Глава 4

Структуре података

Ефикасна организација података у меморији, тако да се подаци могу лако претраживати и мењати представља важан предуслов писања ефикасних програма. Поред примитивних типова података (попут, на пример, статички алоцираних низова или кориснички дефинисаних структурних типова), савремени програмски језици нуде велики број **библиотечких структура података** које олакшавају процес програмирања. У наставку ћемо приказати структуре података, које се толико често употребљавају да су директно подржане у библиотекама већине програмских језика.

Структуре података можемо посматрати преко њихових операција, тј. преко функционалности које пружају својим корисницима. У том светлу за коришћење неке библиотечке структуре података потребно је знати њен апстрактни интерфејс (функције тј. методе које се позивају да би се извршиле операције над подацима складиштеним у склопу структуре), док њена имплементација може остати у потпуности сакривена. При том, веома је важно имати осећај и о сложености сваке операције.

Језик C++