Aram,Shushan,Tigran" → {"Ani", "Aram", "Shushan", "Tigran"}

Նշում՝ օգտագործել find և substr ցիկլի մեջ՝ յուրաքանչյուր ստորակետի դիրքը գտնելով և ենթատող կտրելով։

Խնդիր 9. (Բարդ) - Տողի սեղմում

Իրականացնել պարզ սեղմման ալգորիթմ. հաջորդական կրկնվող նիշերը փոխարինել նիշ + կրկնությունների քանակ ձևով։

Օրինակ՝ "aaabbbccddddee" → "a3b3c2d4e2"

Հուշում՝ անցնել տողի վրայով, հաշվել հաջորդական կրկնությունները, արդյունքը հավաքել նոր std::string-ում։ Մտածել նաև հակառակ ուղղության (ապասեղմման) ֆունկցիայի մասին։

ԳԼՈՒԽ 2.
ԳԾԱՅԻՆ ՏՎՅԱԼՆԵՐԻ ԿԱՌՈՒՑՎԱԾՔՆԵՐ

Գլուխը նվիրված է գծային տվյալների կառուցվածքներին՝ ինչպես ստանդարտ գրադարանի կոնտեյներներին, այնպես էլ ուսանողների կողմից դրանց իրականացմանը: Ներառված են ստատիկ և դինամիկ չափերով կոնտեյներները (std::array, std::vector), կապակցված ցուցակները (std::forward_list, std::list), պահունակը (std::stack), հերթը (std::queue) և երկկողմանի հերթը (std::deque):

Ուսանողները ոչ միայն սովորում են օգտագործել այս կառուցվածքները, այլև իրականացնում են դրանք՝ հասկանալով ներքին աշխատանքի սկզբունքները՝ հիշողության կառավարումը, ցուցիչների վերահասցեավորումը, իտերատորների կառուցումը և ցիկլիկ զանգվածի տրամաբանությունը: Զուգահեռ ուսումնասիրվում են STL (Standard Template Library) ալգորիթմները և դրանց կիրառելիությունը տարբեր կոնտեյներների վրա:

Արդյունքում ուսանողը կարողանում է գիտակցաբար ընտրել համապատասխան կառուցվածքը կոնկրետ խնդրի համար, ինչպես նաև ինքնուրույն կառուցել ռեսուրսները ճիշտ կառավարող C++ դասեր:

ԴԱՍ 4. STD::ARRAY, ԿԻՐԱՌՈՒԹՅՈՒՆՆԵՐ

Տևողություն՝ 90 րոպե

Թեմա՝ std::array կոնտեյները՝ որպես C ոճի ստատիկ զանգվածների անվտանգ և ժամանակակից այլընտրանք։

1. Դասի նպատակը

Ձևավորել պատկերացում std::array կոնտեյների վերաբերյալ՝ որպես ֆիքսված չափի տվյալների կառուցվածք։ Ցույց տալ դրա առավելությունները սովորական զանգվածների (T arr[]) նկատմամբ, ներկայացնել կաղապարային արգումենտների միջոցով չափի փոխանցումը և կիրառել այն, այդ թվում՝ բազմաչափ տվյալների կառուցման մեջ։

2. Մեթոդական ցուցումներ

- ✓ **std::array-ի բնութագրերը։**
  - std::array-ը ստատիկ զանգվածի վրա կառուցված փաթեթավորող (wrapper) կոնտեյներ է։
  - Չափը որոշվում է կոմպիլյացիայի ժամանակ և ծրագրի ընթացքում չի փոփոխվում։
- ✓ **Առավելությունները։**
  - **Չափի հասանելիություն** - size() մեթոդը միշտ հասանելի է (կարիք չկա առանձին փոփոխական պահելու)։
  - **Անվտանգություն** - at(size_type pos) մեթոդը սահմաններից դուրս ինդեքսի դեպքում նե-

տում է std::out_of_range բացառություն, մինչդեռ [] օպերատորը՝ ոչ։

- **Պատճենում** - հնարավոր է ամբողջ զանգվածի վերագրում (arr1 = arr2)։
- **STL համատեղելիություն** - ապահովում է իտերատորներ (begin(), end()), ինչը թույլ է տալիս կիրառել ստանդարտ ալգորիթմներ (օրինակ՝ std::sort)։

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Հեշտ) - Ստեղծում և սկզբնարժեքավորում

Իրականացնել std::array<int, 5> createArray() ֆունկցիան, որը ստեղծում է զանգված, սկզբնարժեքավորում է այն 1-ից 5 թվերով և վերադարձնում այն, ինչպես նաև արտածել ստացված զանգվածի տարրերը։

Խնդիր 2. (Հեշտ) - Լրացում մեկ արժեքով

Իրականացնել void fillArray(std::array<int, 7>& arr, int value) ֆունկցիան, որը լրացնում է զանգվածի բոլոր տարրերը տրված արժեքով։

Օրինակ՝ fillArray(arr, 42) -> բոլոր տարրերը լրացնում է 42-ով։

Խնդիր 3. (Միջին) - Կաղապարներ և զանգվածներ

Իրականացնել int countEven(const std::array<int, N>& arr) կաղապարային ֆունկցիան, որը հաշվում և վերադարձնում է զանգվածի զույգ տարրերի քանակը՝ օգտագործելով size() մեթոդը կամ range-based for ցիկլը։

Կիրառում՝ countEven<6>(arr):

Խնդիր 4. (Միջին) - Շրջում տեղում

Իրականացնել void reverseArray(std::array<T, N>& arr) կաղապարային ֆունկցիան, որը շրջում է զանգվածի տարրերի հաջորդականությունը տեղում (առաջինը դառնում է վերջինը և այլն)։

Նշում՝ կարելի է օգտագործել std::swap կամ երկու ցուցիչի մեթոդը։

Խնդիր 5. (Բարդ) - Մատրիցային գործողություններ

Իրականացնել Matrix3x3 դասը, որը պահում է տվյալները std::array<std::array<int, 3>, 3> կառուցվածքում, ունի 9 արժեք ընդունող կառուցիչ, ինչպես նաև print() մեթոդ՝ մատրիցը արտածելու և transpose() մեթոդ՝ այն տրանսպոնացնելու համար։

4. Տնային հանձնարարություն

Խնդիր 6. (Հեշտ) - Տարրի որոնում

Իրականացնել int findElement(const std::array<T, N>& arr, const T& value) կաղապարային ֆունկցիան, որը վերադարձնում է առաջին հանդիպած տարրի ինդեքսը կամ -1, եթե տարրը չի գտնվել։

Խնդիր 7. (Միջին) - Ցիկլիկ տեղաշարժ

Իրականացնել void shiftLeft(std::array<T, N>& arr, int k) կաղապարային ֆունկցիան, որը զանգվածի տարրերը տեղաշարժում է ձախ k քայլով՝ դուրս եկած տարրերը տեղափոխելով վերջ։

Օրինակ՝ {1, 2, 3, 4, 5} -> 2 քայլ ձախ -> {3, 4, 5, 1, 2}:

Խնդիր 8. (Բարդ) - Հաճախականությունների անալիզ

Իրականացնել std::array<int, 256> countFrequency (const std::array<char, N>& arr) ֆունկցիան, որը վերադարձնում է զանգված, որտեղ յուրաքանչյուր ինդեքս ցույց է տալիս համապատասխան ASCII սիմվոլի հանդիպումների քանակը:

Օրինակ՝ եթե մուտքը {'h', 'e', 'l', 'l', 'o'} է, ապա ելքային զանգվածի 'l' (108) ինդեքսում պետք է լինի 2:

5. Լրացուցիչ առաջադրանքներ

Խնդիր 9. (Հեշտ) - Զանգվածների համեմատում

Իրականացնել bool compareArrays(const std::array<T, N>& a, const std::array<T, N>& b) կաղապարային ֆունկցիան, որը վերադարձնում է true, եթե զանգվածների բոլոր տարրերը համընկնում են, հակառակ դեպքում՝ false:

Ինքնաստուգում՝ թեև std::array-ը թույլ է տալիս օգտագործել == օպերատորը, իրականացնել այն ցիկլի միջոցով՝ ալգորիթմը հասկանալու համար:

Խնդիր 10. (Միջին) - Տվյալների վավերացում

Իրականացնել bool validateArray(const std::array<T, N>& arr, T min, T max) կաղապարային ֆունկցիան, որը վերադարձնում է true, եթե զանգվածի բոլոր տարրերը գտնվում են [min, max] միջակայքում:

Օրինակ՝ validateArray(scores, 0, 100)՝ ստուգելու համար, որ ուսանողների բոլոր գնահատականները վավեր են։

Խնդիր 11. (Միջին) - Զանգվածի մասնակի պատճենում

Իրականացնել std::array<T, M> copySubArray (const std::array<T, N>& arr, size_t start) կաղապարային ֆունկցիան, որը վերադարձնում է նոր զանգված՝ բաղկացած սկզբնական զանգվածի start ինդեքսից սկսվող M թվով տարրերից։ Երաշխավորել, որ հիշողության սահմաններից դուրս գալու սխալ տեղի չի ունենա, այսինքն՝ start + M <= N։

Խնդիր 12. (Բարդ) - Երկչափ զանգվածի մշակում

Իրականացնել std::array<std::array<T, Cols>, Rows> process2DArray(const std::array<std::array<T, Cols>, Rows>& arr, Func f) կաղապարային ֆունկցիան, որը կիրառում է տրված ֆունկցիան երկչափ զանգվածի յուրաքանչյուր տարրի վրա և վերադարձնում նոր զանգված։

Օրինակ՝ փոխանցել ֆունկցիա, որը կրկնապատկում է թվերը։

6. Մեթոդական լրացում Matrix3x3 խնդրի համար

Գերբեռնել operator[]-ը այնպես, որ հնարավոր լինի գրել matrix[i][j] թե՛ կարդալու, թե՛ գրելու համար։ Սա ցույց կտա, թե ինչպես է std::array-ը հեշտացնում բազմաչափ ինդեքսավորումը։

ԴԱՍ 5. STD::VECTOR, ԿԻՐԱՌՈՒԹՅՈՒՆՆԵՐ

Տևողություն՝ 90 րոպե

Թեմա՝ std::vector կոնտեյները՝ որպես դինամիկ փոփոխվող չափով զանգված: Հիշողության կառավարումը (capacity vs size):

1. Դասի նպատակը

Ուսանողը պետք է հասկանա, թե ինչպես է std::vector-ը ավտոմատ կառավարում հիշողությունը, ինչ տարբերություն կա զանգվածի չափի (size) և տարողունակության (capacity) միջև, և երբ է պետք օգտագործել reserve() մեթոդը՝ արդյունավետությունը բարձրացնելու համար:

2. Մեթոդական ցուցումներ

✓ **Ներքին կառուցվածքը:**

- std::vector-ը դաս է, որը պահում է ցուցիչ դեպի կույտ (heap), ընթացիկ չափը (size) և հատկացված հիշողության չափը (capacity):
- Տվյալները պահվում են անընդհատ հիշողության տիրույթում, ինչը ապահովում է արագ մուտք ըստ ինդեքսի (O(1)) և համատեղելիություն C ոճի կիրառական ծրագրավորման միջերեսների (API) հետ:

✓ **Հիշողության վերահատկացում (Reallocation):**

- Բացատրել, թե ինչ է տեղի ունենում, երբ կանչվում է push_back, բայց capacity-ն սպառվել է.

- 1. Ստեղծվում է նոր, ավելի մեծ զանգված (սովորաբար 2 անգամ մեծ՝ կախված ստանդարտ գրադարանի իրականացումից) -> 2. Հին տարրերը պատճենվում/տեղափոխվում են նորի մեջ -> 3. Հին հիշողությունը ազատվում է:
- Սա հաշվարկային տեսանկյունից ծախսատար գործողություն է, դրա համար էլ կարևոր է reserve()-ը:

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Հեշտ) - Ստեղծում և դինամիկ լրացում

Իրականացնել void printVector(int N) ֆունկցիան, որը ստեղծում է std::vector<int>, հերթով ավելացնում է 1-ից N թվերը դրա մեջ (push_back) և արտածում է վեկտորի տարրերը, ինչպես նաև ընթացիկ size()-ը և capacity()-ն:

Խնդիր 2. (Հեշտ) - Տարողունակության աճի ուսումնասիրություն

Իրականացնել void workWithEmptyVector() ֆունկցիան, որը ստեղծում է դատարկ վեկտոր, 1-ից 10 թվերը հերթով ավելացնում է դրա մեջ (push_back) և յուրաքանչյուր ավելացումից հետո արտածում է վեկտորի size()-ը և capacity()-ն:

Նպատակը՝ տեսնել, թե ինչպես է փոփոխվում capacity-ն:

Խնդիր 3. (Հեշտ) - Օգտատիրոջ մուտքագրում

Իրականացնել std::vector<int> createVectorFromInput() ֆունկցիան, որը կարդում է թվեր օգտատիրոջից մինչև մուտքագրվի 0, հավաքում է դրանք վեկտորի մեջ և վերադարձնում վեկտորը։

Խնդիր 4. (Հեշտ) - Տարրերի հեռացում վերջից

Իրականացնել int removeElementsGreaterThan(std::vector<int>& vec, int threshold) ֆունկցիան, որը տեսակավորված վեկտորի վերջից հեռացնում է բոլոր այն տարրերը, որոնք մեծ են տրված շեմից, և վերադարձնում է ջնջված տարրերի քանակը։

Խնդիր 5. (Հեշտ) - Արդյունավետության բարձրացում

Իրականացնել void manageCapacity() ֆունկցիան, որը նախապես կանչում է reserve(500), այնուհետև push_back-ով ավելացնում է 500 տարր և հնարավորություն է տալիս համեմատել capacity-ի փոփոխությունները reserve-ի օգտագործման և չօգտագործման դեպքում։

Խնդիր 6. (Միջին) - Չափի փոփոխություն

Իրականացնել void resizeVector(std::vector<T>& vec, size_t newSize, const T& value) կաղապարային ֆունկցիան, որը փոխում է վեկտորի չափը resize() մեթոդով և անհրաժեշտության դեպքում նոր տարրերը լրացնում է տրված արժեքով։
Օրինակ ՝ -> resizeVector({1, 2}, 5, 42) -> {1, 2, 42, 42, 42}:

4. Տնային հանձնարարություն

Խնդիր 7. (Միջին) - Տեսակավորված վեկտորների միաձուլում

Իրականացնել `std::vector<int> merge(const std::vector<int>& a, const std::vector<int>& b)` ֆունկցիան, որը միաձուլում է երկու տեսակավորված վեկտորները և վերադարձնում է նոր տեսակավորված վեկտոր՝ օգտագործելով երկու ցուցիչի մեթոդը:

Խնդիր 8. (Բարդ) - Դինամիկ մատրիցի ընդլայնված մեթոդներ

Իրականացնել `DynamicMatrix` դասը, որը պահում է տվյալները `std::vector<std::vector<int>>` կառուցվածքում և ապահովում է `addRow()` և `addColumn()` մեթոդներ՝ տող և սյուն դինամիկ ավելացնելու համար: Իրականացնել հետևյալ մեթոդները՝ սահմանների ստուգմամբ.

- ✓ `int getElement(size_t row, size_t col)` - վերադարձնում է տրված տողին ու սյանը համապատասխանող տարրը,
- ✓ `void setElement(size_t row, size_t col, int value)` - տրված տողին ու սյանը համապատասխանող տարրին վերագրում է `value`:

Հուշում՝ սյունակ ավելացնելիս պետք է անցնել բոլոր տողերի վրայով և յուրաքանչյուրին վերջում ավելացնել 0:

Խնդիր 9. (Բարդ) - Վիճակագրություն

Ստեղծել VectorStats դաս, որը պահում է թվերի հավաքածու և հաշվարկում է նրա վիճակագրական ցուցանիշները՝ միջին թվաբանականը (mean) և միջնարժեքը (median):

Դասը պետք է ապահովի հետևյալ հնարավորությունները.

- ✓ Ավելացնել նոր արժեք addValue մեթոդով
- ✓ Հեռացնել գոյություն ունեցող արժեք removeValue մեթոդով
- ✓ Յուրաքանչյուր փոփոխությունից հետո թարմացնել և պահպանել mean և median արժեքները

5. Լրացուցիչ առաջադրանքներ

Խնդիր 10. (Միջին) - Ենթահաջորդականության որոնում

Իրականացնել int findSubsequence(const std::vector<int>& mainVec, const std::vector<int>& subVec) ֆունկցիան, որը վերադարձնում է առաջին հանդիպած ենթահաջորդականության սկզբնական ինդեքսը կամ -1, եթե այն չի գտնվել:

Օրինակ՝ {1, 2, 3, 4, 5} և {3, 4} -> արդյունքը՝ 2:

Խնդիր 11. (Միջին) - Ֆիլտրում ըստ պայմանի

Իրականացնել std::vector<T> filterVector(const std::vector<T>& vec, Predicate pred) կաղապարային ֆունկցիան, որը վերադարձնում է նոր վեկտոր, որը

պարունակում է միայն այն տարրերը, որոնք բավարարում են այդ պայմանին (օրինակ՝ միայն զույգ թվերը)։

Խնդիր 12. (Միջին) - Հարևան նույնական տարրերի խմբավորում

Իրականացնել `std::vector<std::vector<int>> groupConsecutive(const std::vector<int>& vec)` ֆունկցիան, որը խմբավորում է հարևան նույնական տարրերը առանձին վեկտորների մեջ և վերադարձնում է արդյունքը։

Օրինակ՝ {1, 1, 2, 2, 2, 3, 1, 1} -> {{1, 1}, {2, 2, 2}, {3}, {1, 1}}։

Խնդիր 13. (Միջին) - Տարրերի հատում

Իրականացնել `std::vector<int> intersect(const std::vector<int>& a, const std::vector<int>& b)` ֆունկցիան, որը վերադարձնում է այն տարրերը, որոնք առկա են երկու վեկտորներում։

6. Մեթոդական ցուցումներ

- ✓ **Խնդիր 10-ի համար:** Քննարկել երկու ներդրված ցիկլերի աշխատանքը և եզրային դեպքերը (օրինակ՝ երբ որոնվող վեկտորը ավելի մեծ է, քան հիմնականը)։
- ✓ **Խնդիր 12-ի համար:** Ուսանողները հաճախ սխալվում են ինդեքսների հետ աշխատելիս (օրինակ՝ `index+1` օգտագործելիս զանգվածից դուրս գալը)։ Խորհուրդ տվեք օգտագործել ժամանակավոր վեկտոր խմբերը կառուցելու համար։

ԴԱՍ 6. ՎԵԿՏՈՐԻ ԻՐԱԿԱՆԱՑՈՒՄ

Տևողություն՝ 90 րոպե

Թեմա՝ Հիշողության հատկացման (allocation) և օբյեկտի կառուցման (construction) տարանջատումը: placement new-ի և փլուզիչի բացահայտ կանչի (explicit destructor call) կիրառումը:

1. Դասի նպատակը

Ուսանողը պետք է սովորի իրականացնել Vector դասի իր տարբերակը՝ առանց օգտագործելու new T[] ստանդարտ եղանակը: Նա պետք է հասկանա, թե ինչպես հատկացնել դատարկ հիշողություն և օբյեկտները կառուցել միայն անհրաժեշտության դեպքում:

2. Մեթոդական ցուցումներ

✓ **Խնդրի մոտիվացիան:**

- Ինչու՞ T* arr = new T[capacity] մոտեցումը հարմար չէ վեկտորի իրականացման համար:
- *Պատճառ 1:* Այն միանգամից կանչում է լռությամբ (default) կառուցիչը capacity օբյեկտների համար (իսկ եթե օգտագործվելու են ընդամենը 5 օբյեկտներ, ապա մնացած 95-ը ստեղծվում են առանց անհրաժեշտության):
- *Պատճառ 2:* Որոշ տիպեր (օրինակ՝ struct Student) կարող են չունենալ լռությամբ կառուցիչ, և կոդը չի կոմպիլացվի:

- ✓ **Լուծումը (operator new, placement new):**
  - Դատարկ (raw) հիշողություն ստանալու համար օգտագործվում է ::operator new(size_in_bytes):
  - Այդ հիշողության մեջ օբյեկտ կառուցելու համար օգտագործվում է placement new՝ new (address) T(arguments):
  - Օբյեկտը ոչնչացնելու համար (առանց հիշողությունը ազատելու) կանչել ptr->~T():

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Հեշտ) - Կմախքի կառուցում

Ստեղծել MyVector<T> կաղապարային դասը, որն ունի՝

- ✓ T* data - Ցուցիչ դեպի հիշողության սկիզբը:
- ✓ size_t sz - Իրական տարրերի քանակը:
- ✓ size_t cap - Հատկացված հիշողության չափը (տարողունակություն):
- ✓ Կառուցիչ - Սկզբնարժեքավորում nullptr-ով և 0-ով:
- ✓ Փլուզիչ - Առայժմ դատարկ (կլրացվի վերջում):

Խնդիր 2. (Բարդ) - Հիշողության հատկացում

Իրականացնել reserve(size_t new_cap) մեթոդը:

- ✓ *Քայլ 1:* Եթե new_cap <= cap, ոչինչ չանել:
- ✓ *Քայլ 2:* Հատկացնել դատարկ հիշողություն՝ T* new_data = static_cast<T*>(::operator new(new_cap * sizeof(T))):

- ✓ *Քայլ 3 (ցիկլ):* Հին տարրերը տեղափոխել նոր հիշողություն՝ new (new_data + i) T(std::move(data[i])) (օգտագործելով placement new):
- ✓ *Քայլ 4 (ցիկլ):* Կանչել հին օբյեկտների փլուզիչները՝ data[i].~T():
- ✓ *Քայլ 5:* Ազատել հին հիշողությունը՝ ::operator delete(data):
- ✓ *Քայլ 6:* Թարմացնել data և cap փոփոխականները:

Խնդիր 3. (Միջին) - Վերջում տարրի ավելացում (placement new-ի կիրառում)

Իրականացնել void push_back(const T& value) և void push_back(T&& value) ֆունկցիաները, որոնք անհրաժեշտության դեպքում (sz == cap) կանչում են reserve, ապա տեղադրում են նոր տարրը ազատ հիշողության մեջ new (data + sz) T(value) (կամ std::move (value) rvalue-ի դեպքում) և մեծացնում են sz-ը:

Նշում՝ այս իրականացումն ավելի արդյունավետ է, քան սովորական վերագրումը, քանի որ չի պահանջում նախապես ստեղծված օբյեկտ:

Խնդիր 4. (Միջին) - Վերջից տարրի հեռացում և փլուզիչ

- ✓ Իրականացնել pop_back() մեթոդը՝ կանչելով վերջին տարրի փլուզիչը (data[sz - 1].~T()) և նվազեցնելով sz-ը:

- ✓ ~MyVector():
  - ○ Ցիկլով կանչել բոլոր գոյություն ունեցող (0-ից մինչև sz - 1) օբյեկտների փլուզիչները:
  - ○ Վերջում ազատել հիշողությունը՝ ::operator delete(data):

Խնդիր 5. (Հեշտ) - Տարրի հասանելիություն

Իրականացնել T& operator[](size_t index), const T& operator[](size_t index) const և T& at(size_t index), const T& at(size_t index) const մեթոդները:

Որտեղ՝

- ✓ operator[]-ը վերադարձնում է վեկտորի index-րդ տարրի հղումը՝ առանց սահմանների ստուգման: Եթե index-ը դուրս է վեկտորի սահմաններից (index >= sz), վարքը չսահմանված է (undefined behavior):
- ✓ Իսկ at() մեթոդը վերադարձնում է վեկտորի index-րդ տարրի հղումը՝ կատարելով սահմանների ստուգում: Եթե index >= sz, ապա պետք է նետվի բացառություն (std::out_of_range):
- ✓ Երկու մեթոդներն էլ պետք է ունենան const տարբերակներ, որպեսզի հնարավոր լինի կիրառել նաև հաստատուն (const) օբյեկտների համար:

4. Օրինակ (հիմնական գաղափարների ցուցադրում)

```
template <typename T>
class MyVector {
    T* data;
    size_t sz;
    size_t cap;

public:
    MyVector() : data(nullptr), sz(0), cap(0) {}

    void push_back(const T& value) {
        if (sz >= cap) {
            // Եթե capacity-ն 0 է, դարձնել 1, հակառակ դեպքում կրկնապատկել
            reserve(cap == 0 ? 1 : cap * 2);
        }
        // Placement new
        new (data + sz) T(value);
        sz++;
    }

    ~MyVector() {
            clear(); // Ոչնչացնել օբյեկտները
            ::operator delete(data); // Ազատել հիշողությունը
    }
```

```
    void clear() {
        for (size_t i = 0; i < sz; ++i) {
        data[i].~T(); // Փլուզիչի բացահայտ կանչ
        }
    }
    // ...
};
```

5. Տնային հանձնարարություն

Խնդիր 6. (Միջին) - Չափի փոփոխություն

Իրականացնել `resize(size_t new_size, const T& default_val = T())` մեթոդը:

- ✓ Եթե `new_size > sz`, պետք է ավելացնել նոր տարրեր (placement new-ով):
- ✓ Եթե `new_size < sz`, պետք է հեռացնել ավելորդները (փլուզիչի կանչով):

Խնդիր 7. (Բարդ) - Հնգյակի կանոն

Քանի որ դասը կառավարում է ռեսուրսներ (հիշողություն), անհրաժեշտ է, որ այն ունենա՝

- ✓ Պատճենման կառուցիչ (խորը պատճենում):
- ✓ Պատճենման վերագրման օպերատոր:
- ✓ Տեղափոխման կառուցիչ (ռեսուրսների տեղափոխում):
- ✓ Տեղափոխման վերագրման օպերատոր:
- ✓ Փլուզիչ (արդեն գրված է):

ԴԱՍ 7. STD::FORWARD_LIST, ԿԻՐԱՌՈՒԹՅՈՒՆՆԵՐ

Տևողություն՝ 90 րոպե

Թեմա՝ std::forward_list և նրա կիրառումը տարբեր խնդիրներում՝ արդյունավետ տեղադրում և հեռացում, բազմանդամների և առաջադրանքների կառավարման համակարգեր:

1. Դասի նպատակը

Ուսանողը պետք է տիրապետի միակապ ցուցակի առանձնահատկություններին՝ հասկանալով, թե ինչու են օգտագործվում _after վերջավորությամբ մեթոդները (օրինակ՝ insert_after, emplace_after, erase_after, splice_after) և ինչ դեր ունի before_begin() իտերատորը, ինչպես նաև կարողանա կառուցել բարդ ներկայացումներ (օրինակ՝ բազմանդամներ) այս կոնտեյների հիման վրա:

2. Մեթոդական ցուցումներ

- ✓ **Միակողմանի անցում:** Շեշտել, որ std::forward_list::iteratorը չի ապահովում operator-- (հետադարձ քայլ):
- ✓ **before_begin իտերատոր**: Բացատրել flist.before_begin() իտերատորի կարևորությունը, որը թույլ է տալիս աշխատել ցուցակի սկբի տարրի հետ (օրինակ՝ որպեսզի insert_after-ը և

erase_after()-ը աշխատեն նաև սկզբի տարրի համար):

- ✓ **Բարդություն:** Ցույց տալ, որ insert_after-ը ունի O(1) բարդություն, սակայն N-րդ տարրին դիմելը O(N) է: Քանի որ std::forward_list-ը չի ապահովում կամայական հասանելիություն (random access), անհրաժեշտ է անցնել ցուցակի սկզբից:

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Հեշտ) - Ցուցակի ստեղծում

Իրականացնել std::forward_list<int> createAndFillList(int n) ֆունկցիան, որը ստեղծում է ցուցակ և push_front մեթոդով ավելացնում է 1-ից մինչև n թվերը:

Խնդիր 2. (Հեշտ) - Տեղադրում դիրքից հետո

Իրականացնել void insertAfterPosition(std::forward_list<int>& flist, size_t pos, int value) ֆունկցիան, որը ավելացնում է նոր տարր տրված դիրքից հետո:

Խնդիր 3. (Միջին) - Տարրերի հեռացում

Իրականացնել void removeAllOccurrences(std::forward_list<int>& flist, int value) ֆունկցիան, որը հեռացնում է տրված արժեքի բոլոր հանդիպումները:

Խնդիր 4. (Միջին) - Տեղափոխում դեպի սկիզբ

Իրականացնել void moveElementsToFront(std::forward_list<int>& flist, int value) ֆունկցիան, որը տե-

ղափոխում է տրված արժեքով տարրերը ցուցակի սկիզբ։

Խնդիր 5. (Բարդ) - Բազմանդամների դաս

Իրականացնել Polynomial դասը, որը որպես ներքին տվյալների կառուցվածք օգտագործում է std::forward_list<std::pair<int, double>>, որտեղ յուրաքանչյուր տարր ներկայացնում է {աստիճան, գործակից} զույգը։

Իրականացնել՝

- ✓ void addTerm(int degree, double coeff) մեթոդը, որը ավելացնում է նոր անդամ՝ պահպանելով աստիճանների նվազման կարգը։ Եթե degree-ն գոյություն ունի, գումարել coeff-ը։
- ✓ double evaluate(double x) const մեթոդը, որը հաշվարկում է բազմանդամի արժեքը տրված x կետում։
- ✓ Polynomial derivative() const մեթոդը, որը վերադարձնում է նոր Polynomial օբյեկտ՝ հանդիսանալով տվյալ բազմանդամի ածանցյալը։

4. Տնային հանձնարարություն

Խնդիր 6. (Միջին) - Առաջադրանքների կառավարիչ

Իրականացնել TaskManager դասը, որը որպես անդամ փոփոխական պահում է std::vector<std::forward_list<std::string>>, որտեղ վեկտորի յուրաքան-

յուր ինդեքս ներկայացնում է առաջնահերթության մակարդակ (0՝ ամենաբարձր)։

Իրականացնել՝

- ✓ std::string getNextTask() մեթոդը, որը վերադարձնում և հեռացնում է ամենաբարձր առաջնահերթություն ունեցող առաջին առաջադրանքը։
- ✓ void printAllTasks() const մեթոդը, որը արտածում է բոլոր առաջադրանքները՝ ըստ առաջնահերթությունների։

Խնդիր 7. (Բարդ) - Ներդրման տեսակավորում

Իրականացնել void insertionSort(std::forward_list<int>& flist) ֆունկցիան, որը տեսակավորում է std::forward_list<int>-ը՝ օգտագործելով ներդրման տեսակավորման (Insertion Sort) ալգորիթմը։

Հուշում՝ կարելի է օգտագործել լրացուցիչ ցուցակ և տարրերը տեղադրել համապատասխան դիրքում։

5. Լրացուցիչ առաջադրանքներ

Խնդիր 8. (Միջին) - Տեսակավորված ցուցակների միաձուլում

Իրականացնել void mergeSortedLists(std::forward_list<int>& list1, std::forward_list<int>& list2) ֆունկցիան, որը միավորում է երկու տեսակավորված ցուցակները առաջինում այնպես, որ արդյունքը մնա տեսակավորված, իսկ երկրորդ ցուցակը դառնա դատարկ։

Հուշում՝ օգտագործել std::forward_list::merge մեթոդը կամ իրականացնել համապատասխան տրամաբանություն իտերատորների միջոցով։

Խնդիր 9. (Միջին) - Հեռացում ըստ պայմանի

Իրականացնել size_t removeIfCondition(std::forward_list<int>& flist, bool (*predicate)(int)) ֆունկցիան, որը հեռացնում է բոլոր այն տարրերը, որոնք բավարարում են տրված պայմանին, և վերադարձնում է հեռացված տարրերի քանակը։

Օրինակ՝ Հեռացնել բոլոր կենտ թվերը կամ բոլոր բացասական թվերը։

Խնդիր 10. (Բարդ) - Ցուցակի բաժանում

Իրականացնել std::pair<std::forward_list<int>, std::forward_list<int>> splitList(const std::forward_list<int>& flist) ֆունկցիան, որը բաժանում է ցուցակը երկու մասի՝ առաջինում տեղադրելով զույգ, իսկ երկրորդում՝ կենտ տարրերը՝ պահպանելով դրանց հարաբերական կարգը։

Խնդիր 11. (Միջին) - Կրկնվող տարրերի հեռացում

Իրականացնել void removeConsecutiveDuplicates (std::forward_list<int>& flist) ֆունկցիան, որը հեռացնում է հարևան նույնական տարրերը ցուցակից։

Օրինակ՝ {1, 2, 2, 3, 3, 3, 4, 2} -> {1, 2, 3, 4, 2}։

Հուշում՝ ուսանողները կարող են օգտագործել flist.unique() մեթոդը կամ իրականացնել այն՝ օգտագործելով erase_after։

Խնդիր 12. (Բարդ) - Ցուցակի միջնամասի որոնում

Իրականացնել int findMiddle(const std::forward_list<int>& flist) ֆունկցիան, որը մեկ անցումով գտնում և վերադարձնում է ցուցակի մեջտեղի տարրը՝ օգտագործելով «դանդաղ» և «արագ» ցուցիչների մեթոդը։

ԴԱՍ 8. ԿԱՊԱԿՑՎԱԾ ՑՈՒՑԱԿՆԵՐԻ ԻՐԱԿԱՆԱՑՈՒՄ

Տևողություն՝ 90 րոպե

Թեմա՝ Միակապ և երկկապ ցուցակներ, հանգույցների կառավարում, դինամիկ հիշողության կառավարում։

1. Դասի նպատակը

Ուսանողը պետք է հասկանա ցուցակների ներքին կառուցվածքը հիշողության և ցուցիչների տեսանկյունից։ Կապակցված ցուցակների իրականացումը զարգացնում է դինամիկ հիշողության հետ աշխատելու հմտությունները և նախապատրաստում ավելի բարդ կառուցվածքների (ծառեր, գրաֆներ) ուսումնասիրմանը։

2. Մեթոդական ցուցումներ

- ✓ **Պատկերային ներկայացում:** Գրատախտակին գծել հանգույցները (node) և սլաքներով ցույց

տալ next ու prev (երկկապ ցուցակի դեպքում) կապերը: Ցույց տալ, թե ինչպես են փոփոխվում տարր ավելացնելիս կամ հեռացնելիս:

- ✓ **Եզրային դեպքեր:** Քննարկել՝ ինչ է լինում, եթե ցուցակը դատարկ է (head == nullptr, որտեղ head-ը առաջին հանգույցի ցուցիչն է) կամ ունի միայն մեկ տարր:
- ✓ **Հիշողության արտահոսք (memory leak)**: Շեշտել փլուզիչի կարևորությունը, որը պետք է հաջորդականորեն ջնջի բոլոր հանգույցները:

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Հեշտ) - Միակապ ցուցակի հանգույց

Իրականացնել LinkedList դաս, որը պարունակում է head ցուցիչ դեպի առաջին հանգույցը, իսկ յուրաքանչյուր հանգույց ունի next ցուցիչ: Իրականացնել հետևյալ մեթոդները.

- ✓ push_front(int value) - ստեղծում է նոր Node հանգույց և այն տեղադրում ցուցակի սկզբում,
- ✓ void pop_front() - հեռացնում է առաջին հանգույցը, եթե ցուցակը դատարկ չէ:

Խնդիր 2. (Միջին) - Տարրի հեռացում ըստ արժեքի

Ավելացնել void remove(int value) մեթոդ միակապ ցուցակին:

Նշում՝ անհրաժեշտ է պահպանել նախորդ հանգույցի հասցեն, որպեսզի կապը հանգույցների միջև չկորչի հեռացնելուց հետո:

Խնդիր 3. (Բարդ) - Երկկապ ցուցակ

Իրականացնել DoublyLinkedList դաս, որն ունի head և tail ցուցիչներ համապատասխանաբար առաջին և վերջին հանգույցների վրա, իսկ յուրաքանչյուր հանգույց ունի next և prev ցուցիչներ: Իրականացնել void push_back(int value) և void printReverse() const մեթոդները:

Հարց՝ ինչու՞ է երկկապ ցուցակն ավելի արդյունավետ pop_back գործողության համար, քան միակապը:

4. Տնային հանձնարարություն

Խնդիր 4. (Միջին) - Ցուցակի շրջում

Իրականացնել ֆունկցիա, որը շրջում է միակապ ցուցակը (առանց լրացուցիչ զանգված օգտագործելու, միայն ցուցիչների վերահասցեավորմամբ):

Խնդիր 5. (Միջին) - Փլուզիչ, պատճենման կառուցիչ, վերագրման օպերատոր

Միակապ և երկապ ցուցակի համար իրականացնել՝

- ✓ Փլուզիչ, որը ապահովում է ցուցակի բոլոր հանգույցների համար հատկացված հիշողության ազատումը:
- ✓ Պատճենման կառուցիչ - կատարում է խորը պատճենում (deep copy):

- ✓ Վերագրման օպերատոր (operator=) - կատարում է խորը պատճենում, հաշվի առնել որ օբյեկը կարող է ինքն իրեն վերագրվել (self assignment):

ԴԱՍ 9. ՄԻԱԿԱՊ ՑՈՒՑԱԿԻ ԻՏԵՐԱՏՈՐՆԵՐԻ ԻՐԱԿԱՆԱՑՈՒՄ

Տևողություն՝ 90 րոպե

Թեմա՝ իտերատորի իրականացում, օպերատորների գերբեռնում (++, *, ==, !=), ցուցակի աբստրակցիա:

1. Դասի նպատակը

Սովորեցնել ուսանողին ստեղծել իտերատոր (iterator) աբստրակցիա, որը թույլ է տալիս հաջորդաբար դիմել ցուցակի տարրերին առանց Node* ցուցիչների հետ ուղղակի աշխատելու:

2. Մեթոդական ցուցումներ

- ✓ **Իտերատորը որպես անցման աբստրակցիա:** Իտերատորը թույլ է տալիս հաջորդաբար դիմել կոնտեյների տարրերին՝ առանց իմանալու կոնտեյների ներքին կառուցվածքը:
- ✓ **Շարահյուսություն:** Ցույց տալ, որ begin()-ը վերադարձնում է իտերատոր, որը հղվում է առաջին տարրի վրա, իսկ end()-ը վերադարձնում է իտերատոր վերջին էլեմենտի «հաջորդի» վրա:

- ✓ **Range-based for ցիկլ**: Նշել, որ range-based for-ը աշխատում է, եթե առկա են՝ begin(), end(), operator++, operator*, operator!=: Բացատրել նաև, որ for (int x : myList) կոդը համարժեք է հետևյալ կոդին՝

```
for (auto it = myList.begin(); it != myList.end();
++it) {
 int x = *it;
}
```

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Միջին) - Iterator դաս

Նախորդ դասին իրականացված LinkedList դասի ներսում ստեղծել Iterator դաս:

Իրականացնել՝

- ✓ Կառուցիչ - Սկզբնարժեքավորում է իտերատորը տրված Node* ցուցիչով:
- ✓ int& operator*(), const int& operator*() const - Վերադարձնում է ptr->data-ի հղումը:
- ✓ Iterator& operator++() - Պրեֆիքս ինկրեմենտ, տեղափոխում է ptr-ը հաջորդ հանգույցի վրա և վերադարձնում է հղում նույն իտերատորին:
- ✓ Iterator operator++(int) - Պոստֆիքս ինկրեմենտ, վերադարձնում է իտերատոր ընթացիկ հանգույցի վրա, սակայն տեղափոխում է ptr-ը հաջորդ հանգույցի վրա:

- ✓ bool operator!=(const Iterator& other) const - Ստուգում է՝ արդյոք իտերատորները տարբեր են:
- ✓ bool operator==(const Iterator& other) const - Ստուգում է՝ արդյոք իտերատորները հավասար են:
- ✓ LinkedList դասում ավելացնել Iterator begin() և Iterator end() մեթոդները, որտեղ begin() մեթոդը վերադարձնում է իտերատոր, որը ցույց է տալիս ցուցակի առաջին տարրը, իսկ end() մեթոդը վերադարձնում է իտերատոր, որի ներքին ցուցիչը հավասար է nullptr:

Խնդիր 2. (Հեշտ) - Ցիկլերի համատեղելիություն

Փորձարկել iterator-ի համատեղելիությունը range-based for ցիկլի հետ:

Օրինակ՝

```
LinkedList myList;
myList.push_front(10);
myList.push_front(20);
// Պահանջում է begin(), end() և operator ++, operator*,
operator!= ֆունկցիաների իրականացում
for (int x : myList) {
   std::cout << x << " ";
}
```

4. Տնային հանձնարարություն

Խնդիր 3. (Միջին) - Հաստատուն իտերատոր

Իրականացնել ConstIterator, որը թույլ կտա միայն կարդալ տարրերը: Բացատրել, թե ինչպես է ConstIterator-ը ապահովում տվյալների անփոփոխելիությունը const LinkedList օբյեկտների շրջանցման ժամանակ, օրինակ, երբ ֆունկցիան ունի const LinkedList& list պարամետր:

ԴԱՍ 10. STD::LIST, ԿԻՐԱՌՈՒԹՅՈՒՆՆԵՐ

Տևողություն՝ 90 րոպե

Թեմա՝ std::list, std::find, std::count_if, իտերատորների կատեգորիաներ և ֆունկցիաները որպես պրեդիկատներ:

1. Դասի նպատակը

Ուսանողը պետք է կարողանա կիրառել ստանդարտ ալգորիթմները (std::find, std::count_if և այլն) տարբեր կոնտեյներների վրա և հասկանա իտերատորների կատեգորիաների սահմանափակումները:

2. Մեթոդական ցուցումներ

- ✓ **Իտերատորների կատեգորիաները:** std::list-ը ապահովում է երկուղղված (Bidirectional) իտերատորներ, իսկ std::forward_list-ը՝ միայն միաուղղված (Forward): Ոչ մեկը չի ապահովում

կամայական հասանելիություն (random access), այսինքն՝ չի կարելի գրել it + 5:

- Նշել, որ որոշ ալգորիթմներ (օրինակ՝ std::sort) պահանջում են պատահական հասանելիության իտերատորներ, այդ պատճառով չեն աշխատում std::list-ի հետ:

✓ **Ֆունկցիան որպես պրեդիկատ:** Սովորեցնել փոխանցել ֆունկցիան որպես պարամետր (function pointer, std::function կամ lambda), որը վերադարձնում է bool և օգտագործվում է որպես պայման:

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Հեշտ) - Որոնում և հաշվարկ

Իրականացնել bool isEven(int n) ֆունկցիան, որը ստուգում է թվի զույգությունը: Օգտագործել std::count_if և փոխանցել isEven-ը՝ հաշվելու զույգ թվերի քանակը std::list<int> տիպի օբյեկտի մեջ:

Խնդիր 2. (Միջին) - std::sort-ի սահմանափակումը

Կիրառել std::sort վեկտորի և ցուցակի վրա: Փորձել կիրառել std::sort-ը std::list-ի նկատմամբ և վերլուծել կոմպիլյացիայի սխալի պատճառը: Օգտագործել myList.sort() (անդամ-ֆունկցիան) և ներկայացնել, որ std::list-ը տրամադրում է իր անդամ-ֆունկցիաները (օ-

րինակ՝ list.sort()), որոնք օպտիմիզացված են կապակցված ցուցակների համար

Խնդիր 3. (Միջին) - Տվյալների տեղափոխում

Օգտագործելով std::copy և std::back_inserter, պատճենել տարրերը std::vector-ից դեպի std::list: Այնուհետև օգտագործել std::reverse ցուցակի տարրերը շրջելու համար:

Խնդիր 4. (Հեշտ) – Առաջին համապատասխան տարրը

Օգտագործելով std::find_if, գտնել առաջին կենտ թիվը ցուցակում:

Խնդիր 5. (Միջին) – Պայմանով հաշվարկ

Օգտագործելով std::count_if, հաշվել այն տարրերի քանակը ցուցակում, որոնք մեծ են տրված k թվից:

Խնդիր 6. (Միջին) – Հեռացում տարբեր եղանակներով

Հեռացնել բոլոր բացասական թվերը ցուցակից՝ std::remove_if կամ erase կիրառելով:

4. Տնային հանձնարարություն

Խնդիր 7. (Միջին) - Ուսանողների ֆիլտրում

Ստեղծել Student ստուկտուրա (անուն, գնահատական): Ստեղծել std::list<Student>: Իրականացնել ֆունկցիա bool isFailing(const Student& s), որը վերադարձնում է true, եթե գնահատականը < 60: Օգտագործել myList.remove_if(isFailing)՝ անբավարար ստացած ուսանողներին ցուցակից հեռացնելու համար:

Խնդիր 8. (Միջին) – Երկու ցուցակների միավորում

Տրված են երկու տեսակավորված std::list<int>: Միավորել դրանք մեկ տեսակավորված ցուցակի մեջ (merge):

Խնդիր 9. (Միջին) – Պատճենում պայմանով

Օգտագործելով std::copy_if, պատճենել միայն զույգ թվերը std::list-ից դեպի std::vector:

Խնդիր 10. (Բարդ) – Առավելագույնի որոնում

Օգտագործելով std::max_element, գտնել ամենամեծ տարրը: Գրել նաև հատուկ համեմատման ֆունկցիա (custom comparator):

6. Լրացուցիչ առաջադրանքներ

Խնդիր 11. (Բարդ) – Ենթահերթի շրջում

Իտերատորներով շրջել միայն [it1, it2) միջակայքը (std::reverse):

Խնդիր 12. (Բարդ) – Պրեդիկատ որպես օբյեկտ

Իրականացնել ֆունկցիոնալ օբյեկտ (functor)՝ դասի կամ ստրուկտուրայի տեսքով, որը գերբեռնում է operator()-ը և ստուգում է՝ արդյոք տրված թիվը պատկանում է [a, b] միջակայքին (a և b սահմանները փոխանցվում են կոնստրուկտորի միջոցով): Օգտագործելով այս ֆունկցիոնալ օբյեկտը և std::count_if ալգորիթմը՝ հաշվել std::list կոնտեյների այն տարրերի քանակը, որոնք բավարարում են նշված պայմանին:

Խնդիր 13. (Հեշտ) – Իտերատորների սահմանափակումներ

Փորձել օգտագործել it + n՝

✓ vector-ի դեպքում,

✓ list-ի դեպքում։

Բացատրել՝ ինչու է աշխատում միայն առաջին դեպքում։

ԴԱՍ 11. STD::DEQUE, ԿԻՐԱՌՈՒԹՅՈՒՆՆԵՐ

Տևողություն՝ 90 րոպե

Թեմա՝ Երկկողմանի հերթ (double-ended queue), տարրերի ավելացում և հեռացում երկու ծայրերից։

1. Դասի նպատակը

Ձևավորել պատկերացում std::deque կոնտեյների կառուցվածքի և աշխատանքի սկզբունքների վերաբերյալ, ցույց տալ դրա տարբերությունները std::vector-ի համեմատ և նախապատրաստել ուսանողին հասկանալու std::queue ադապտերի աշխատանքը։

2. Մեթոդական ցուցումներ

✓ **Երկկողմանի հերթ**: Նշել, որ std::deque-ը թույլ է տալիս տարրերի ավելացում և հեռացում ինչպես սկզբից (push_front, pop_front), այնպես էլ վերջից (push_back, pop_back)՝ O(1) ժամանակում։

- ✓ **Կառուցվածքային առանձնահատկություն**: Տիպիկ իրականացումներում std::deque-ն չի պահվում մեկ շարունակական հիշողության հատվածում, այլ կազմված է բլոկներից, ինչը թույլ է տալիս արդյունավետ աշխատել երկու ծայրերից:
- ✓ **Հասանելիություն**: Նշել, որ ապահովվում է ինդեքսով հասանելիություն (operator[], at()), ինչպես std::vector-ի դեպքում: Նաև ապահովվում է կամայական հասանելությամբ իտերատոր (random access):
- ✓ **Համեմատություն vector-ի հետ**: Ընդգծել, որ vector-ում սկզբից տարր ավելացնելը ժամանակային ավելի մեծ բարդություն ունի (O(N)), մինչդեռ deque-ում՝ O(1) է:
- ✓ **Կիրառման ոլորտներ**: Նշել, որ std::deque-ը կիրառվում է այն խնդիրներում, որտեղ անհրաժեշտ է արագ աշխատանք երկու ծայրերից (օրինակ՝ սահող պատուհան, սիմուլյացիաներ, լայնությամբ շրջանցման որոշ տարբերակներ):

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Հեշտ) - Ստեղծում և լրացում երկու կողմից

Ստեղծել դատարկ std::deque<int> և լրացնել այն՝ վերջում ավելացնելով 1-ից N միջակայքի կենտ թվերը,

իսկ սկզբում՝ զույգ թվերը: Արտածել երկկողմանի հերթի տարրերը:

Խնդիր 2. (Հեշտ) - Երկու կողմից հեռացում

Իրականացնել void removeFromBothEnds (std::deque<int>& d, int k) ֆունկցիան, որը հերթով հեռացնում է տարրեր երկկողմանի հերթի սկզբից և վերջից՝ k անգամ՝ յուրաքանչյուր քայլից հետո արտածելով մնացած տարրերը:

Խնդիր 3. (Հեշտ) - Երկկողմանի հերթի պտտում

Իրականացնել template <typename T> void rotate Deque(std::deque<T>& d, int k) ֆունկցիա, որը պտտում է երկկողմանի հերթը k դիրքով դեպի ձախ՝ օգտագործելով std::deque-ի հիմնական գործողությունները:

4. Տնային հանձնարարություն

Խնդիր 4. (Միջին) - Ժամանակագրական բուֆեր (Logger)

Իրականացնել Logger դաս, որը պահում է առավելագույնը N վերջին հաղորդագրությունները՝ օգտագործելով std::deque<std::string>:

Խնդիր 5. (Բարդ) - «Օձ» խաղի մոդելավորում

Իրականացնել SnakeGame դասը, որը մոդելավորում է «Օձ» խաղի պարզ խաղադաշտ:

Պայմաններ՝

- ✓ Խաղադաշտը ունի width × height չափեր:
- ✓ Օձի մարմինը պահվում է std::deque<std::pair<int, int>> կոնտեյներում,

որտեղ յուրաքանչյուր (x, y) զույգը ներկայացնում է մարմնի հատվածի կոորդինատը։

- ✓ Օձի գլուխը գտնվում է երկկողմանի հերթի սկզբում, իսկ պոչը՝ վերջում։
- ✓ Յուրաքանչյուր շարժումից հետո գլուխը ավելացվում է push_front() մեթոդով, իսկ պոչը հեռացվում է pop_back() մեթոդով։
- ✓ Խաղադաշտում պատահական ձևով հայտնվում է սնունդ, որը օձը կարող է «ուտել»՝ երկարելու համար։

Պահանջվում է իրականացնել՝

- ✓ SnakeGame(int width, int height, int startX, int startY) կառուցիչը, որը ստեղծում է խաղադաշտը և օձը՝ սկզբնական դիրքում մեկ հատվածով։
- ✓ Շարժման մեթոդներ՝
  - void moveUp() մեթոդը,
  - void moveDown() մեթոդը,
  - void moveLeft() մեթոդը,
  - void moveRight() մեթոդները, որոնք տեղափոխում են օձը համապատասխան ուղղությամբ։
- ✓ void print() const մեթոդը, որը արտածում է խաղադաշտը կոնսոլում՝ օգտագործելով հետևյալ նշանները՝
  - '@' – օձի գլուխ,

- 'O' – օձի մարմին,
- '.' – դատարկ բջիջ,
- '*' – սնունդ:

✓ void spawnFood() մեթոդը, որը ավելացնում է սնունդ պատահական ազատ բջջում:

Նշում՝ ապահովել, որ օձը չի դուրս գալիս խաղադաշտի սահմաններից:

Խնդիր 6. (Միջին) - Սահող պատուհանի մաքսիմում

Տրված է զանգված և պատուհանի չափը K: Գտնել յուրաքանչյուր պատուհանի առավելագույն տարրը O(N) ժամանակում՝ օգտագործելով std::deque:

ԴԱՍ 12. STD::QUEUE, ԿԻՐԱՌՈՒԹՅՈՒՆՆԵՐ

Տևողություն՝ 90 րոպե

Թեմա՝ FIFO (First-In-First-Out) սկզբունք, std::queue ադապտեր, կիրառություններ և լայնությամբ շրջանցման հիմունքներ:

1. Դասի նպատակը

Ուսանողը պետք է սովորի կիրառել հերթը սիմուլյացիոն խնդիրներում (սպասարկման սրահներ, տպիչներ) և օգտագործել այն լայնությամբ շրջանցման ալգորիթմներում (լաբիրինթոս, տիրույթի ներկում (Flood fill)) և հասկանալ, որ std::queue-ն կոնտեյներ-ա-

դապտեր է, որը լռությամբ կառուցված է std::deque-ի վրա և սահմանափակում է հասանելի գործողությունները՝ ապահովելով FIFO վարքագիծ։

2. Մեթոդական ցուցումներ

- ✓ **Ադապտեր գաղափար**: Նշել, որ std::queue-ն տվյալների պահոց չէ, այլ փաթեթավորող (wrapper) է, որը օգտագործում է std::deque (կամ այլ համապատասխան կոնտեյներ) և թույլ է տալիս աշխատել միայն սահմանափակ ինտերֆեյսով (push, pop, front)՝ թաքցնելով ավելորդ ֆունկցիոնալը։
- ✓ **FIFO:** Առաջին մտնող տարրը առաջինն է դուրս գալիս: Կարելի է օգտագործել խանութի հերթի օրինակը: Ներկայացնել, որ հերթը չի ապահովում կամայական տարրի հասանելիություն (ի տարբերություն std::vector-ին std::deque-ի). հասանելի է միայն հերթի սկզբի տարրը։ Սա կարևոր է՝ հասկանալու հերթի սահմանափակումները և տարբերությունը std::deque-ից, որտեղ հասանելի են նաև push_front() և pop_back() գործողությունները։
- ✓ **Լայնությամբ շրջանցում:** Բացատրել, թե ինչու է հերթն անհրաժեշտ լաբիրինթոսում ամենակարճ ճանապարհը գտնելու համար (ալիքային մեթոդ)։ Լայնությամբ շրջանցման ժա-

մանակ հերթը ապահովում է, որ հանգույցները մշակվեն մակարդակ առ մակարդակ։

- ✓ **Նշում:** pop() մեթոդը չի վերադարձնում արժեք, այն միայն հեռացնում է տարրը հերթից։ Արժեքը ստանալու համար պետք է կանչել front() մեթոդը, ապա՝ pop()։

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Հեշտ) - Հերթի սպասարկում

Իրականացնել CashDesk դասը, որը պահում է գնորդների հերթը std::queue<std::string> կոնտեյների միջոցով և ապահովում է հետևյալ գործողությունները՝ void addCustomer(const std::string& name) մեթոդը ավելացնում է գնորդին հերթ, void serve() մեթոդը սպասարկում է հերթի առաջին գնորդին (հեռացնում է նրան հերթից), իսկ void show() const մեթոդը արտածում է հերթում գտնվող բոլոր գնորդներին։

Նպատակը՝ հասկանալ std::queue-ի ինտերֆեյսը և դրա սահմանափակումները՝ համեմատած std::deque-ի հետ։

Խնդիր 2. (Միջին) - Թղթախաղ «Հարբեցողը»

Մոդելավորել թղթախաղ՝ օգտագործելով երկու հերթ (std::queue)։ Յուրաքանչյուր քայլում խաղացողները հանում են իրենց հերթի սկզբի քարտը, համեմատում դրանք, և ավելի մեծ քարտ ունեցող խաղացողը հաղթում է այդ քայլում ու երկու քարտերն էլ ավելաց-

նում է իր հերթի վերջում (օրինակ՝ 0 քարտը հաղթում է 9-ին)։ Խաղը ավարտվում է, երբ խաղացողներից մեկի հերթը դատարկվում է, և հաղթող է համարվում մյուս խաղացողը։ Եթե խաղը չի ավարտվում 10^6 քայլից հետո, արտածել "draw"։

Խնդիր 3. (Բարդ) - Տիրույթի ներկում

Տրված է մատրիցա, որի վանդակները պարունակում են գույներ (թվերով ներկայացված)։ Տրված է մեկնարկային կետի կոորդինատ և նոր գույն։ Կատարել տիրույթի ներկում՝ փոխելով մեկնարկային կետին կապված նույն գույն ունեցող բոլոր վանդակների գույնը նոր գույնով՝ օգտագործելով լայնությամբ շրջանցման մոտեցում։

4. Տնային հանձնարարություն

Խնդիր 4. (Միջին) - Շոկոլադի խանութ

Խանութի առջև կանգնած է n գնորդից բաղկացած հերթ։ Խանութը վաճառում է միայն շոկոլադ, և յուրաքանչյուր անգամ, երբ գնորդը մտնում է խանութ, խանութի շոկոլադի քանակը ավելանում է 1 կգ-ով։ Երբ հասնում է գնորդի հերթը, նա փորձում է գնել իրեն անհրաժեշտ քանակությամբ շոկոլադ։

Խանութի աշխատանքի կանոնները հետևյալն են.

- եթե խանութում առկա է բավարար շոկոլադ՝ գնորդի պահանջը բավարարելու համար, ապա նա գնում է իր պահանջված քանակը և հեռանում է հերթից,

- եթե շոկոլադը բավարար չէ, ապա գնորդը ոչինչ չի գնում, դուրս է գալիս խանութից և կանգնում հերթի վերջում։

Սկզբում խանութում շոկոլադ չկա (0 կգ)։

Պահանջվում է որոշել, թե յուրաքանչյուր գնորդ քանի անգամ է մտել խանութ, մինչև կարողացել է գնել իր պահանջված շոկոլադը։

5. Լրացուցիչ առաջադրանքներ

Խնդիր 5. (Հեշտ) - Գումարի հաշվարկ

Իրականացնել ֆունկցիա, որը ստանում է std::queue<int> և վերադարձնում է դրա տարրերի գումարը: Ֆունկցիայի ավարտին հերթը պետք է լինի դատարկ (բոլոր տարրերը հանվում են):

Նպատակը՝ հասկանալ, թե ինչպես է կատարվում տարրերի մշակումը և հեռացումը ցիկլի մեջ:

Խնդիր 6. (Միջին) - Տրանսպորտային հոսք

Ստեղծել խաչմերուկում մեքենաների հոսքի մոդել, որտեղ մեքենաները հերթագրվում են իրենց համարներով: Իրականացնել՝

- ✓ void addCar(const std::string& plate) մեթոդը՝ մեքենան հերթ ավելացնելու համար,
- ✓ void allowToPass() մեթոդը՝ մեկ մեքենայի անցումը թույլատրելու համար (հեռացնելով այն հերթից և արտածելով համարը),
- ✓ void printQueue() const մեթոդը, որը արտածում է սպասող մեքենաների քանակը և հերթում հաջորդ մեքենայի համարը։

Խնդիր 7. (Միջին) - Տպիչի աշխատանք

Իրականացնել տպիչի հերթ, որն ունի երկու առաջնահերթության մակարդակ: Իրականացնել՝

- ✓ երկու հերթ՝ highPriority և normalPriority,
- ✓ void addJob(const std::string& name, const std::string& priority) մեթոդը, որը ավելացնում է առաջադրանքը համապատասխան հերթում,
- ✓ void process() մեթոդը, որը արտածում է բոլոր առաջադրանքները հետևյալ կարգով՝ նախ highPriority հերթից, ապա normalPriority հերթից:

Խնդիր 8. (Բարդ) - Շրջանաձև հեռացում (Josephus Problem)

Տրված է N երեխաներից կազմված խումբ, որոնք կանգնած են շրջանով: Հաշվելով սկսած առաջին երեխայից՝ յուրաքանչյուր K-րդ երեխան հեռացվում է շրջանից, մինչև բոլոր երեխաները հեռացվեն: Գրել ծրագիր, որը արտածում է երեխաների հեռացման հերթականությունը՝ օգտագործելով հերթ (std::queue):

Օրինակ ՝N=5, K=3 -> Պատասխան՝ 3, 1, 5, 2, 4:

Խնդիր 9. (Բարդ) - Կարճագույն ուղի լաբիրինթոսում

Տրված է N × M չափերի լաբիրինթոս, որտեղ '.' ազատ վանդակ է, իսկ X՝ պատ: Տրված են մեկնարկային և վերջնական կետերի կոորդինատները: Գրել ծրագիր, որը գտնում է ամենակարճ ուղին մեկնարկային կետից

մինչև վերջնական կետ՝ օգտագործելով լայնությամբ շրջանցման մոտեցում:

Խնդիր 10. (Միջին) - Հեռավորությունների աղյուսակ

Տրված է 0 և 1 թվերից կազմված մատրիցա: Յուրաքանչյուր վանդակի համար գտնել հեռավորությունը մինչև մոտակա 1-ը՝ օգտագործելով լայնությամբ շրջանցման մոտեցում:

Խնդիր 11. (Հեշտ) - Գրանցման պատուհան

Ստեղծել համակարգ մասնակիցների (Participant) գրանցման համար: Յուրաքանչյուր մասնակից ունի name և ID: Օգտագործել std::queue<Participant> սպասողներին պահելու համար: Իրականացնել process Registration() ֆունկցիա, որը հերթով գրանցում է մասնակիցներին և ազատում հերթը:

6. Մեթոդական ցուցում

Նշել, որ std::queue-ը լռությամբ օգտագործում է std::deque-ը որպես հիմք, սակայն կարող է կառուցվել նաև այլ կոնտեյների վրա (օրինակ՝ std::list), եթե այն ապահովում է անհրաժեշտ գործողությունները:

ԴԱՍ 13. ՀԵՐԹԻ ԻՐԱԿԱՆԱՑՈՒՄԸ ԿԱՊԱԿՑՎԱԾ ՑՈՒՑԱԿԻ ԵՎ ՑԻԿԼԻԿ ՋԱՆԳՎԱԾԻ ՄԻՋՈՑՈՎ

Տևողություն՝ 90 րոպե

Թեմա՝ Հերթ (Queue)՝ հիմնված միակապ ցուցակի և ցիկլիկ զանգվածի վրա։

1. Դասի նպատակը

Իրականացնել հերթ կապակցված ցուցակի միջոցով (դինամիկ հիշողությամբ) և զանգվածի հիման վրա՝ օգտագործելով ցիկլիկ ինդեքսավորում։ Զարգացնել front և rear ցուցիչների (կամ ինդեքսների) ճիշտ կառավարման հմտությունները և հասկանալ տարբեր իրականացումների առավելություններն ու սահմանափակումները։

2. Մեթոդական ցուցումներ

- ✓ **Ընդհանուր գաղափար**: Հերթը կարելի է իրականացնել տարբեր տվյալների կառուցվածքների վրա՝ կախված պահանջներից (դինամիկություն, հիշողություն, արագագործություն)։
- ✓ **Կապակցված ցուցակով իրականացում.**
  - **Սկիզբ և վերջ:** Հերթից տարրը հեռացվում է սկզբից (front), իսկ ավելացվում է վերջից (rear)։
  - **O(1) բարդություն:** Շեշտել, որ rear ցուցիչի առկայության դեպքում տարրի ավելացումը կատարվում է հաստատուն ժամանակում O(1), հակառակ դեպքում անհրաժեշտ կլիներ անցնել ամբողջ կառուցվածքով՝ O(N)։
  - **Հիշողության կառավարում:** Հիշեցնել, որ յուրաքանչյուր pop գործողության ժամա-

նակ անհրաժեշտ է ազատել համապատասխան հանգույցը (delete)՝ հիշողության արտահոսքից խուսափելու համար։

- **Սահմանային դեպքեր:** Մշակել այն իրավիճակը, երբ հերթում մնում է միայն մեկ տարր և այն հեռացվում է (այդ դեպքում և՛ front-ը, և՛ rear-ը պետք է դառնան nullptr)։

✓ **Ցիկլիկ զանգվածով իրականացում.**

- **Ինչու՞ ցիկլիկ:** Զանգվածում տարրը սկզբից հեռացնելիս մնացած բոլոր տարրերը պետք է տեղափոխվեն ձախ (պահանջում է O(N) տեղափոխում), իսկ ցիկլիկ զանգվածում պարզապես փոխվում է frontIndex-ը (O(1))։
- **Ցիկլիկ ինդեքսավորում:** Շեշտել (index + 1) % capacity բանաձևի կարևորությունը՝ ինդեքսների «շրջապտույտ» ապահովելու համար։
- **Ցուցիչների կառավարում**: Ներկայացնել frontIndex, rearIndex, size փոփոխականների դերը։
- **Դինամիկ վերաբաշխում**: Ընդգծել, որ զանգվածը լցվելու դեպքում անհրաժեշտ է նոր հիշողություն հատկացնել և տարրերը վերադասավորել՝ պահպանելով ճիշտ հերթականությունը (FIFO կարգը)։

3. Առաջադրանքներ լսարանային աշխատանքի համար

Մաս 1. Կապակցված ցուցակով հերթ

Խնդիր 1. (Հեշտ) - Կառուցվածքի սահմանում

Սահմանել Node կառուցվածքը և LinkedQueue դասը: Հանգույցը պետք է պարունակի տվյալ (int data) և հղում հաջորդին (Node* next): Node* next սկզբնարժեքավորել nullptr-ով: Դասի մեջ սահմանել front և rear ցուցիչները և սկզբնարժեքավորել դրանք nullptr-ով:

Խնդիր 2. (Միջին) - Տարրի ավելացում և հեռացում

Իրականացնել հիմնական մեթոդները՝

- ✓ void push(int val) մեթոդը, որը ավելացնում է նոր տարր հերթի վերջում և մշակում է այն դեպքը, երբ հերթը դատարկ է,
- ✓ void pop() մեթոդը, որը հեռացնում է տարրը հերթի սկզբից՝ համապատասխանաբար թարմացնելով front և rear ցուցիչները և ազատելով հիշողությունը:

Խնդիր 3. (Հեշտ) - Առաջին տարրի ստացում և դատարկության ստուգում

Իրականացնել՝

- ✓ bool isEmpty() const մեթոդը, որը վերադարձնում է true, եթե հերթում տարր չկա,
- ✓ int peek() const մեթոդը, որը վերադարձնում է հերթի առաջին (front) հանգույցի արժեքը՝ ա-

ռանց այն հեռացնելու, և մշակում է այն դեպքը, երբ հերթը դատարկ է։

Մաս 2. Ցիկլիկ զանգվածով հերթ

Խնդիր 4. (Հեշտ) - Կառուցվածքի հիմք

Տրված է CircularQueue դասի հենքը։ Իրականացնել կառուցիչը և փլուզիչը։

Խնդիր 5. (Միջին) - Ավելացում

Իրականացնել void push(const T& value) գործողությունը՝ հերթի վերջում տարր ավելացնելու համար։ Տարրը պետք է տեղադրվի հերթի վերջում՝ օգտագործելով ցիկլիկ ինդեքսավորում։

Խնդիր 6. (Միջին) - Հեռացում

Իրականացնել void pop() գործողությունը՝ հերթի սկզբից տարրը հեռացնելու համար։ Եթե հերթը դատարկ է, պետք է ապահովել համապատասխան ստուգում։ Հեռացման ընթացքում frontIndex-ը պետք է թարմացվի ցիկլիկ ձևով, և size-ը նվազեցվի։

Խնդիր 7. (Հեշտ) - Առաջին տարրի ստացում

Իրականացնել T& front() գործողությունը, որը վերադարձնում է հերթի առաջին տարրը՝ առանց այն հեռացնելու։

4. Տնային հանձնարարություն

Խնդիր 8. (Միջին) - Չափի ստացում և արտածում (LinkedQueue)

Ավելացնել size փոփոխական, որը կթարմացվի յուրաքանչյուր push և pop գործողության ժամանակ։

Իրականացնել display() ֆունկցիա, որը կարտածի հերթի բոլոր տարրերը՝ սկզբից մինչև վերջ։

Խնդիր 9. (Բարդ) - Փլուզիչ (LinkedQueue)

Իրականացնել փլուզիչ, որը ազատում է բոլոր հանգույցները։

Խնդիր 10. (Բարդ) - Հերթի շրջում (LinkedQueue)

Իրականացնել ֆունկցիա, որը փոխում է հերթի տարրերի հաջորդականությունը հակառակ ուղղությամբ։

Հուշում՝ կարելի է օգտագործել ժամանակավոր պահունակ կամ լուծել ռեկուրսիվ եղանակով։

Խնդիր 11. (Բարդ) - Ամբողջական իրականացում (CircularQueue)

Ամբողջացնել CircularQueue դասը։ Իրականացնել դինամիկ հիշողության վերաբաշխում (resize()), երբ զանգվածը լցված է, ապահովելով տարրերի ճիշտ հերթականությունը (FIFO), և print() մեթոդ, որը արտածում է տարրերը հերթականությամբ։

Խնդիր 12. (Միջին) - Հերթի պատճենում

Իրականացնել խորը պատճենում մի հերթից մյուսը։

5. Մեթոդական ցուցում

Այս թեմայում ուսանողները հաճախ սխալվում են ցուցիչների և ինդեքսների կառավարման մեջ։ Խրախուսել գրաֆիկական պատկերում (սխեմաներ, նկարներ)՝ նախքան կոդ գրելը։

ԴԱՍ 14. STD::STACK, ԿԻՐԱՌՈՒԹՅՈՒՆՆԵՐ ՊԱՐԶ ԽՆԴԻՐՆԵՐՈՒՄ

Տևողություն՝ 90 րոպե

Թեմա՝ std::stack-ի գործողությունները, տվյալների շրջում, թվային համակարգեր և պարզ սիմուլյացիաներ։

1. Դասի նպատակը

Ուսանողը պետք է տիրապետի պահունակի հիմնական մեթոդներին (push, pop, top) և հասկանա LIFO սկզբունքի կիրառությունը պարզ խնդիրներում՝ տվյալների շրջում, թվային ներկայացումներ և տողերի մշակման հիմունքներ։

2. Մեթոդական ցուցումներ

- ✓ **LIFO-ի պատկերավորում:** Ցույց տալ, որ պահունակում մենք հասանելիություն ունենք միայն ամենավերջին ավելացված տարրին։
- ✓ **Նշում:** Շեշտել, որ դատարկ պահունակից top() կամ pop() դատարկ պահունակի դեպքում առաջացնում են չսահմանված վարք (undefined behavior), ուստի նախ պետք է ստուգել !st.empty() պայմանը։
- ✓ **Շրջում:** Պահունակը բնական «շրջիչ» է. եթե տարրերը լցնենք պահունակի մեջ և հանենք, դրանք կլինեն հակառակ հերթականությամբ։

- ✓ **Պահունակի գործողությունների բարդություն:** Քննարկել, որ push, pop, top գործողությունները կատարվում են O(1) ժամանակում:

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Հեշտ) - Տողի շրջում և պալինդրոմի ստուգում

Տրված է տող: Օգտագործելով պահունակ՝ ստանալ տողի հակառակ տարբերակը: Ստուգել՝ արդյոք տողը պալինդրոմ է (կարդացվում է նույն ձևով ձախից աջ և աջից ձախ):

Օրինակ՝ "abba" → պալինդրոմ է

Խնդիր 2. (Միջին) - Տասնորդականից երկուական համակարգ

Տրված է տասնորդական ամբողջ թիվ: Օգտագործելով պահունակ՝ ներկայացնել այն երկուական համակարգում:

Խնդիր 3. (Հեշտ) - Հարևան կրկնվող նիշերի հեռացում

Տրված է տող: Պահանջվում է հեռացնել տողում գտնվող բոլոր հարևան նույնական նիշերի զույգերը՝ օգտագործելով պահունակ:

Սկզբում պահունակը դատարկ է: Անցնել տողի նիշերով և կիրառել հետևյալ կանոնը.

- ✓ եթե ընթացիկ նիշը համընկնում է պահունակի գագաթի նիշի հետ, ապա հեռացնել այն (pop),

- ✓ հակառակ դեպքում՝ ավելացնել պահունակում (push)։

Արդյունքում ստացվում է տող, որտեղ հարևան նույնական նիշերի զույգեր այլևս չկան։

Օրինակ՝ "abbaca" → "ca"։

Խնդիր 4. (Հեշտ) - Բլոկների աշտարակ

Օգտագործելով std::stack<char>՝ իրականացնել պարզ հրամանների մշակում.

- ✓ add x – ավելացնել x նիշը պահունակի գագաթին,
- ✓ remove – հեռացնել պահունակի գագաթի նիշը,
- ✓ show – արտածել պահունակի պարունակությունը՝ առանց այն փոփոխելու։

Նշում՝ քանի որ std::stack-ը թույլ չի տալիս ուղիղ անցնել տարրերով, show հրամանը կարելի է իրականացնել՝ օգտագործելով ժամանակավոր պահունակ (կամ այլ օժանդակ կառուցվածք), որպեսզի հիմնական պահունակը չփոխվի։

4. Տնային հանձնարարություն

Խնդիր 5. (Միջին) - Undo/Redo համակարգ

Իրականացնել տեքստի խմբագրման պարզ մոդել՝ օգտագործելով երկու պահունակ.

- ✓ մեկ պահունակ՝ կատարված գործողությունների համար,
- ✓ մեկ պահունակ՝ չեղարկված (undo) գործողությունների համար։

Ապահովել չեղարկման (undo) և վերականգնման (redo) գործողությունների իրականացում։

Խնդիր 6. (Միջին) - Միջին տարրի հեռացում

Տրված է պահունակ: Իրականացնել ֆունկցիա, որը հեռացնում է պահունակի միջին տարրը՝ օգտագործելով ժամանակավոր պահունակ:

Հուշում՝ Փորձեք ժամանակավորապես տեղափոխել տարրերը այլ կառուցվածքում:

5. Լրացուցիչ առաջադրանքներ

Խնդիր 7. (Միջին) - Պահունակի տեսակավորում

Տրված է պահունակ: Պահանջվում է տեսակավորել նրա տարրերը աճման կարգով այնպես, որ պահունակի գագաթին գտնվի ամենամեծ տարրը՝ օգտագործելով միայն մեկ լրացուցիչ պահունակ:

Հուշում՝ կարելի է օգտագործել երկրորդ պահունակ՝ այն միշտ պահելով տեսակավորված վիճակում: Յուրաքանչյուր նոր տարր ավելացնելիս ժամանակավորապես տեղափոխել երկրորդ պահունակ այն տարրերը, որոնք խանգարում են դրա ճիշտ տեղադրմանը:

Խնդիր 8. (Հեշտ) - Թվային համակարգեր (Բազա N)

Ընդլայնել խնդիր 2.-ը: Իրականացնել ֆունկցիա, որը թիվը վերածում է N հիմքով համակարգի (2 <= N <= 16): 16-ական համակարգի դեպքում օգտագործել A, B, C, D, E, F սիմվոլները:

Խնդիր 9. (Միջին) - Պահունակի պատճենում

Իրականացնել ֆունկցիա, որը պատճենում է մեկ պահունակի պարունակությունը մյուսի մեջ՝ պահպանելով տարրերի նույն հաջորդականությունը:

ԴԱՍ 15. STD::STACK, ԿԻՐԱՌՈՒԹՅՈՒՆՆԵՐ ԲԱՐԴ ԽՆԴԻՐՆԵՐՈՒՄ

Տևողություն՝ 90 րոպե

Թեմա՝ Փակագծային հաջորդականություններ, արտահայտությունների հաշվարկ, Zuma խաղի տրամաբանություն։

1. Դասի նպատակը

Ուսանողը պետք է կարողանա կիրառել պահունակը բարդ խնդիրներում՝ ներառյալ արտահայտությունների մշակում, կառուցվածքային վավերացում և արդյունավետ ալգորիթմների կառուցում։

2. Մեթոդական ցուցումներ

- **Երբ օգտագործել պահունակ:** Պահունակը կիրառվում է այն խնդիրներում, որտեղ անհրաժեշտ է հետևել «վերջին ավելացված, առաջին մշակվող» սկզբունքին կամ պահպանել միջանկյալ վիճակներ (օրինակ՝ փակագծեր, արտահայտություններ, հաջորդ տարրերի որոնում)։
- **Փակագծերի հաշվեկշիռ:** Սա պահունակի դասական կիրառությունն է։ Բացվող փակագիծը պետք է փակվի համապատասխան փակվողով։
- **Պոստֆիքս նշագրություն:** Ցույց տալ, թե ինչպես է պահունակը թույլ տալիս հաշվել արտահայտությունները առանց գործողություն-

ների առաջնահերթությունների մասին մտահոգվելու (քանի որ դրանք արդեն հաշվի են առնված դասավորության մեջ)։

- **Խորությամբ շրջանցում (DFS) և պահունակ։** Խորությամբ շրջանցումը կարելի է իրականացնել իտերատիվ ձևով՝ օգտագործելով պահունակ։ Պահունակը պահում է այն գագաթները, որոնք դեռ պետք է մշակվեն, ապահովելով խորությամբ շրջանցման տրամաբանությունը։ Գրաֆների դեպքում անհրաժեշտ է նաև պահել այցելված գագաթների հավաքածու (visited), խուսափելու համար նույն գագաթների կրկնակի մշակումից և անվերջ ցիկլերից։

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Միջին) - Ճիշտ փակագծային հաջորդականություն

Տրված է փակագծերից կազմված տող։ Ստուգել՝ արդյոք փակագծերը ճիշտ են զուգակցված և ներդրված ((), [], {})։ Պետք է հաշվի առնել թե՛ համապատասխանությունը, թե՛ ներդրվածության ճիշտ հերթականությունը։

Խնդիր 2. (Բարդ) - Պոստֆիքս արտահայտության հաշվարկ

Տրված է պոստֆիքս նշագրությամբ արտահայտություն (օրինակ՝ "2 3 4 * +")։ Օգտագործելով պահունակ՝ հաշվարկել արտահայտության արժեքը։

Հուշում՝ օպերատորի դեպքում անհրաժեշտ է օգտագործել պահունակում պահված օպերանդները։

Խնդիր 3. (Բարդ) - Zuma խաղի մոդելավորում

Տրված է գունավոր փուչիկների հաջորդականություն։ Իրականացնել նոր փուչիկների ավելացում և շղթայական «պայթեցում»՝ երբ առաջանում են 3 և ավելի նույն գույնի հարևան փուչիկների, դրանք հեռացվում են։ Հաշվի առնել, որ հեռացումից հետո կարող են առաջանալ նոր խմբեր (շղթայական ռեակցիա)։

Խնդիր 4. (Միջին) - Հաջորդ փոքր տարրը ձախից

Տրված է թվերի զանգված։ Յուրաքանչյուր տարրի համար գտնել իրենից ձախ գտնվող առաջին ավելի փոքր տարրի ինդեքսը։ Եթե այդպիսի տարր գոյություն չունի, վերադարձնել -1։

Հուշում՝ փորձել սկզբում կազմել պարզ՝ $O(N^2)$ բարդությամբ լուծում, ապա մտածել, թե ինչպես կարելի է այն բարելավել մինչև $O(N)$՝ օգտագործելով համապատասխան տվյալների կառուցվածք և պահելով միայն այն տարրերը, որոնք կարող են լինել պատասխան։

4. Տնային հանձնարարություն

Խնդիր 5. (Միջին) - Ֆայլային համակարգի սիմուլյացիա (pwd / cd)

Իրականացնել պարզ ֆայլային համակարգի սիմուլյացիա՝ օգտագործելով պահունակ (std::stack<std::string>), որը պահում է ընթացիկ ուղին (path)։

Ապահովել հետևյալ հրամանների իրականացումը՝

- ✓ cd <folder> – տեղափոխվել նշված պանակ (ավելացնել այն ուղուն),
- ✓ cd .. – վերադառնալ նախորդ պանակ (հեռացնել վերջին պանակը ուղուց),
- ✓ pwd – արտածել ընթացիկ ամբողջ ուղին՝ արմատից սկսած։

Ենթադրել, որ սկզբնական վիճակում գտնվում ենք արմատային պանակում (/)։

Օրինակ՝

Տրված հրամանների հաջորդականություն՝

cd home → cd user → cd docs → pwd → cd .. → pwd

Արտածում՝

/home/user/docs

/home/user

Նշումներ՝

- Եթե արդեն գտնվում ենք արմատում, ապա cd .. հրամանը չի փոխում ուղին։
- Ուղին արտածել ձևաչափով, օրինակ՝ /home/user/docs։
- Քանի որ std::stack-ը չի աջակցում տարրերի ուղիղ անցում, pwd հրամանը կարելի է իրականացնել՝ պատճենելով պահունակը կամ օգտագործելով օժանդակ կոնտեյներ՝ ուղին ճիշտ կարգով հավաքելու համար՝ առանց հիմնական պահունակը փոփոխելու։

5. Լրացուցիչ առաջադրանքներ

Խնդիր 6. (Միջին) – Դիտարկիչի պատմություն

Իրականացնել BrowserHistory դասը, որն ունի visit(url), back(steps) և forward(steps) մեթոդները:

Հուշում՝ օգտագործել երկու պահունակ՝ մեկը նախորդ էջերի, մյուսը՝ հաջորդ էջերի համար:

Խնդիր 7. (Բարդ) - Ինֆիքսից պոստֆիքս

Իրականացնել ֆունկցիա, որը սովորական մաթեմատիկական արտահայտությունը (օրինակ՝ (A+B)*C) վերածում է պոստֆիքս գրառման (AB+C*):

Հուշում՝ հաշվի առնել գործողությունների առաջնահերթությունները և փակագծերը:

Խնդիր 8. (Բարդ) - Min-Stack

Իրականացնել պահունակ, որը, բացի սովորական գործողություններից, ունի նաև getMin() մեթոդ, որը վերադարձնում է պահունակի ամենափոքր տարրը O(1) ժամանակում:

Հուշում՝ կարելի է օգտագործել լրացուցիչ օժանդակ պահունակ, որը յուրաքանչյուր քայլում պահում է տվյալ պահի նվազագույն արժեքը (օրինակ՝ յուրաքանչյուր push գործողության ժամանակ օժանդակ պահունակում պահպանել ընթացիկ նվազագույնը):

Խնդիր 9. (Միջին) - HTML թեգերի վավերացում

Տրված է տող, որը պարունակում է HTML թեգեր (օրինակ՝): Ստուգել՝ արդյոք բոլոր թեգերը ճիշտ են փակված:

Տրամաբանություն՝ նման է փակագծերի խնդրին, սակայն անհրաժեշտ է պահել թեգերի անունները։

Խնդիր 10. (Բարդ) - Ամենաերկար ճիշտ փակագծային ենթատող

Տրված է միայն (և) սիմվոլներից կազմված տող։ Գտնել ամենաերկար ենթատողի երկարությունը, որը հանդիսանում է ճիշտ փակագծային հաջորդականություն։

Օրինակ՝ ")()())" -> արդյունքը 4 է ("()()")։

Խնդիր 11. (Բարդ) – Գրաֆի խորությամբ շրջանցում (DFS) պահունակի միջոցով

Տրված է գրաֆ՝ ներկայացված հարևանության ցուցակով։ Իրականացնել գրաֆի խորությամբ շրջանցում (DFS)՝ օգտագործելով std::stack։ Արտածել գագաթները այն հերթականությամբ, որով դրանք այցելվում են։

Նշում՝ չօգտագործել ռեկուրսիա, այլ իրականացնել ալգորիթմը իտերատիվ ձևով։

ԴԱՍ 16. ՍՏԵԿԻ ԻՐԱԿԱՆԱՑՈՒՄ

Տևողություն՝ 90 րոպե

Թեմա՝ Պահունակի իրականացում հիմնված զանգվածի և կապակցված ցուցակի հիման վրա։

1. Դասի նպատակը

Ուսանողը պետք է կարողանա իրականացնել պահունակ՝ օգտագործելով զանգված և կապակցված ցուցակ։ Պետք է հասկանա LIFO սկզբունքը, գագաթի (top) դերը և տարբեր իրականացման մոտեցումների միջև տարբերությունները։

2. Մեթոդական ցուցումներ

- **Պահունակի գաղափարը։** Պահունակը տվյալների կառուցվածք է, որտեղ բոլոր գործողությունները կատարվում են միայն մեկ կետում՝ գագաթում։ push-ը ավելացնում է տարր, pop-ը հեռացնում է այն, իսկ top-ը վերադարձնում է ընթացիկ արժեքը։
- **Զանգվածով իրականացում։** Այս դեպքում տվյալները պահվում են զանգվածում, իսկ top-ը ներկայացվում է որպես ինդեքս։ Գործողությունները պարզ են և արագ, սակայն առաջանում է չափի սահմանափակման խնդիր։
- **Չափի կառավարում։** Եթե զանգվածը լցվում է, անհրաժեշտ է մեծացնել այն՝ ստեղծելով նոր, ավելի մեծ զանգված և պատճենելով հին տվյալները։ Սա կարևոր քայլ է իրականացման մեջ։
- **Կապակցված ցուցակով իրականացում։** Այստեղ պահունակը կազմված է հանգույցներից, և top-ը ցույց է տալիս առաջին հանգույցը։ Տար-

րի ավելացնումը և հեռացումը կատարվում են ցուցակի սկզբում: Այս տարբերակը դինամիկ է և չունի ֆիքսված չափ:

- **Սխալների մշակում:** Պետք է ստուգել պահունակի դատարկ լինելը pop և top գործողություններից առաջ: Ցուցակային տարբերակում կարևոր է նաև հիշողության ճիշտ ազատումը:
- **Դատարկ պահունակի վարքագիծ:** Քննարկել, թե ինչ պետք է տեղի ունենա, երբ pop կամ top կանչվում է դատարկ պահունակի վրա (բացառություն նետել կամ հաղորդել սխալ):

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Հեշտ) - ArrayStack

Իրականացնել պահունակ՝ օգտագործելով զանգված: Ավելացնել push, pop, top և isEmpty մեթոդները:

Խնդիր 2. (Միջին) - Դինամիկ մեծացում

Իրականացնել void resize() ֆունկցիան ArrayStack-ում, որը մեծացնում է զանգվածի չափը (capacity), երբ այն լցվում է:

Խնդիր 3. (Միջին) - LinkedStack

Իրականացնել պահունակ՝ օգտագործելով միակապ ցուցակ: Ավելացնել push, pop, top մեթոդները: Ապահովել հիշողության ազատումը pop գործողության ժամանակ:

Խնդիր 4. (Միջին) - Համեմատություն

Իրականացնել նույն գործողությունները երկու տարբերակներով (զանգվածով և միակապ ցուցակով իրականացված պահունակերի համար) և վերլուծել դրանց վարքը՝ ժամանակային բարդության, հիշողության օգտագործման և սահմանափակումների տեսանկյունից։

Խնդիր 5. (Բարդ) - MinStack

Իրականացնել պահունակ, որը կարող է վերադարձնել նվազագույն տարրը O(1) ժամանակում։

4. Տնային հանձնարարություն

Խնդիր 6. (Միջին) - Պահունակի պատճենում

Իրականացնել պատճենման մեխանիզմ՝ պահպանելով տարրերի հերթականությունը։

Խնդիր 7. (Բարդ) - Արտահայտության հաշվարկ

Օգտագործելով պահունակի իրականացում՝

- ✓ իրականացնել ինֆիքս արտահայտության փոխակերպումը պոստֆիքս ձևի,
- ✓ հաշվարկել ստացված պոստֆիքս արտահայտության արժեքը։

ԳԼՈՒԽ 3.
ՈՉ ԳԾԱՅԻՆ ՏՎՅԱԼՆԵՐԻ ԿԱՌՈՒՑՎԱԾՔՆԵՐ

Գլուխը նվիրված է ոչ գծային տվյալների կառուցվածքներին՝ ծառերին, բուրգերին և հեշ-աղյուսակներին: Այն ներառում է ինչպես ստանդարտ գրադարանի (STL) պատրաստի կոնտեյներների կիրառումը, այնպես էլ դրանց ներքին կառուցվածքի ինքնուրույն իրականացումը:

Գլխում ուսումնասիրվում են ոչ գծային տվյալների կառուցվածքների հիմնական թեմաները՝ երկուական ծառեր, ռեկուրսիվ ալգորիթմներ, ծառերի շրջանցումներ, երկուական որոնման ծառեր, կարմիր-սև ծառեր, բուրգեր և առաջնահերթությունների հերթեր, ինչպես նաև հեշ-աղյուսակներ և դրանց կիրառություններ: Բացի այդ, ներառված են մի շարք լրացուցիչ թեմաներ և խնդիրներ, որոնք նպաստում են նյութի խորացված ըմբռնմանը:

Գլուխը նպատակ ունի զարգացնել երկու հիմնական կարողություն. ընտրել համապատասխան տվյալների կառուցվածքը կոնկրետ խնդրի համար ու հասկանալ դրա ներքին աշխատանքը՝ մինչև իրականացման մակարդակ: Գլխի ավարտին ուսանողը կարողա-

նում է ճանաչել այն խնդիրները, որոնցում ոչ գծային կառուցվածքների կիրառումը բերում է զգալի արդյունավետության աճի։

ԴԱՍ 17. ԵՐԿՈՒԱԿԱՆ ԾԱՌ ԵՎ ՌԵԿՈՒՐՍԻԱ

Տևողություն՝ 90 րոպե

Թեմա՝ Ծառի կառուցվածք, ռեկուրսիվ ալգորիթմներ (տարրերի քանակ, գումար, ծառի բարձրություն), հայելային արտապատկերում։

1. Դասի նպատակը

Ուսանողը պետք է կարողանա կիրառել ռեկուրսիա երկուական ծառերի մշակման խնդիրներում։ Ծառերի հետ կապված բազմաթիվ խնդիրներ լուծվում են նույն ընդհանուր ռեկուրսիվ սխեմայով. ընթացիկ հանգույցի համար կատարվում է կոնկրետ գործողություն, իսկ ձախ և աջ ենթածառերը մշակվում են ռեկուրսիվ կանչերի միջոցով։ Ուսանողը պետք է հասկանա ռեկուրսիայի բազայի (nullptr ստուգում) կարևորությունը, կարողանա իրականացնել ծառի հիմնական բնութագրերը հաշվող ֆունկցիաներ (countNodes, sumNodes, treeHeight) և կիրառի ռեկուրսիան կառուցվածքային փոփոխությունների համար (հայելային արտապատկերում)։

2. Մեթոդական ցուցումներ

- ✓ **Ռեկուրսիայի բազան:** Միշտ շեշտել, որ պետք է ստուգել ընթացիկ հանգույցի գոյությունը if

(node == nullptr) return … , հակառակ դեպքում ծրագիրը կարող է վթարային ավարտ ունենալ (crash):

- ✓ **Ռեկուրսիայի ընդհանուր սխեմա և շրջանցումներ**:
  Նշել, որ երկուական ծառերի հետ կապված բազմաթիվ խնդիրներ լուծվում են նույն ռեկուրսիվ կառուցվածքով՝ ձախ ենթածառ → ընթացիկ հանգույց → աջ ենթածառ: Կախված նրանից, թե որտեղ է կատարվում գործողությունը, ստացվում են տարբեր շրջանցումներ՝ preorder (գործողությունը՝ սկզբում), inorder (գործողությունը՝ մեջտեղում) և postorder (գործողությունը՝ վերջում):
- ✓ **Պատկերավորում:** Գրատախտակին գծել փոքր ծառ և քայլ առ քայլ ցույց տալ, թե ինչպես են ռեկուրսիվ ֆունկցիաները բարձրանում ու իջնում ճյուղերով:
- ✓ **Հիշողություն:** Հիշեցնել դինամիկ ստեղծված հանգույցները հեռացնելու անհրաժեշտության մասին:

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Հեշտ) - Ծառի բնութագրեր

Իրականացնել TreeNode ստրուկտուրան երկուական ծառի հանգույցի համար, որը ներառում է տվյալ

(data), ցուցիչներ ձախ և աջ զավակների վրա: Օգտագործելով նույն ռեկուրսիվ սխեման՝ (ձախ ենթածառ → ընթացիկ հանգույց → աջ ենթածառ), իրականացնել հետևյալ ֆունկցիաները՝

- ✓ int countNodes(TreeNode* root) ֆունկցիան, որը վերադարձնում է ծառում գտնվող հանգույցների ընդհանուր քանակը,
- ✓ int sumNodes(TreeNode* root) ֆունկցիան, որը վերադարձնում է բոլոր հանգույցների արժեքների գումարը,
- ✓ int treeHeight(TreeNode* root) ֆունկցիան, որը վերադարձնում է ծառի բարձրությունը (ամենաերկար ճանապարհը արմատից մինչև տերև):

Խնդիր 2. (Միջին) - Տերևների քանակ

Տրված է երկուական ծառ: Պետք է իրականացնել int countLeaves(Node* root) ֆունկցիան, որը հաշվում և վերադարձնում է ծառում գտնվող տերև հանգույցների ընդհանուր քանակը, որտեղ տերև է համարվում այն հանգույցը, որի համար left == nullptr && right == nullptr:

Խնդիր 3. (Բարդ) - Ծառի արտացոլանք

Տրված է երկուական ծառ: Պետք է իրականացնել mirrorTree(root) ֆունկցիան, որը յուրաքանչյուր հանգույցի համար տեղերով փոխում է ձախ և աջ զավակների ցուցիչները (այսինքն՝ ծառը դարձնում է իր հայելային պատկերը):

Հարց՝ ի՞նչ կլինի, եթե ծառը երկու անգամ «հայելային» դարձնենք:

4. Տնային հանձնարարություն

Խնդիր 4. (Միջին) - Նույնական ծառեր

Իրականացնել ֆունկցիա, որը ստուգում է՝ արդյոք երկու ծառերը նույնական են, այսինքն՝ ունեն նույն կառուցվածքը և համապատասխան հանգույցներում նույն արժեքները:

Խնդիր 5. (Միջին) - Մաքսիմումի և մինիմումի որոնում

Իրականացնել ֆունկցիա, որը գտնում է երկուական ծառում պարունակվող առավելագույն և նվազագույն արժեքները:

5. Լրացուցիչ առաջադրանքներ

Խնդիր 6. (Բարդ) - Ծառի հավասարակշռվածության ստուգում

Իրականացնել ֆունկցիա, որը ստուգում է՝ արդյոք ծառը հավասարակշռված է:

Նշում՝ ծառը հավասարակշռված է, եթե յուրաքանչյուր հանգույցի համար նրա ձախ և աջ ենթածառերի բարձրությունների տարբերությունը չի գերազանցում 1-ը (|h_{left} - h_{right}| <= 1):

Խնդիր 7. (Բարդ) - Արմատից մինչև տերև ճանապարհի գումար

Տրված է երկուական ծառ (binary tree) և ամբողջ թիվ S: Պետք է իրականացնել bool hasPathSum(Node* root, int sum) ֆունկցիան, որը ստուգում է՝ արդյոք գո-

յություն ունի այնպիսի ճանապարհ արմատից մինչև որևէ տերև, որի վրա գտնվող հանգույցների արժեքների գումարը հավասար է S-ի: Ֆունկցիան պետք է վերադարձնի true, եթե նման ճանապարհ գոյություն ունի, հակառակ դեպքում՝ false:

Խնդիր 8. (Բարդ) - Ծառի շեղվածության ստուգում

Տրված է երկուական ծառ: Ստուգել՝ արդյոք այն շեղված է (skewed), այսինքն՝ յուրաքանչյուր հանգույց ունի առավելագույնը մեկ զավակ (կամ միայն ձախ, կամ միայն աջ):

Նշում՝ նման դեպքում ծառը կառուցվածքային առումով համարժեք է կապակցված ցուցակի, քանի որ չունի ճյուղավորում:

ԴԱՍ 18. ԾԱՌԻ ՇՐՋԱՆՑՈՒՄՆԵՐ

Տևողություն՝ 90 րոպե

Թեմա՝ Ծառի խորությամբ ու լայնությամբ շրջանցումներ:

1. Դասի նպատակը

Հասկանալ տարբերությունը խորությամբ և լայնությամբ շրջանցումների միջև:

2. Մեթոդական ցուցումներ

- ✓ **Խորությամբ շրջանցում:** Ներկայացնել ռեկուրսիվ իրականացման ընթացքում ֆունկցիաների կանչի պահունակի աշխատանքը:

- ✓ **Լայնությամբ շրջանցում:** Պարզաբանել հերթի դերը՝ որպես մեխանիզմ, որը հնարավորություն է տալիս պահպանել հաջորդ մակարդակի հանգույցները, մինչ մշակվում են ընթացիկ մակարդակի հանգույցները:
- ✓ **Postorder շրջանցման կիրառությունը:** Նշել, որ postorder շրջանցումը ծառի ջնջման բնական և անվտանգ եղանակն է, քանի որ նախ մշակվում (կամ ջնջվում) են զավակ հանգույցները, ապա՝ ծնողը:

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Հեշտ) - Խորությամբ շրջանցման եռյակ

Իրականացնել հետևյալ ֆունկցիաները՝

- ✓ void preorder(TreeNode* root),
- ✓ void inorder(TreeNode* root),
- ✓ void postorder(TreeNode* root):

Արտածել արդյունքները և համեմատել դրանք:

Խնդիր 2. (Միջին) - Լայնությամբ շրջանցում

Օգտագործելով std::queue<TreeNode*>, արտածել ծառը ըստ մակարդակների:

Մեթոդ՝ root-ը դնել հերթի մեջ -> քանի դեռ հերթը դատարկ չէ՝ հանել տարրը, արտածել այն և հերթի մեջ դնել նրա ձախ ու աջ զավակներին:

Խնդիր 3. (Բարդ) - Զիգզագով շրջանցում

Արտածել ծառը մակարդակներով՝ փոխելով ուղ-

ղությունը. առաջին մակարդակը՝ ձախից աջ, երկրորդը՝ աջից ձախ և այլն։

Հուշում՝ օգտագործել երկու պահունակ կամ երկկողմանի հերթ։

4. Տնային հանձնարարություն

Խնդիր 4. (Միջին) - K-րդ մակարդակի հանգույցներ

Տրված է երկուական ծառ և ամբողջ թիվ k։ Պետք է իրականացնել int countLevel(TreeNode* root, int k) ֆունկցիան, որը հաշվում և վերադարձնում է ծառի k-րդ մակարդակում գտնվող հանգույցների քանակը։ Ենթադրվում է, որ արմատը գտնվում է 0-րդ մակարդակում։

Խնդիր 5. (Միջին) - Ծառի մաքրում

Տրված է երկուական ծառ։ Պետք է իրականացնել void clearTree(TreeNode* root) ֆունկցիան, որը ռեկուրսիվ կերպով հեռացնում է ծառի բոլոր հանգույցները հիշողությունից՝ օգտագործելով postorder շրջանցում (նախ՝ ձախ ենթածառը, ապա՝ աջը, և վերջում՝ ընթացիկ հանգույցը)։ Ֆունկցիան պետք է ամբողջությամբ ազատի ծառի կողմից զբաղեցված հիշողությունը։

5. Լրացուցիչ առաջադրանքներ

Խնդիր 6. (Միջին) - Գտել հաջորդ տարրը

Տրված է հանգույցի ցուցիչը։ Գտնել դրան հաջորդող տարրը ըստ inorder շրջանցման (առանց ամբողջ ծառը շրջանցելու, եթե ծառի հանգույցն ունի հղում ծնողի վրա)։

Խնդիր 7. (Միջին) - Տրամագիծ

Տրված է երկուական ծառ: Պետք է գտնել ծառի տրամագիծը, այսինքն՝ երկու տերևների միջև եղած ամենաերկար ճանապարհի երկարությունը: Երկարությունը սահմանվում է որպես այդ ճանապարհում գտնվող հանգույցների քանակը:

ԴԱՍ 19. ԵՐԿՈՒԱԿԱՆ ԾԱՌԻ ԻՏԵՐԱՏՈՐԻ ԻՐԱԿԱՆԱՑՈՒՄ

Տևողություն՝ 90 րոպե

Թեմա՝ Ծառի իտերատորի իրականացում պահունակի միջոցով:

1. Դասի նպատակը

Սովորել ծառի շրջանցումը իրականացնել առանց ռեկուրսիայի և կառուցել իտերատոր, որը համատեղելի է range-based for (for (auto x : tree)) կառուցվածքի հետ:

2. Մեթոդական ցուցումներ

- ✓ **Պահունակի դերը**: Պահունակը օգտագործվում է շրջանցման ընթացքում նախորդ հանգույցների (ancestor) ուղին պահելու համար և փոխարինում է ռեկուրսիվ կանչերի պահունակին:
- ✓ **Իտերատորի ձևանմուշ:** Բացատրել operator++, operator* և operator!= գործողությունների աշ-

խատանքը: Իտերատորը պահում է իր վիճակը պահունակում (std::stack<TreeNode*>), որը ներկայացնում է ծառի այն ուղին, որով պետք է շարժվել հաջորդ տարրին հասնելու համար: operator*()-ը վերադարձնում է պահունակի վերին հանգույցի արժեքը, իսկ operator++()-ը հանում է այդ հանգույցը և, եթե կա աջ ենթածառ, պահունակում ավելացնում է դրա ձախագույն ճյուղը՝ ապահովելով inorder հաջորդականությունը: operator!=-ը համեմատում է երկու իտերատորների վիճակները և օգտագործվում է որոշելու համար՝ ավարտվե՞լ է արդյոք շրջանցումը:

- ✓ **Սկիզբ և ավարտ:** begin() մեթոդը պետք է սկզբնարժեքավորի իտերատորը՝ պահունակում ավելացնելով արմատից մինչև ամենաձախ հանգույցի ամբողջ ուղին, իսկ end() մեթոդը պետք է ներկայացվի որպես դատարկ պահունակով իտերատոր, որը ցույց է տալիս շրջանցման ավարտը:
- ✓ **Բարդություն և հատկություններ:** Այս մոտեցումը ապահովում է շրջանցում՝ միջինում O(1) ժամանակ մեկ քայլի համար (ամորտիզացված), իսկ օգտագործվող հիշողությունը O(h) է, որտեղ h-ը ծառի բարձրությունն է: Պահունակի կառուցվածքը պահպանում է այն պայմա-

նը, որ վերին տարրը միշտ հաջորդ վերադարձվող հանգույցն է, ինչը ապահովում է ալգորիթմի ճշտությունը։

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Միջին) - Իտերատիվ inorder

Իրականացնել իտերատիվ inorder շրջանցումը std::stack-ի միջոցով։

Խնդիր 2. (Բարդ) - Երկուական ծառի իտերատոր

Իրականացնել InorderIterator դասը և իրականացնել՝

- ✓ InorderIterator(TreeNode* root) կառուցիչը, որը սկզբնարժեքավորում է իտերատորը՝ պահունակում ավելացնելով արմատից սկսվող ձախագույն ճյուղը,
- ✓ InorderIterator& operator++() օպերատորը, որը անցնում է հաջորդ հանգույցին inorder կարգով,
- ✓ int operator*() const օպերատորը, որը վերադարձնում է ընթացիկ հանգույցի արժեքը,
- ✓ bool operator!=(const InorderIterator& other) const օպերատորը, որը թույլ է տալիս համեմատել իտերատորները և օգտագործել ցիկլում։

Խնդիր 3. (Միջին) - begin() և end()

Ավելացնել մեթոդներ ծառի դասի մեջ, որպեսզի հնարավոր լինի գրել for (auto x : myTree)։

- ✓ begin()-ը պետք է սկսի շրջանցումը՝ պահունակում ավելացնելով արմատից մինչև ամենաձախ հանգույցի ուղին,
- ✓ end()-ը պետք է ներկայացվի որպես դատարկ պահունակով իտերատոր:

4. Տնային հանձնարարություն

Խնդիր 4. (Միջին) - Գտնել հաջորդ տարրը

Տրված է հանգույցի ցուցիչ: Գտնել դրան հաջորդող տարրը ըստ inorder շրջանցման (առանց ամբողջ ծառը շրջանցելու, օգտագործելով միայն ենթածառի տրամաբանությունը):

Խնդիր 5. (Միջին) - Preorder իտերատոր

Իրականացնել նույն իտերատորի տրամաբանությունը preorder շրջանցման համար: Preorder շրջանցման դեպքում հաջորդ հանգույցին անցումը համեմատաբար ավելի պարզ է, քանի որ չի պահանջում ձախ ենթածառի նախնական ամբողջական շրջանցում:

ԴԱՍ 20. ԵՐԿՈՒԱԿԱՆ ՈՐՈՆՄԱՆ ԾԱՌ

Տևողություն՝ 90 րոպե

Թեմա՝ Երկուական որոնման ծառի հատկությունները, տեղադրում (insert), որոնում (search), վավերականության ստուգում (validation) և մինիմալ/մաքսիմալ տարրի որոնում:

1. Դասի նպատակը

Ուսանողը պետք է հասկանա երկուական որոնման ծառի հիմնական կանոնը (յուրաքանչյուր հանգույցի ձախ ենթածառի բոլոր արժեքները փոքր են տվյալ հանգույցի արժեքից, իսկ աջ ենթածառի բոլոր արժեքները՝ մեծ) և թե ինչպես է այն ապահովում O(log N) արագագործություն միջինում (բալանսավորված ծառի դեպքում) և O(N) վատագույն դեպքում:

2. Մեթոդական ցուցումներ

- **Կանոնի խստությունը:** Երկուական որոնման ծառի հիմնական կանոնը պետք է պահպանվի բոլոր հանգույցների համար. յուրաքանչյուր հանգույցի ձախ ենթածառի բոլոր արժեքները փոքր են տվյալ հանգույցից, իսկ աջ ենթածառի բոլոր արժեքները՝ մեծ: Ցանկացած խախտում նշանակում է, որ ծառը չի հանդիսանում երկուական որոնման ծառ:
- **Բալանսավորման խնդիրը:** Ցույց տալ, որ եթե թվերը հերթականությամբ ենք ներդնում (1, 2, 3, 4, …, N), ծառը վերածվում է գծային կառուցվածքի (կապակցված ցուցակի), և գործողությունների բարդությունը դառնում է O(N):

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Հեշտ) - Տեղադրում և որոնում

Իրականացնել TreeNode* insert(TreeNode* root, int

val) և TreeNode* search(TreeNode* root, int val) ֆունկցիաները: Եթե ներդրվող արժեքը արդեն գոյություն ունի, այն չավելացնել:

Խնդիր 2. (Միջին) - Ծառի վավերականության ստուգում

Իրականացնել bool isValidBST(TreeNode* root) ֆունկցիան, որը վերադարձնում է true, եթե ծառը երկուական որոնման ծառ է, հակառակ դեպքում՝ false:

Կարևոր՝ միայն անմիջական զավակներին համեմատելը բավարար չէ:

Խնդիր 3. (Հեշտ) - Մինիմում և մաքսիմում

Իրականացնել TreeNode* findMinBST(TreeNode* root) և TreeNode* findMaxBST(TreeNode* root) ֆունկցիաները՝ օգտագործելով այն փաստը, որ մինիմալ արժեքը գտնվում է ամենաձախ, իսկ մաքսիմալ արժեքը ամենաաջ հանգույցում:

Խնդիր 4. (Միջին) - Տարրերի քանակ միջակայքում

Իրականացնել int countInRange(TreeNode* root, int low, int high) ֆունկցիան, որը հաշվում է այն հանգույցների քանակը, որոնց արժեքները գտնվում են [low, high] միջակայքում: Պետք է օգտագործել երկուական որոնման ծառի հատկությունները՝ բաց թողնելով այն ճյուղերը, որոնց արժեքները դուրս են նշված միջակայքից:

4. Տնային հանձնարարություն

Խնդիր 5. - Տեսակավորված զանգվածից դեպի երկուական որոնման ծառ

Տրված է աճման կարգով տեսակավորված զանգված: Կառուցել մաքսիմալ հավասարակշռված երկուական որոնման ծառ:

Հուշում՝ յուրաքանչյուր ենթազանգվածի համար որպես արմատ ընտրել միջին տարրը:

Խնդիր 6. (Միջին) - Ծառի վերածումը տեսակավորված զանգվածի

Իրականացնել ֆունկցիա, որը երկուական որոնման ծառի բոլոր տարրերը ավելացնի std::vector<int>-ի մեջ այնպես, որ վեկտորը լինի տեսակավորված:

Նպատակը՝ հասկանալ Inorder շրջանցման և երկուական որոնման ծառի կապը:

5. Նշում

Երկուական որոնման ծառի հիմնական գործողությունները (որոնում, տեղադրում, հեռացում) ունեն O(h) բարդություն, որտեղ h-ը ծառի բարձրությունն է: Բալանսավորված ծառի դեպքում h=O(logN), իսկ վատագույն դեպքում h=O(N):

ԴԱՍ 21. ԵՐԿՈՒԱԿԱՆ ՈՐՈՆՄԱՆ ԾԱՌԻՑ ՏԱՐՐԻ ՀԵՌԱՑՈՒՄ (DELETE), ԻՏԵՐԱՏՈՐ

Տևողություն՝ 90 րոպե

Թեմա՝ Հանգույցի հեռացում, K-րդ փոքրագույն տարր, ճանապարհի տարրերի գումար և երկուական որոնման ծառի իտերատոր։

1. Դասի նպատակը

Ուսանողը պետք է տիրապետի երկուական որոնման ծառի ամենաբարդ գործողությանը՝ հեռացմանը, և սովորի օգտագործել երկուական որոնման ծառն որպես տեսակավորված հաջորդականություն՝ inorder շրջանցման միջոցով։

2. Մեթոդական ցուցումներ

- ✓ **Հեռացման 3 դեպքերը:**
  - Տերև է -> պարզապես հեռացնել։
  - Ունի 1 զավակ -> փոխարինել զավակով։
  - Ունի 2 զավակ → գտնել inorder հաջորդը (աջ ենթածառի ամենափոքր տարրը), պատճենել նրա արժեքը ընթացիկ հանգույցում և ապա հեռացնել այդ հաջորդ հանգույցը աջ ենթածառում։
- ✓ **Կարևոր դիտարկում:** Հեռացման գործողության ժամանակ անհրաժեշտ է պահպանել երկուական որոնման ծառի կանոնը։
- ✓ **Երկուական որոնման ծառի իտերատոր:** Երկուական որոնման ծառի inorder շրջանցումը տալիս է թվերը աճման կարգով։ Իտերատորը պետք է իրականացնի այդ շրջանցումը std::stack կառուցվածքի միջոցով, ինչը թույլ է տալիս

ստանալ հաջորդ տարրը միջինում O(1) ժամանակում (ամորտիզացված), իսկ հիշողությունը O(h) է, որտեղ h-ը ծառի բարձրությունն է։

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Բարդ) - Հանգույցի հեռացում

Իրականացնել `TreeNode* deleteNode(TreeNode* root, int key)` ֆունկցիան՝ մշակելով բոլոր 3 դեպքերը (տերև, մեկ զավակ, երկու զավակ)։

Խնդիր 2. (Միջին) - k-րդ փոքրագույն տարրը

Իրականացնել `int findKthSmallest(TreeNode* root, int k)` ֆունկցիան, որը վերադարձնում է երկուական որոնման ծառի k-րդ փոքրագույն տարրը։ Ենթադրել, որ $1 \le k \le N$։

Հուշում՝ օգտագործել inorder շրջանցում և հաշվիչ։

Խնդիր 3. (Բարդ) - Երկուական որոնման ծառի իտերատորի իրականացում

Ստեղծել `BSTIterator` դաս, որը ապահովում է երկուական ծառի inorder շրջանցում՝ իրականացնելով իտերատորի հետևյալ օպերատորները.

- ✓ `operator++` - անցում հաջորդ տարրին,
- ✓ `operator*` - ընթացիկ տարրի արժեքի ստացում,
- ✓ `operator!=` - համեմատություն (ցիկլում օգտագործելու համար)։

Ներքին կառուցվածքում օգտագործել `std::stack<TreeNode*>`՝ ծառի ձախ ճյուղը պահելու համար, ինչը

հնարավորություն է տալիս ստանալ հաջորդ տարրը ամորտիզացված O(1) ժամանակում։

4. Տնային հանձնարարություն

Խնդիր 4. (Բարդ) - Ճանապարհի գումար

Իրականացնել `bool hasPathSum(TreeNode* root, int sum)` ֆունկցիան, որը ստուգում է՝ կա՞ արդյոք ուղի արմատից տերև, որի տարրերի գումարը հավասար է տրված թվին։

Խնդիր 5. (Բարդ) - Միջակայքի գումար

Իրականացնել `int rangeSumBST(TreeNode* root, int low, int high)` ֆունկցիան, որը վերադարձնում է [low, high] միջակայքում գտնվող բոլոր հանգույցների արժեքների գումարը։ Պետք է օգտագործել երկուական որոնման ծառի հատկությունները՝ բաց թողնելով ոչ պիտանի ճյուղերը։

ԴԱՍ 22. ԿԱՐՄԻՐ-ՍԵՎ ԾԱՌ

Տևողություն՝ 90 րոպե

Թեմա՝ Կարմիր-սև ծառերի հատկությունները, պտույտներ (rotations), վերաներկում։

1. Դասի նպատակը

Ուսանողը պետք է հասկանա բալանսավորման մեխանիզմը, թե ինչպես են կարմիր-սև ծառերի կանոնները սահմանափակում ծառի բարձրությունը, և

հասկանա, թե ինչպես են կարմիր-սև ծառերը ապահովում O(logN) բարդություն բոլոր հիմնական գործողությունների համար, ինչպես նաև դրանց կիրառությունը std::set կոնտեյներում։

2. Մեթոդական ցուցումներ

- ✓ **Կարմիր-սև ծառերի կանոններ:**
  - Յուրաքանչյուր հանգույց կամ կարմիր է, կամ սև։
  - Արմատը սև է։
  - Բոլոր NIL (դատարկ) տերևները համարվում են սև։
  - Կարմիր հանգույցը չի կարող ունենալ կարմիր զավակներ։
  - Յուրաքանչյուր հանգույցից դեպի բոլոր NIL տերևներ տանող ճանապարհներում սև հանգույցների քանակը նույնն է (սև բարձրություն)։
- ✓ **Պտույտներ (LL, LR, RL, RR):** Բացատրել, թե ինչպես են ձախ և աջ պտույտները վերադասավորում հանգույցները՝ պահպանելով երկուական որոնման ծառի հատկությունը և վերականգնելով բալանսը։
- ✓ **Հորեղբոր կանոն (Uncle Rule):** Ներդրման ժամանակ որոշումը (վերաներկել, թե պտտել) կախված է հորեղբոր գույնից։

- ✓ **Հիմնական հատկություն:** Կարմիր-սև ծառերը ապահովում են ծառի բարձրության սահմանափակում՝ O(logN), ինչի արդյունքում բոլոր հիմնական գործողությունները ունեն նույն բարդությունը:

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Միջին) - Կարմիր-Սև ծառի քայլ առ քայլ պատկերում

Տրված է հաջորդականություն՝ 10, 20, 30, 15, 25, 5: Կառուցել կարմիր-սև ծառը՝ տարրերը տեղադրելով նշված հերթականությամբ: Յուրաքանչյուր տեղադրումից հետո պատկերել ծառի կառուցվածքը և նշել հանգույցների գույները: Նշել, թե որ դեպքն է առաջանում (RR, LL, LR, RL), և կատարել համապատասխան պտույտները: Անհրաժեշտության դեպքում նշել վերաներկման քայլերը:

Ստուգում՝ վերջնական ծառի համար համոզվել, որ բավարարվում են կարմիր-սև ծառի բոլոր կանոնները, և արմատը սև է:

Խնդիր 2. (Հեշտ) - Չկրկնվող տարրեր և հատում

Իրականացնել `vector<int> removeDuplicates(const vector<int>& nums)` և `vector<int> findIntersection(const vector<int>& a, const vector<int>& b)` ֆունկցիաները:

Հուշում՝ `std::set`-ը ավտոմատ հեռացնում է կրկնվողները և պահում է տարրերը տեսակավորված:

Խնդիր 3. (Բարդ) - Բլոկային ծածկագրերի ստուգում

Տրված է երկուական տող և թիվ k: Իրականացնել bool hasAllBinaryCodes(const string& s, int k) ֆունկցիան, որը ստուգում է՝ արդյոք տողում առկա են բոլոր հնարավոր 2^k ենթատողերը:

Լուծում՝ օգտագործել std::set<string>՝ բոլոր k երկարության ենթատողերը պահելու համար, ապա ստուգել բազմության չափը (size == 2^k):

4. Տնային հանձնարարություն

Խնդիր 4. (Միջին) - Ամենամոտ տարրը

Տրված է տեսակավորված բազմություն (կամ վեկտոր) և նպատակային արժեք: Գտնել այն տարրը, որը թվային արժեքով ամենամոտն է նպատակային արժեքին:

Հուշում՝ օգտագործել lower_bound ֆունկցիան և համեմատել ստացված դիրքի հարևան տարրերը:

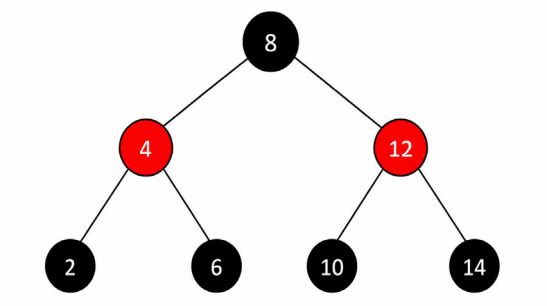
Խնդիր 5. (Միջին) - Սիմետրիկ տարբերություն

Տրված են երկու վեկտորներ: Գտնել այն տարրերը, որոնք առկա են միայն մեկում, բայց ոչ երկուսում միաժամանակ:

5. Լրացուցիչ առաջադրանքներ

Խնդիր 6. (Միջին) - Կարմիր-Սև ծառերի հատկությունների վավերացում

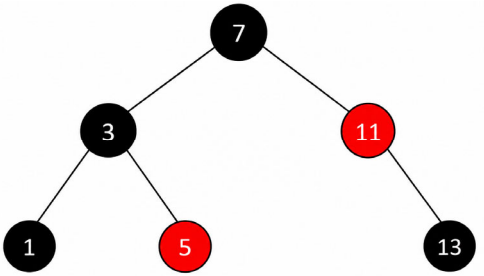
Տրված է ծառի պատկերը: Ստուգել կարմիր-սև ծառերի բոլոր 5 հատկությունները: Եթե որևէ հատկություն խախտված է, նշել՝ ո՞րը և ինչո՞ւ:



Նպատակ՝ սովորել արագ տարբերել վավեր կարմիր-սև ծառերը սովորական երկուական որոնման ծառից:

Խնդիր 7. (Միջին) - Սև բարձրության հաշվարկ

Տրված է պատրաստի կարմիր-սև ծառ: Յուրաքանչյուր հանգույցի համար հաշվել նրա «սև բարձրությունը» (քանի սև հանգույց կա տվյալ կետից մինչև տերև տանող ճանապարհին):

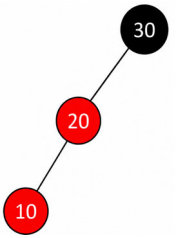


Ստուգում՝ պարզել, արդյոք ծառը բավարարում է կարմիր-սև ծառի հատկություններին, այսինքն՝ բոլոր

ճանապարհներով դեպի NIL տերևներ սև հանգույցների քանակը նույնն է:

Խնդիր 8. (Միջին) - Կրկնակի պտույտների վերլուծություն

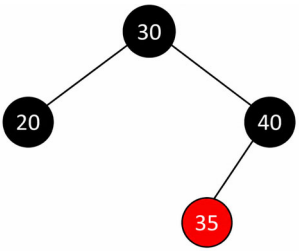
Տրված է իրավիճակ, երբ ծառի մեջ առաջացել է **LR** կամ **RL** անհավասարակշռություն:



Առաջադրանք ՝ ցույց տալ քայլ առ քայլ, թե ինչպես է կատարվում կրկնակի պտույտը և ինչպես են փոխվում հանգույցների գույները:

Խնդիր 9. (Բարդ) - Հեռացում և «Կրկնակի սև»

Տրված է RBT, որտեղից պետք է հեռացնել սև հանգույց (օրինակ՝ 20): Նկարագրել, թե որտեղ է առաջանում «կրկնակի սև» խնդիրը և ինչպես է այն լուծվում (վերաներկում կամ պտույտ):



ԴԱՍ 23. STD::SET, ԿԻՐԱՌՈՒԹՅՈՒՆՆԵՐ

Տևողություն՝ 90 րոպե

Թեմա՝ Տարրերի կրկնությունների բացակայություն, լոգարիթմական որոնում և միջակայքային հարցումներ։

1. Դասի նպատակը

Ուսանողը պետք է հասկանա, որ std::set կառուցվածքը ապահովում է տարրերի եզակիությունը և չի թույլատրում կրկնվող արժեքներ։ Նա պետք է հասկանա, թե որ խնդիրներում է std::set-ը թույլ տալիս գործողությունները կատարել O(logN) ժամանակում՝ գծային որոնման O(N) ժամանակի փոխարեն և սովորի օգտագործել lower_bound և upper_bound մեթոդները՝ տվյալների միջակայքների և ենթաբազմությունների հետ արդյունավետ աշխատելու համար։

2. Մեթոդական ցուցումներ

- ✓ **Ավտոմատ տեսակավորում:** std::set-ը միշտ պահում է տարրերը աճման կարգով, ի տարբերություն std::vector-ի, որտեղ տարրերը պահվում են ավելացման հերթականությամբ։
- ✓ **Որոնման բարդություն:** std::set-ի ներքին իրականացումը հավասարակշռված երկուական որոնման ծառ է (սովորաբար՝ կարմիր-սև ծառ), ինչի շնորհիվ հիմնական գործողու-

թյունները կատարվում են O(logN) ժամանակում:

- ✓ **Սահմանների գաղափարը:** Ուսանողներին պարզաբանել տարբերությունը.
  - lower_bound(x): Առաջին տարրը, որը մեծ կամ հավասար է x-ին:
  - upper_bound(x): Առաջին տարրը, որը խիստ մեծ է x-ից:

Այս մեթոդները վերադարձնում են իտերատորներ, ոչ թե արժեքներ:

- ✓ **Իտերատորների հետ աշխատանք:** Սա լավ հնարավորություն է ամրապնդելու իտերատորներով միջակայք նշելու հմտությունը (օրինակ՝ set::erase մեթոդում):
- ✓ **std::multiset**: Եթե անհրաժեշտ է պահել նույն արժեքի մի քանի կրկնություններ, կարելի է օգտագործել std::multiset, որը, ի տարբերություն std::set-ի, թույլ է տալիս տարրերի կրկնություններ: Այն նույնպես պահում է տարրերը տեսակավորված կարգով և տարրերի ավելացումը կատարվում է insert() ֆունկցիայով:

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Հեշտ) - Առաջին կրկնվող տարրը

Տրված է թվերի վեկտոր: Գտնել այն առաջին թիվը, որը վեկտորում հանդիպում է երկրորդ անգամ:

Խնդիր 2. (Հեշտ) - Ենթաբազմության ստուգում

Իրականացնել ֆունկցիա, որը ստանում է երկու վեկտոր՝ subset և superset: Ստուգել՝ արդյոք առաջինի բոլոր տարրերը կան երկրորդի մեջ:

Մեթոդ՝ լցնել superset-ը բազմության մեջ և set::count() կամ set::find() մեթոդով ստուգել subset-ի յուրաքանչյուր տարրը:

Խնդիր 3. (Բարդ) - Տարրերի քանակը միջակայքում

Տրված է std::set<int> և [low, high] միջակայքը: Հաշվել տարրերի քանակը՝ օգտագործելով lower_bound() և upper_bound(): Լուծումը պետք է լինի O(logN+K), որտեղ K-ն միջակայքում գտնվող տարրերի քանակն է:

Հուշում՝ օգտագործել std::distance ֆունկցիան ստացված երկու իտերատորների միջև:

4. Տնային հանձնարարություն

Խնդիր 4. (Միջին) - Բացակայող թվերի որոնում

Տրված է 1-ից N թվերի վեկտոր, որտեղ որոշ թվեր պակասում են: Գտնել բոլոր բացակայող թվերը աճման կարգով:

Քայլեր՝ լցնել վեկտորի տարրերը բազմության մեջ, ապա ցիկլով անցնել 1-ից N և ստուգել բացակայողները:

Խնդիր 5. (Բարդ) - Միջակայքի հեռացում

Իրականացնել ֆունկցիա, որը std::set-ից հեռացնում է բոլոր այն տարրերը, որոնք ընկած են [low, high] միջակայքում:

Հուշում՝ գտնել low-ի և high-ի համապատասխան իտերատորները և օգտագործել erase(it_start, it_end):

Խնդիր 6. (Հեշտ) - Բազմությունների հատում և միավորում

Իրականացնել երկու ֆունկցիա, որոնք ստանում են երկու բազմություն և վերադարձնում են նոր բազմություններ, որոնք հանդիսանում են դրանց հատումը (intersection) և միավորումը (union):

ԴԱՍ 24. STD::MAP, ԿԻՐԱՌՈՒԹՅՈՒՆՆԵՐ

Տևողություն՝ 90 րոպե

Թեմա՝ Բանալի-արժեք (key-value) զույգեր, հաճախականության հաշվարկ, խմբավորում և բարդ ասոցիացիաներ:

1. Դասի նպատակը

Ուսանողը պետք է սովորի տվյալների միջև կապեր հաստատել (օրինակ՝ բառ -> հաճախականություն) և հասկանալ, թե ինչպես է std::map կառուցվածքը ապահովում բանալի-արժեք զույգերի տեսակավորված պահպանում և O(logN) բարդությամբ գործողություններ:

2. Մեթոդական ցուցումներ

- ✓ **Օպերատոր []:** m[key] կանչելիս, եթե key բանալիով տարր գոյություն չունի, այն ավտոմատ

ավելացվում է std::map-ի մեջ լռությամբ արժեքով (կանչվում է լռությամբ կառուցիչ դասերի դեպքում և հակառակ դեպքում՝ 0)։

- ✓ **Բանալիների տիպերը:** std::map-ը թույլ է տալիս որպես բանալի օգտագործել ոչ միայն թվեր, այլ նաև տողեր (std::string) կամ այլ համեմատելի տիպեր։
- ✓ **Տեսակավորում:** std::map-ը միշտ պահում է տարրերը տեսակավորված կարգով ըստ բանալիների։
- ✓ **Իտերացիա և հեռացում:** Իտերացիայի ընթացքում տարրերի հեռացման դեպքում անհրաժեշտ է ճիշտ կառավարել իտերատորները (օրինակ՝ erase(it++) կամ std::erase_if՝ C++20-ում), հակառակ դեպքում կարող են առաջանալ անվավեր իտերատորներ։
- ✓ **Ներքին իրականացում և բարդություն:** std::map-ի ներքին իրականացումը հավասարակշռված երկուական որոնման ծառ է (սովորաբար՝ կարմիր-սև ծառ), ինչի շնորհիվ հիմնական գործողությունները կատարվում են O(logN) ժամանակում։
- ✓ **std::multimap**: Եթե անհրաժեշտ է պահել նույն բանալիով մի քանի արժեքներ, կարելի է օգտագործել std::multimap, որը, ի տարբերություն std::map-ի, թույլ է տալիս բանալիների կրկնու-

թյուններ: Այն նույնպես պահում է տարրերը տեսակավորված ըստ բանալիների, սակայն չի տրամադրում operator[], և տարրերի ավելացումը կատարվում է insert() ֆունկցիայով:

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Հեշտ) - Հաճախականության հաշվիչ

Իրականացնել std::map<int, int> countFrequency (const std::vector<int>& nums) ֆունկցիան, որը հաշվում է տրված վեկտորի տարրերի հաճախականությունները և վերադարձնում է std::map, որտեղ յուրաքանչյուր թվին համապատասխանում է նրա հանդիպումների քանակը:

Խնդիր 2. (Միջին) - Անագրամների խմբավորում

Տրված է բառերի վեկտոր (օրինակ՝ "eat", "tea", "tan", "ate"): Խմբավորել դրանք այնպես, որ անագրամները լինեն նույն ցուցակի մեջ:

Հուշում՝ որպես std::map-ի բանալի օգտագործել բառի տեսակավորված տարբերակը ("aet"):

Խնդիր 3. (Միջին) - Երկկողմանի բառարան

Իրականացնել դաս BiMap, որը ապահովում է որոնում ինչպես բանալուց արժեք, այնպես էլ արժեքից բանալի:

Հուշում՝ օգտագործել երկու std::map:

4. Տնային հանձնարարություն

Խնդիր 4. (Միջին) - Ամենաերաճախ հանդիպող բառը

Տրված է տեքստ: Գտնել այն բառը, որը հանդիպում է առավելագույն անգամ: Պետք է անտեսել մեծատառ/փոքրատառ տարբերությունը (case-insensitive) և կետադրական նշանները:

Խնդիր 5. (Միջին) - Իզոմորֆ տողեր

Տրված են երկու տողեր: Ստուգել՝ արդյոք դրանք իզոմորֆ են:

Նշում՝ երկու տողերը համարվում են իզոմորֆ, եթե առաջին տողի յուրաքանչյուր նիշ կարելի է փոխարինել այնպես, որ ստացվի երկրորդ տողը՝ պահպանելով նիշերի փոխադարձ համապատասխանությունը:

Օրինակ՝ "egg" և "add" → true:

Խնդիր 6. (Միջին) - Զանգվածի ռանգերի ձևափոխում

Տրված է թվերի զանգված: Յուրաքանչյուր տարր փոխարինել իր ռանգով՝ այն դիրքով, որը տվյալ տարրը կունենար տեսակավորված (աճման կարգով) ցուցակում: Նույն արժեք ունեցող տարրերը պետք է ստանան նույն ռանգը:

5. Լրացուցիչ առաջադրանքներ

Խնդիր 7. (Միջին) - Խմբավորում ըստ առաջին սիմվոլի

Իրականացնել std::map<char, std::vector<std::string>>

groupByFirstChar(const std::vector<std::string>& words) ֆունկցիա, որը ստանում է բառերի վեկտոր և խմբավորում է դրանք ըստ առաջին տառի։

Օրինակ՝ {"apple", "banana", "apricot"} -> 'a': ["apple", "apricot"], 'b': ["banana"]:

Խնդիր 8. (Միջին) - Երկու map-երի միաձուլում

Իրականացնել ֆունկցիա, որը ստանում է երկու std::map<std::string, int> և միացնում է դրանք։ Եթե բանալին (key) առկա է երկու std::map-երում էլ, ապա նոր std::map-ում դրանց արժեքները պետք է գումարվեն։

Խնդիր 9. (Միջին) - Առաջին չկրկնվող սիմվոլը

Տրված է տող։ Օգտագործելով std::map, գտնել առաջին սիմվոլը, որը տողում հանդիպում է ընդամենը մեկ անգամ։

Արդյունք՝ վերադարձնել այդ սիմվոլի ինդեքսը կամ -1, եթե այդպիսի սիմվոլ չկա։

Խնդիր 10. (Բարդ) - Թոփ K հաճախակի տարրերը

Տրված է թվերի վեկտոր և թիվ K։ Գտնել այն K տարրերը, որոնք ամենահաճախն են հանդիպում վեկտորում։

Հուշում՝ նախ հաշվել հաճախականությունները std::map-ի մեջ, ապա տվյալները տեղափոխել վեկտոր և սորտավորել ըստ հաճախականության։

Խնդիր 11. (Բարդ) - Միջակայքի գումար

Տրված է std::map<int, int>, որտեղ բանալիները դիրքերն են, իսկ արժեքները՝ թվերը։ Իրականացնել

ֆունկցիա, որը հաշվում է բոլոր այն տարրերի գումարը, որոնց բանալիները ընկած են [left, right] միջակայքում։

Պահանջ՝ օպտիմալացնել որոնումը՝ օգտագործելով lower_bound() մեթոդը (O(logN+K) բարդությամբ, որտեղ K-ն միջակայքում գտնվող տարրերի քանակն է)։

Խնդիր 12. (Միջին) - Ցածր հաճախականությամբ տարրերի հեռացում

Իրականացնել void removeLowFrequency (std::map<int, int>& freq, int threshold) ֆունկցիան, որտեղ std::map-ում բանալին ներկայացնում է տարրը, իսկ արժեքը՝ նրա հաճախականությունը։ Ֆունկցիան պետք է հեռացնի բոլոր այն տարրերը, որոնց հաճախականությունը փոքր է տրված շեմից։ Իտերացիայի ընթացքում տարրերի հեռացման դեպքում անհրաժեշտ է ճիշտ կառավարել իտերատորները։

ԴԱՍ 25. STD::PRIORITY_QUEUE, ԿԻՐԱՌՈՒԹՅՈՒՆՆԵՐ

Տևողություն՝ 90 րոպե

Թեմա՝ std::priority_queue (առաջնահերթություններով հերթ), մաքսիմալ (max-heap) և մինիմալ բուրգ (min-heap), K-րդ տարրի որոնում, մեդիանայի որոնում։

1. Դասի նպատակը

Ուսանողը պետք է սովորի կիրառել std::priority_queue-ն այն խնդիրներում, որտեղ անհրաժեշտ է միշտ արդյունավետ հասանելիություն ունենալ ամենամեծ (կամ ամենափոքր) տարրին՝ առանց ամբողջ զանգվածը տեսակավորելու: Ուսանողը պետք է հասկանա, որ std::priority_queue-ը թույլ է տալիս արդյունավետ կատարել գործողություններ O(logN) ժամանակում յուրաքանչյուր ավելացման կամ հեռացման համար:

2. Մեթոդական ցուցումներ

- ✓ **Բուրգ և հիշողության կույտ:** Հստակեցնել, որ այս բուրգը տվյալների կառուցվածք է և կապ չունի դինամիկ հիշողության կույտի (heap memory) հետ:
- ✓ **Մաքսիմալ բուրգ:** std::priority_queue-ը լռությամբ մաքսիմալ բուրգ է, այսինքն՝ գագաթում գտնվում է ամենամեծ տարրը:
- ✓ **Մինիմալ բուրգ գրելու ձևը:** Ուսանողների համար հաճախ բարդ է հիշել std::priority_queue<int, vector<int>, greater<int>> գրելաձևը: Բացատրել յուրաքանչյուր պարամետրի դերը:
- ✓ **Արդյունավետություն:** Ցույց տալ, որ K-րդ ամենամեծ տարրը գտնելու համար կարիք չկա տեսակավորել ամբողջ N չափանի զանգվածը:

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Հեշտ) - Բուրգային տեսակավորում (HeapSort)

Իրականացնել `vector<int> heapSort(const vector<int>& nums)` ֆունկցիան՝ օգտագործելով `std::priority_queue`: Ֆունկցիան պետք է վերադարձնի նոր վեկտոր, որը պարունակում է սկզբնական տարրերը տեսակավորված աճման կարգով:

Նշում՝ այս մոտեցումը տարբերվում է հաջորդ դասում ուսումնասիրվող in-place Heap Sort-ից. այստեղ օգտագործվում է լրացուցիչ տվյալների կառուցվածք (std::priority_queue), հետևաբար լուծումը պահանջում է O(N) լրացուցիչ հիշողություն և չի հանդիսանում in-place:

Խնդիր 2. (Միջին) - K-րդ փոքրագույն/մեծագույն տարրը

Իրականացնել `int kthSmallest(const vector<int>& nums, int k)` և `int kthLargest(const vector<int>& nums, int k)` ֆունկցիաները, որոնք վերադարձնում են համապատասխանաբար k-րդ փոքրագույն և k-րդ մեծագույն տարրերը տրված զանգվածում:

Հուշում՝ k-րդ մեծագույնը գտնելու համար օգտագործել k չափանի մինիմալ բուրգ:

Խնդիր 3. (Բարդ) - Մեդիանայի որոնում

Իրականացնել դաս `MedianFinder`, որը ապահովում է թվերի ավելացում և մեդիանայի ստացում:

Մեթոդ՝ օգտագործել երկու բուրգ՝ մաքսիմալ բուրգ (ձախ կեսի համար) և մինիմալ բուրգ (աջ կեսի համար)։

4. Տնային հանձնարարություն

Խնդիր 4. (Բարդ) - Ամենահաճախ հանդիպող K տարրերը

Տրված է ամբողջ թվերից բաղկացած վեկտոր և ամբողջ թիվ K։ Գտնել վեկտորի այն K տարրերը, որոնք հանդիպում են առավելագույն հաճախությամբ։

Հուշում՝

- ✓ օգտագործել std::map՝ տարրերի հաճախությունները հաշվելու համար,
- ✓ օգտագործել std::priority_queue՝ ամենահաճախ հանդիպող տարրերը ընտրելու համար։

Խնդիր 5. (Միջին) - Առաջադրանքների պլանավորում

Իրականացնել կառուցվածք, որը պահում է առաջադրանքներ և յուրաքանչյուր պահին վերադարձնում է ամենաբարձր առաջնահերթություն ունեցողը՝ օգտագործելով std::priority_queue։

5. Լրացուցիչ առաջադրանքներ

Խնդիր 6. (Բարդ) - K տեսակավորված ցուցակների միաձուլում

Տրված է տեսակավորված վեկտորների վեկտոր։ Օգտագործելով std::priority_queue, միավորել բոլոր ցուցակները մեկ ընդհանուր տեսակավորված վեկտորի մեջ։

- *Ինչո՞ւ է սա կարևոր*՝ սա դասական խնդիր է, որն օգտագործվում է մեծ տվյալների (Big Data) մշակման ժամանակ, երբ տվյալները չեն տեղավորվում հիշողության մեջ և պետք է մասամաս միացվեն։
- *Բարդություն*՝ O(NlogK), որտեղ N-ը բոլոր տարրերի քանակն է, իսկ K-ն՝ ցուցակների քանակը։

Խնդիր 7. (Միջին) - K մոտակա կետերը սկզբնակետին

Տրված է կոորդինատների հարթության վրա գտնվող կետերի վեկտոր (Point {int x, y}) և թիվ K: Գտնել այն K կետերը, որոնք ամենամոտն են գտնվում կոորդինատների սկզբնակետին (0, 0)։ Հեռավորությունը հաշվել որպես $x^2 + y^2$ (քառակուսի արմատը կարելի է չհաշվել՝ արագության համար)։

Հուշում՝ օգտագործել K չափանի մաքսիմալ բուրգ, որտեղ պահվում են ընտրված K կետերը։ Եթե նոր կետի հեռավորությունը փոքր է, քան բուրգի գագաթի (ամենահեռու) կետի հեռավորությունը, ապա այն փոխարինել։

ԴԱՍ 26. ՏԵՍԱԿԱՎՈՐՈՒՄ՝ ՀԻՄՆՎԱԾ ԲՈՒՐԳԻ ՎՐԱ

Տևողություն՝ 90 րոպե

Թեմա՝ Բուրգի (heap) կառուցում O(N) ժամանակում, տեսակավորում առանց լրացուցիչ հիշողության (in-place Heap Sort)։

1. Դասի նպատակը

Ուսանողը պետք է հասկանա բուրգի կառուցվածքային առավելությունները և սովորի կիրառել heapify գործողությունը զանգվածի վրա՝ առանց օժանդակ տվյալների կառուցվածքների օգտագործման։

2. Մեթոդական ցուցումներ

- ✓ **Build-Heap O(N) ժամանակում (որտեղ N-ը տարրերի քանակն է):** Նշել, թե ինչու է զանգվածի վերջից դեպի սկիզբ կատարվող siftDown (heapify) գործողությունների շարքը ունի O(N) բարդություն, մինչդեռ հերթով տարրեր ավելացնելը (siftUp) ունի O(Nlog N)։
- ✓ **In-place տրամաբանություն:** Կարևոր է ընդգծել, որ զանգվածը կարելի է բաժանել երկու մասի՝ «բուրգի մաս» և «տեսակավորված մաս»։ Յուրաքանչյուր քայլում բուրգի գագաթը (առավելագույն տարրը) տեղափոխվում է զանգվածի վերջ, և բուրգի սահմանը փոքրացվում է։
- ✓ **Ինդեքսավորում:** Նշել ինդեքսների կապը՝ i հանգույցի զավակներն են 2i + 1 և 2i + 2, իսկ ծնողը՝ (i-1)/2։

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Բարդ) - In-place Heap Sort

Իրականացնել void inPlaceHeapSort(vector<int>& arr) ֆունկցիան։

- **Քայլ 1:** Կառուցել մաքսիմալ բուրգ տրված վեկտորից O(N) ժամանակում (սկսած N/2 - 1 ինդեքսից դեպի 0)։
- **Քայլ 2:** Քանի դեռ բուրգի չափը > 1, տեղերով փոխել arr[0]-ն և վերջին տարրը, ապա կանչել heapify գագաթի համար։

Խնդիր 2. (Միջին) - K ամենափոքր տարրերը

Գտնել K ամենափոքր տարրերը O(N + KlogN) ժամանակում, որտեղ N-ը տարրերի քանակն է։

Մեթոդ՝ կառուցել մինիմալ բուրգ O(N)-ում, ապա կատարել K հատ extractMin (գագաթի հեռացում)։

Խնդիր 3. (Հեշտ) - Բուրգի վավերության ստուգում

Իրականացնել bool isMaxHeap(const vector<int>& arr) ֆունկցիան, որը ստուգում է՝ արդյոք տրված վեկտորը հանդիսանում է վավեր մաքսիմալ բուրգ։

4. Տնային հանձնարարություն

Խնդիր 4. (Միջին) - Պարանների միացման նվազագույն արժեք

Տրված են պարանների երկարություններ։ Միացնել դրանք այնպես, որ ընդհանուր ծախսը լինի նվա-

զագույն (երկու պարան միացնելու ծախսը դրանց երկարությունների գումարն է)։

Հուշում՝ յուրաքանչյուր քայլում ընտրել երկու ամենակարճ պարանները և միացնել դրանք՝ օգտագործելով մինիմալ բուրգ։

Խնդիր 5. (Բարդ) - K-րդ ամենամեծ տարրը հաջորդականությունում

Տրված է ամբողջ թվերի հաջորդականություն և K բնական թիվ։ Պահանջվում է յուրաքանչյուր նոր տարրի ավելացումից հետո որոշել և վերադարձնել տվյալ պահին առկա տարրերի բազմության K-րդ ամենամեծ տարրը՝ օգտագործելով K չափանի մինիմալ բուրգ, որի արմատը պահպանում է այդ արժեքը։

5. Լրացուցիչ առաջադրանքներ

Խնդիր 6. (Միջին) - Առաջնահերթությունների հերթ՝ համեմատման օպերատորի վերասահմանմամբ

Իրականացնել հիվանդանոցի ընդունարանի հերթի սիմուլյացիա։ Յուրաքանչյուր հիվանդ ունի անուն և «ծանրության աստիճան» (1-10)։ Օգտագործել std::priority_queue՝ համապատասխան կառուցվածքով և operator< վերասահմանմամբ այնպես, որ ամենածանր հիվանդը սպասարկվի առաջինը։

Խնդիր 7. (Բարդ) - Heapify և աստիճանական ներդրման համեմատություն

Իրականացնել ծրագիր, որը համեմատում է O(N) բարդությամբ Heapify ալգորիթմի և N հատ O(logN)

բարդությամբ տարրերի հերթական ավելացման (insertion) միջոցով բուրգ կառուցելու ժամանակը մեծ չափի տվյալների համար (օրինակ՝ 1,000,000 տարր)։

Նշում՝ In-place Heap Sort ալգորիթմը ապահովում է O(NlogN) ժամանակային բարդություն և O(1) լրացուցիչ հիշողություն, ինչը նրա հիմնական առավելություններից է։

ԴԱՍ 27. ՄԱՔՍԻՄԱԼ ԲՈՒՐԳԻ ԻՐԱԿԱՆԱՑՈՒՄ

Տևողություն՝ 90 րոպե

Թեմա՝ Բուրգի ներքին կառուցվածքը զանգվածի վրա, siftUp, siftDown և heapify (Ֆլոյդի ալգորիթմ)։

1. Դասի նպատակը

Ուսանողը պետք է հասկանա, թե ինչպես է հնարավոր երկուական ծառը ներկայացնել զանգվածի միջոցով և ինչպես են աշխատում բուրգի վերականգնման մեխանիզմները։

2. Մեթոդական ցուցումներ

- ✓ **Ինդեքսավորում:** Նշել i ինդեքսով հանգույցի զավակների (2i+1, 2i+2) և ծնողի ((i-1)/2) ինդեքսների կապը։
- ✓ **SiftUp vs SiftDown:** siftUp-ը կիրառվում է ներդրման ժամանակ (ներքևից վերև), իսկ

siftDown-ը՝ հեռացման կամ վերականգնման ժամանակ (վերևից ներքև)։

- ✓ **Heapify:** Կարևոր է ընդգծել, որ զանգվածը բուրգի վերածելը կատարվում է O(N) ժամանակում և ոչ թե O(NlogN), քանի որ ստորին մակարդակների հանգույցների մեծ մասը siftDown գործողություն չի պահանջում։

3. Առաջադրանքներ լսարանային աշխատանքի համար (Իրականացում)

Խնդիր 1. (Հեշտ) - Ինդեքսների հաշվարկ և վերև բարձրացում (SiftUp)

Տրված է բուրգի զանգվածային ներկայացում։ Իրականացնել՝

- ➢ int parent(int i) ,
- ➢ int leftChild(int i),
- ➢ int rightChild(int i),
- ➢ void siftUp(int i)

մեթոդները, որտեղ siftUp-ը վերականգնում է բուրգի հատկությունը՝ տարրը տեղափոխելով վերև մինչև իր ճիշտ դիրքը։

Խնդիր 2. (Միջին) - Տարրի ավելացում և հեռացում բուրգից

Պահանջվում է իրականացնել void push(int value) և void pop() գործողությունները, որտեղ push-ը ավելացնում է տարրը բուրգ և վերականգնում կառուցվածքը siftUp-ի միջոցով, իսկ pop-ը հեռացնում է արմատը՝

այն փոխարինելով վերջին տարրով և կիրառելով siftDown:

Խնդիր 3. (Բարդ) - Բուրգի կառուցում գծային ժամանակում

Տրված է վեկտոր: Պահանջվում է իրականացնել կառուցիչ, որը O(N) ժամանակում այն դարձնում է բուրգ՝ սկսելով վերջին ներքին հանգույցից (N/2–1) և հերթականորեն կիրառելով siftDown մինչև արմատ:

4. Տնային հանձնարարություն

Խնդիր 4. (Հեշտ) - isHeap (մաքսիմալ բուրգի ստուգում)

Տրված է ամբողջ թվերից բաղկացած վեկտոր: Իրականացնել ստատիկ մեթոդ, որը ստուգում է՝ արդյոք այդ վեկտորը ներկայացնում է վավեր մաքսիմալ բուրգ:

Հուշում՝ յուրաքանչյուր տարրի համար ստուգել, որ այն մեծ կամ հավասար է իր զավակներին (եթե դրանք գոյություն ունեն):

Խնդիր 5. (Միջին) - Priority Queue-ի իրականացում

Օգտագործելով իրականացված MaxHeap կառուցվածքը՝ իրականացնել պարզ MyPriorityQueue դաս: Դասը պետք է ապահովի հիմնական գործողությունները՝

- տարրի ավելացում (push),
- առավելագույն տարրի վերադարձ (top),

- առավելագույն տարրի հեռացում (pop),
- դատարկության ստուգում (empty)։

ԴԱՍ 28. STD::UNORDERED_SET, STD::UNORDERED_MAP, ԿԻՐԱՌՈՒԹՅՈՒՆՆԵՐ

Տևողություն՝ 90 րոպե

Թեմա՝ Ալգորիթմական խնդիրներ std::unordered_set և std::unordered_map կիրառմամբ։

1. Դասի նպատակը

Ուսանողը պետք է հասկանա std::unordered_set-ի և std::unordered_map-ի ներքին կառուցվածքը (զամբյուղներ, բեռնվածության գործակից, կոլիզիաներ) և կարողանա նպատակային կիրառել դրանք ալգորիթմական խնդիրների լուծման ժամանակ։ Մասնավորապես, ուսանողը պետք է ճանաչի այն բնորոշ խնդրաձևերը (կրկնությունների հեռացում, ենթազանգվածների գումար, անագրամների խմբավորում), որոնցում հեշ-աղյուսակի կիրառումը թույլ է տալիս նվազեցնել ժամանակային բարդությունը $O(N^2)$-ից մինչև $O(N)$՝ հավելյալ հիշողության հաշվին։

2. Մեթոդական ցուցումներ

- ✓ **Ժամանակ-հիշողություն փոխզիջում:** Շեշտել, որ հեշ-աղյուսակը ապահովում է հիմնական

գործողությունների բարձր արագություն (ամորտիզացված O(1))՝ լրացուցիչ հիշողության օգտագործման հաշվին (O(N)):

- ✓ **Բանալու ընտրություն:** Նշել, որ խնդրի արդյունավետ լուծման համար կարևոր է ճիշտ ընտրել բանալին. այն կարող է լինել թիվ, տող կամ պրեֆիքսային գումար՝ կախված խնդրի բնույթից:
- ✓ **Map և Unordered_map:** Այն դեպքերում, երբ տարրերի տեսակավորված լինելը պահանջված չէ, նպատակահարմար է օգտագործել std::unordered_map, քանի որ այն ապահովում է ամորտիզացված O(1) բարդություն՝ ի տարբերություն std::map-ի O(logN)-ի:
- ✓ **Մեկ անցմամբ լուծում:** Հեշ-աղյուսակների կիրառմամբ հաճախ հնարավոր է մշակել տվյալները մեկ անցմամբ՝ յուրաքանչյուր տարրի համար անմիջապես ստուգելով կամ թարմացնելով համապատասխան արժեքը (std::unordered_map կամ std::unordered_set): Սա թույլ է տալիս խուսափել կրկնակի ցիկլերից և ապահովել O(N) բարդությամբ լուծում:

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Հեշտ) - Տեխնիկական վարժություն

Ստեղծել std::unordered_map<string, int> կառուցվածք: Լցնել 5 քաղաքների անուններ և նրանց բնակ-

չությունը: Օգտագործել find() կամ count() մեթոդը ստուգելու համար՝ արդյոք «Yerevan»-ը կա ցուցակում: Արտածել bucket_count() և load_factor() արժեքները:

Խնդիր 2. (Հեշտ) - Կրկնությունների հեռացում

Տրված է վեկտոր: Օգտագործելով std::unordered_set՝ վերադարձնել նոր վեկտոր, որը պարունակում է միայն չկրկնվող տարրեր՝ պահպանելով դրանց առաջին հանդիպման հերթականությունը:

Խնդիր 3. (Հեշտ) - Առաջին չկրկնվող սիմվոլը

Գտնել տրված տողի առաջին չկրկնվող սիմվոլը՝ ապահովելով O(N) ժամանակային բարդություն:

Խնդիր 4. (Միջին) - Ամենաերկար հաջորդական թվերի հաջորդականությունը

Գտնել զանգվածի մեջ ամենաերկար հաջորդական թվերի հաջորդականությունը (օրինակ՝ 1, 2, 3, 4) O(N) ժամանակում:

Հուշում՝ լցնել բոլորը բազմության մեջ և սկսել հաշվել միայն այն թվից, որը շղթայի սկիզբն է (num - 1-ը չկա բազմությունում):

Խնդիր 5. (Միջին) - Զույգերի որոնում տրված տարբերությամբ

Տրված վեկտորում գտնել բոլոր այն թվային զույգերը, որոնց տարբերությունը հավասար է K-ի:

Խնդիր 6. (Միջին) - Ենթազանգվածների գումար

Գտնել այն ենթազանգվածների քանակը, որոնց գումարը հավասար է տրված K-ին՝ օգտագործելով

պրեֆիքսային գումարներ և հաճախականությունների հեշ-աղյուսակ։

Խնդիր 7. (Միջին) - Երկու թվերի գումար

Գտնել զանգվածում այն երկու տարրերի ինդեքսները, որոնց գումարը հավասար է տրված թվին։

4. Տնային հանձնարարություն

Խնդիր 8. (Հեշտ) - Առաջին չկրկնվող սիմվոլի ինդեքս

Գտնել տրված տողի առաջին չկրկնվող սիմվոլի ինդեքսը՝ օգտագործելով հաճախականությունների հաշվարկ։

Խնդիր 9. (Միջին) - Իզոմորֆ տողեր

Տրված են երկու տողեր։ Ստուգել՝ արդյոք դրանք իզոմորֆ են։ Երկու տող համարվում են իզոմորֆ, եթե առաջին տողի յուրաքանչյուր սիմվոլին կարելի է համապատասխանեցնել երկրորդ տողի միակ սիմվոլ՝ այնպես, որ պահպանվի սիմվոլների կարգը և համապատասխանությունը լինի միարժեք (մեկ սիմվոլը միշտ համապատասխանի նույն սիմվոլին, և տարբեր սիմվոլներ չհամապատասխանեն նույն սիմվոլին)։

Օրինակ՝

- "egg" և "add" → true
- "foo" և "bar" → false

Խնդիր 10. (Միջին) - Երկու զանգվածների հատում

Տրված են ամբողջ թվերից բաղկացած երկու զանգվածներ A և B։ Կառուցել նոր զանգված, որը պա-

րունակում է երկու զանգվածներին միաժամանակ պատկանող տարրերը՝ առանց կրկնությունների, O(N+M) ամորտիզացված ժամանակային բարդությամբ, որտեղ N-ը և M-ը համապատասխանաբար A և B զանգվածների չափերն են:

Խնդիր 11. (Միջին) - Անագրամի ստուգում

Տրված են երկու տողեր: Ստուգել՝ արդյոք դրանք անագրամ են՝ օգտագործելով մեկ հեշ-աղյուսակ և հաշվիչների հավասարակշռման գաղափարը:

Խնդիր 12. (Միջին) - Բառերի համապատասխանություն

Տրված է նիշերի տող (օրինակ՝ "abba") և բառերից բաղկացած տող (օրինակ՝ "dog cat cat dog"): Ստուգել՝ արդյոք բառերի հաջորդականությունը համապատասխանում է տրված նիշերի տողին:

Համապատասխանությունը ճիշտ է, եթե յուրաքանչյուր նիշին համապատասխանում է ճիշտ մեկ բառ, և յուրաքանչյուր բառ համապատասխանում է ճիշտ մեկ նիշի (այսինքն՝ համապատասխանությունը միարժեք է երկու ուղղությամբ):

Օրինակ՝

- "abba" և "dog cat cat dog" → true
- "abba" և "dog cat cat fish" → false
- "aaaa" և "dog cat cat dog" → false

5. Լրացուցիչ առաջադրանքներ

Խնդիր 13. (Միջին) - Անագրամների խմբավորում

Տրված է բառերի ցուցակ: Խմբավորել բառերը այն-

պես, որ բոլոր անագրամները հայտնվեն նույն խմբում, և վերադարձնել արդյունքը որպես vector<vector<string>>: Որպես հեշ-աղյուսակի բանալի օգտագործել բառի տեսակավորված տեսքը (օրինակ՝ "eat", "tea", "ate" → "aet") կամ տառերի հաճախականության աղյուսակ:

Խնդիր 14. (Միջին) - Անընդհատ ենթազանգվածի գումար

Տրված է ամբողջ թվերից բաղկացած զանգված և ամբողջ թիվ K: Ստուգել՝ արդյոք զանգվածում գոյություն ունի առնվազն երկու տարր պարունակող շարունակական ենթազանգված, որի տարրերի գումարը բազմապատիկ է K-ի:

Խնդիր 15. (Միջին) - Շարունակական ենթազանգված

Տրված է զանգված, որը պարունակում է միայն 0 և 1 արժեքներ: Գտնել ամենաերկար շարունակական ենթազանգվածի երկարությունը, որի մեջ 0-ների և 1-երի քանակները հավասար են՝ օգտագործելով պրեֆիքսային գումարների մոտեցում:

Օրինակ՝

- {0, 1} → արդյունք՝ 2
- {0, 1, 0} → արդյունք՝ 2
- {0, 0, 1, 0, 0, 0, 1, 1} → արդյունք՝ 6

Հուշում՝ 0-ները կարող եք դիտարկել որպես -1, և խնդիրը վերածել հետևյալին՝ գտնել ամենաերկար ենթազանգվածը, որի տարրերի գումարը հավասար է 0-ի:

Խնդիր 16. (Բարդ) - Քառյակների որոնում տրված գումարով

Տրված է ամբողջ թվերից բաղկացած զանգված և թիվ: Գտնել բոլոր եզակի քառյակները (a, b, c, d), որոնց գումարը հավասար է տրված թվին, որտեղ տարրերը վերցված են տարբեր ինդեքսներից: Խնդիրը լուծել զույգերի գումարները հեշ-աղյուսակում նախապես պահելու մոտեցմամբ:

ԴԱՍ 29. ՀԵՇ-ԱՂՅՈՒՍԱԿԻ ԻՐԱԿԱՆԱՑՈՒՄ

Տևողություն՝ 90 րոպե

Թեմա՝ Հեշավորում, կոլիզիաների լուծում, բեռնվածության գործակից և վերահեշավորում:

1. Դասի նպատակը

Ուսանողը պետք է սովորի զրոյից կառուցել հեշ-աղյուսակ՝ օգտագործելով std::vector և std::list: Հասկանա, թե ինչպես է հեշ-ֆունկցիան վերածվում զանգվածի ինդեքսի և ինչպես կառավարել աղյուսակի չափսերը:

2. Մեթոդական ցուցումներ

- ✓ **Հեշ-ֆունկցիա:** std::hash-ը սահմանված տիպերի տվյալները փոխակերպում է ամբողջ թվի (size_t), որը այնուհետև բերվում է զանգվածի սահմանների մեջ՝ կիրառելով % buckets.size() գործողությունը:

- ✓ **Կոլիզիաներ:** Նշել, որ տարբեր բանալիներ կարող են ունենալ նույն ինդեքսը: «Շղթաների մեթոդը» լուծում է այս խնդիրը՝ յուրաքանչյուր վանդակում (bucket) պահելով կապակցված ցուցակ:
- ✓ **Բեռնվածության գործակից:** Կարևոր է նշել, որ բեռնվածության գործակիցը սահմանվում է որպես տարրերի քանակի և վանդակների (bucket) քանակի հարաբերություն: Երբ այս արժեքը անցնում է որոշ շեմ (օրինակ՝ 0.75), անհրաժեշտ է կատարել վերահեշավորում՝ արդյունավետությունը պահպանելու համար:
- ✓ **Հեշ-ֆունկցիայի որակ:** Ընդգծել, որ հեշ-ֆունկցիան պետք է հավասարաչափ բաշխի բանալիները վանդակների (bucket) միջև, հակառակ դեպքում կոլիզիաները կաճեն և արդյունավետությունը կնվազի:
- ✓ **Վերահեշավորում:** Ընդգծել, որ աղյուսակի չափի մեծացման անհրաժեշտությունը առաջանում է բեռնվածության գործակցի աճի դեպքում, և վերահեշավորման ընթացքում բոլոր տարրերը վերահաշվարկվում և վերաբաշխվում են նոր, ավելի մեծ աղյուսակում՝ կիրառելով նույն հեշ-ֆունկցիան նոր վանդակների (bucket) քանակով:

- ✓ **Ժամանակային բարդություն**: Նշել, որ ներդրման, որոնման և հեռացման գործողությունները կատարվում են ամորտիզացված O(1) ժամանակում, սակայն վատ հեշ-ֆունկցիայի դեպքում կարող են դառնալ O(N):

3. Առաջադրանքներ լսարանային աշխատանքի համար (Իրականացում)

Խնդիր 1. (Միջին) - Հեշ-աղյուսակի կառուցվածք

Սահմանել HashTable<K, V> դասը՝ օգտագործելով std::vector<std::list<std::pair<K, V>>> որպես հիմնական պահոց՝ ապահովելով տարրերի պահպանումը ըստ հեշավորման սկզբունքի:

Խնդիր 2. (Միջին) - Տեղադրում և որոնում

Իրականացնել հեշ-աղյուսակում տարրի ներդրման և որոնման գործողությունները՝ ապահովելով, որ նոր բանալու դեպքում այն ավելացվի, իսկ արդեն գոյություն ունեցող բանալու դեպքում համապատասխան արժեքը թարմացվի:

Խնդիր 3. (Բարդ) - Վերահեշավորում

Իրականացնել վերահեշավորման մեխանիզմ, որը մեծացնում է աղյուսակի չափը և վերաբաշխում բոլոր տարրերը նոր ինդեքսներով՝ ապահովելով բոլոր տարրերի ճիշտ տեղափոխումը նոր աղյուսակ:

4. Տնային հանձնարարություն

Խնդիր 4. (Միջին) - Տարրի հեռացում ըստ բանալու

Իրականացնել տարրի հեռացման գործողություն

հեշ-աղյուսակից՝ ըստ տրված բանալու՝ գտնելով համապատասխան վանդակը (bucket), որոնելով տարրը շղթայում և ճիշտ կերպով հեռացնելով այն՝ պահպանելով կառուցվածքի ամբողջականությունը։

Խնդիր 5. (Միջին) - Հեշ-ֆունկցիայի արդյունավետության գնահատում

Իրականացնել size_t getMaxBucketSize() const ֆունկցիան, որը վերադարձնում է հեշ-աղյուսակի ամենաերկար շղթայի երկարությունը՝ գնահատելով հեշ-ֆունկցիայի արդյունավետությունը (փոքր արժեքի դեպքում կոլիզիաները քիչ են, մեծ արժեքի դեպքում՝ շատ)։

ԴԱՍ 30. ԲԱՑ ՀԱՍՑԵԱՎՈՐՈՒՄ

Տևողություն՝ 90 րոպե

Թեմա՝ Կոլիզիաների լուծումը առանց լրացուցիչ ցուցակների։ Գծային, քառակուսային և կրկնակի հեշավորում։

1. Դասի նպատակը

Ուսանողը պետք է հասկանա, թե ինչպես է կոլիզիայի դեպքում «ազատ տեղ» որոնվում նույն զանգվածի շրջանակներում՝ բաց հասցեավորման մեթոդներով։ Կարևոր է նաև պատկերացնել հիշողության լոկալության (cache locality) դերը նման կառուցվածքների արդյունավետության մեջ։

2. Մեթոդական ցուցումներ

- ✓ **Գծային հետազոտում:** Ներկայացնել, որ եթե h(k) դիրքը զբաղված է, ապա հաջորդաբար ստուգվում են h(k)+1, h(k)+2, … դիրքերը: Կարևոր է անդրադառնալ առաջնային կլաստերիզացիայի խնդրին, երբ տարրերը կուտակվում են հարակից դիրքերում:
- ✓ **Քառակուսային հետազոտում:** Նշել, որ կիրառվում է h(k) + i² ձևը: Այս մոտեցումը նվազեցնում է գծային կլաստերիզացիայի ազդեցությունը, սակայն ամբողջությամբ չի վերացնում այն:
- ✓ **Կրկնակի հեշավորում:** Քայլի չափը որոշվում է երկու հեշ-ֆունկցիաների միջոցով՝ $h_1(k)+i*h_2(k)$: Այս մեթոդը սովորաբար ապահովում է ավելի հավասարաչափ բաշխում և նվազեցնում կլաստերիզացիան:
- ✓ **Հետաձգված հեռացում:** Նշել, որ բաց հասցեավորման դեպքում տարրը չի հեռացվում ֆիզիկապես, այլ բջիջում նշվում է հատուկ արժեքով (օրինակ՝ DELETED): Այդպիսի բջիջը չի համարվում դատարկ, ուստի որոնումը շարունակվում է, սակայն այն կարող է կրկին օգտագործվել նոր տարրի տեղադրման համար:

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Բարդ) - Հեշ-աղյուսակի իրականացում

Իրականացնել հեշ-աղյուսակ՝ օգտագործելով բաց

հասցեավորում: Աղյուսակը պետք է ապահովի հետևյալ գործողությունները՝

- void insert(const T& key) - տարրի ավելացում,
- bool contains(const T& key) const - տարրի առկայության ստուգում,
- void remove(const T& key) - տարրի հեռացում:

Կոնֆլիկտները լուծել գծային հետազոտման մեթոդով: Հաշվի առնել բեռնվածության գործակցի ազդեցությունը և ապահովել, որ ավելացման և որոնման գործողությունները կատարվեն ամորտիզացված O(1) ժամանակում:

Խնդիր 2. (Միջին) - Արագագործության համեմատություն

Համեմատել շղթայական հեշավորման և գծային հետազոտման մեթոդների արագագործությունը մեծ չափի տվյալների դեպքում: Գեներացնել մեծ քանակի պատահական տվյալներ (օրինակ՝ 10^6 տարր), չափել ներդրման և առկայության ստուգման գործողությունների կատարման ժամանակը, ինչպես նաև փորձարկել տարբեր բեռնվածության գործակցի արժեքների դեպքում: Վերլուծել ստացված արդյունքները և գնահատել մեթոդների արդյունավետությունը տարբեր պայմաններում:

4. Տնային հանձնարարություն

Խնդիր 3. (Միջին) – Քառակուսային հետազոտման իրականացում

Նախորդ խնդրում իրականացված հեշ-աղյուսակը ձևափոխել այնպես, որ կոնֆլիկտները լուծվեն

քառակուսային հետազոտմամբ (quadratic probing)՝ $h(k)+i^2$ բանաձևով:

Համեմատել արդյունքները գծային հետազոտման հետ՝ հատկապես բեռնվածության մեծ արժեքների դեպքում:

Խնդիր 4. (Միջին) – Կրկնակի հեշավորում

Իրականացնել հեշ-աղյուսակ, որտեղ կոնֆլիկտները լուծվում են կրկնակի հեշավորմամբ՝ $h_1(k)+i*h_2(k)$:

Նշում՝ ընտրել այնպիսի $h_2(k)$, որը երբեք 0 չի դառնում և ապահովում է ամբողջ աղյուսակի անցումը:

Խնդիր 5. (Բարդ) – Կլաստերիզացիայի ուսումնասիրություն

Իրականացնել փորձարկում, որտեղ նույն տվյալների վրա կիրառվում են՝

- գծային հետազոտում,
- քառակուսային հետազոտում,
- կրկնակի հեշավորում:

Հաշվել և համեմատել միջին որոնման քայլերի քանակը և ամենաերկար հաջորդական զբաղված հատվածի (cluster) երկարությունը: Վերլուծել, թե որ մեթոդն է ավելի լավ աշխատում տարբեր բեռնվածության գործակցի դեպքում:

Խնդիր 6. (Բարդ) – Վերահեշավորում (Rehashing)

Լրացնել հեշ-աղյուսակի իրականացումը այնպես, որ երբ բեռնվածության գործակիցը գերազանցում է տրված շեմը (օրինակ՝ 0.7), ստեղծվի նոր, ավելի մեծ

աղյուսակ և բոլոր տարրերը տեղափոխվեն նոր աղյուսակ՝ վերահեշավորման միջոցով։ Վերլուծել, թե ինչպես է այս գործընթացը ազդում insert և contains գործողությունների արդյունավետության վրա, մասնավորապես՝ բացատրել, թե ինչու է վերահեշավորումը թանկ գործողություն, սակայն ապահովում է գործողությունների ամորտիզացված O(1) բարդություն։

ԳԼՈՒԽ 4.
ՀԻՇՈՂՈՒԹՅԱՆ ԿԱՌԱՎԱՐՈՒՄ ԵՎ ԽԵԼԱՑԻ ՑՈՒՑԻՉՆԵՐ

Այս գլխում ներկայացվում են ժամանակակից ծրագրավորման լեզուներում ռեսուրսների կառավարման սկզբունքները՝ հիմնված RAII (Resource Acquisition Is Initialization) գաղափարի վրա։ Ուսումնասիրվում են std::unique_ptr, std::shared_ptr և std::weak_ptr խելացի ցուցիչները C++ ստանդարտ գրադարանում, դրանց կիրառությունը և առավելությունները դասական դինամիկ հիշողության կառավարման նկատմամբ։

Արդյունքում ուսանողը կարողանում է անվտանգ և արդյունավետ կառավարել ռեսուրսները՝ խուսափելով հիշողության արտահոսքերից և սխալներից։

ԴԱՍ 31. STD::UNIQUE_PTR, ԿԻՐԱՌՈՒԹՅՈՒՆՆԵՐ

Տևողություն՝ 90 րոպե

Թեմա՝ RAII սկզբունք, բացառիկ տնօրինում, std::move և զրոյական ծախսերի աբստրակցիա։

1. Դասի նպատակը

Ձևավորել պատկերացում, որ std::unique_ptr-ն ապահովում է ռեսուրսների անվտանգ կառավարում՝ առանց լրացուցիչ հաշվարկային ծախսերի, և հանդիսանում է նախընտրելի ընտրություն բացառիկ սեփականատիրության դեպքերում:

2. Մեթոդական ցուցումներ

- ✓ **Տնօրինում:** unique_ptr-ը չի կարող պատճենվել, այն կարող է միայն տեղափոխվել (std::move): Յուրաքանչյուր ռեսուրս ունի միայն մեկ սեփականատեր:
- ✓ **Կյանքի տևողություն:** Ցույց տալ, որ երբ std::unique_ptr-ը դուրս է գալիս իր տեսանելիության սահմաններից ({ } բլոկից) և չի տեղափոխվում, համապատասխան օբյեկտը ավտոմատ ջնջվում է:
- ✓ **Հատուկ ջնջիչներ:** Նշել, որ std::unique_ptr-ն կարող է կիրառվել ոչ միայն հիշողության, այլ նաև այլ ռեսուրսների (օրինակ՝ ֆայլեր, սոքետներ) կառավարման համար:

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Հեշտ) - Գործարանի ձևանմուշ

Իրականացնել std::unique_ptr<Shape> createShape (const std::string& type) ֆունկցիա: Ցույց տալ, թե ինչպես է տնօրինման իրավունքը փոխանցվում կանչող ֆունկցիային:

Խնդիր 2. (Միջին) - Միակապ ցուցակ՝ unique_ptr-ով

Վերափոխել նախկինում իրականացված միակապ ցուցակը այնպես, որ next ցուցիչը լինի std::unique_ptr:

Հարց՝ ի՞նչ խնդիր կարող է առաջանալ շատ երկար ցուցակի դեպքում:

Խնդիր 3. (Բարդ) - Զանգվածի կառավարում

Ստեղծել դինամիկ ամբողջ թվերի զանգված երկու տարբեր եղանակներով՝ new int[n] և std::unique_ptr<int[]> օգտագործմամբ: Երկու դեպքում էլ լրացնել զանգվածը արժեքներով և օգտագործել այն ծրագրում: Համեմատել այս երկու մոտեցումները՝ հիշողության կառավարման, անվտանգ օգտագործման և հնարավոր սխալների (օրինակ՝ հիշողության արտահոսք) տեսանկյունից:

Նշում՝ ուշադրություն դարձնել, թե ինչպես է std::unique_ptr-ը ավտոմատ ազատում հիշողությունը տեսանելիության տիրույթից դուրս գալու դեպքում:

4. Տնային հանձնարարություն

Խնդիր 4. (Միջին) - unique_ptr-ի փոխանցում ֆունկցիային

Իրականացնել ֆունկցիա, որը որպես արգումենտ ստանում է std::unique_ptr<T> և օգտագործում է այն: Փորձարկել երկու տարբերակ՝

✓ փոխանցում std::move-ով,

✓ փոխանցում հղումով (const std::unique_ptr<T>&):

Վերլուծել՝ ինչ տարբերություն կա սեփականատիրության տեսանկյունից:

Խնդիր 5. (Միջին) - Ռեսուրսի կառավարում

Իրականացնել դաս, որը բացում է ֆայլ (օրինակ՝ std::fstream) և օգտագործում է std::unique_ptr՝ ֆայլի ավտոմատ փակումը ապահովելու համար: Ցույց տալ, որ ֆայլը փակվում է նույնիսկ բացառության (exception) դեպքում:

Խնդիր 6. (Բարդ) - Երկկապ ցուցակ

Փորձել իրականացնել երկկապ ցուցակ, որտեղ՝

- next ցուցիչը կլինի std::unique_ptr,
- prev ցուցիչը՝ սովորական:

Բացատրել՝ ինչու հնարավոր չէ երկու կողմն էլ պահել std::unique_ptr-ով:

ԴԱՍ 32. STD::SHARED_PTR, STD::WEAK_PTR, ԿԻՐԱՌՈՒԹՅՈՒՆՆԵՐ

Տևողություն՝ 90 րոպե

Թեմա՝ std::shared_ptr, std::weak_ptr, հղումների քանակ և շրջանաձև կախվածությունների վերացում:

1. Դասի նպատակը

Ձևավորել պատկերացում std::shared_ptr-ի աշխատանքի մեխանիզմի (հղումների հաշվարկ, կառա-

վարման բլոկ) վերաբերյալ և հասկացնել շրջանաձև կախվածությունների առաջացման պատճառներն ու դրանց կանխման եղանակները։

2. Մեթոդական ցուցումներ

- ✓ **Հղումների հաշվարկ**: Օբյեկտը ջնջվում է միայն այն դեպքում, երբ վերջին shared_ptr-ը դադարում է հղվել դրան։
- ✓ **Շրջանաձև կախվածություն:** Ներկայացնել իրավիճակ, որտեղ երկու օբյեկտ հղվում են միմյանց shared_ptr-ով, ինչի արդյունքում նրանց հղումների հաշվարկը երբեք չի դառնում 0, և օբյեկտները չեն ջնջվում։
- ✓ **Թույլ հղում:** Նշել, որ std::weak_ptr-ը չի մասնակցում հղումների հաշվարկին և օգտագործվում է որպես «դիտորդ»։ Այն թույլ է տալիս ստուգել օբյեկտի գոյությունը (lock() մեթոդով)՝ առանց դրա կյանքի տևողությունը երկարացնելու։

3. Առաջադրանքներ լսարանային աշխատանքի համար

Խնդիր 1. (Միջին) - Շրջանաձև կախվածության մոդելավորում

Ստեղծել Student և University դասեր, որտեղ ուսանողը պահում է shared_ptr<University>, իսկ համալսարանը՝ shared_ptr<Student>: Ցույց տալ, որ փլուզիչները չեն կանչվում։ Լուծել խնդիրը՝ օգտագործելով weak_ptr։

Խնդիր 2. (Միջին) - Քեշի պարզ իրականացում

Իրականացնել պարզ քեշ համակարգ, որը պահում է օբյեկտները weak_ptr-ների միջոցով: Եթե օբյեկտը արդեն ջնջված է հիմնական ծրագրում, քեշը պետք է դա հայտնաբերի՝ օգտագործելով lock() մեթոդը:

Խնդիր 3. (Բարդ) - Դիտորդ նախագծման ձևանմուշ (Observer Pattern)

Իրականացնել իրադարձությունների բաշխման համակարգ, որտեղ սուբյեկտը պահում է իր դիտորդների ցուցակը std::weak_ptr-ներով՝ խուսափելու համար շրջանաձև կախվածություններից:

4. Տնային հանձնարարություն

Խնդիր 4. (Հեշտ) - shared_ptr-ի պատճենում և հաշվարկ

Գրել ծրագիր, որտեղ ստեղծվում է std::shared_ptr և մի քանի անգամ պատճենվում է: Արտածել use_count() արժեքը տարբեր կետերում և բացատրել փոփոխությունները:

Խնդիր 5. (Հեշտ) - weak_ptr-ի վավերության ստուգում

Ստեղծել std::shared_ptr և համապատասխան std::weak_ptr: Ցույց տալ, թե ինչպես է expired() և lock() մեթոդների վարքը փոխվում shared_ptr-ի հեռացումից հետո:

Խնդիր 6. (Բարդ) - Երկկողմանի կապ ունեցող օբյեկտներ

Իրականացնել երկու դաս, որոնք հղվում են մի-

մյանց: Սկզբում օգտագործել միայն std::shared_ptr և ցույց տալ խնդիրը, ապա վերափոխել լուծումը՝ կիրառելով std::weak_ptr:

Տեսակ	Տնօրինում	Ծախսեր	Հիմնական կիրառություն
std::unique_ptr	Բացառիկ	O(1)	Ռեսուրսների հիմնական կառավարում
std::shared_ptr	Կիսված	O(հղումների հաշվարկ + կառավարման բլոկ)	Մի քանի սեփականատեր
std::weak_ptr	Դիտորդ	O(1)	Ցիկլերի կանխում, քեշ համակարգեր

Մարիամ Մերոբի Հարությունյան
Հայկ Կարենի Ասլանյան

ՏՎՅԱԼՆԵՐԻ ԿԱՌՈՒՑՎԱԾՔՆԵՐ և ՕԲՅԵԿՏԱ-ԿՈՂՄՆՈՐՈՇՎԱԾ ԾՐԱԳՐԱՎՈՐՈՒՄ

Գործնական պարապմունքների ձեռնարկ

Գ.Հ.Խ գլխավոր խմբագիր – Մ.Է. Ավագյան
Խմբագիր – Ա.Ս.Եսայան
Համակարգչային էջադրում – Ա.Գ.Անտոնյան

Ռուս-հայկական համալսարանի
Գիտական հրապարակումների խմբագրության
հասցե՝ 0051, ք. Երևան, փող. Հովսեփ Էմին, 123
հեռ./ֆաքս՝ (+374 12) 77-57-75 (ներքին՝ 392)
էլ.փոստ՝ maria.avakian@rau.am

Պատվեր № 10
Ստորագրված է հրապարակման՝ 23.06.2026թ.
Չևաչափ՝ 60x70$^{1}/_{16}$
Ծավալ՝ 10 պայմանական տ.թ.
Տպաքանակը՝ 250 օրինակ։

ISBN 978-9939-67-418-6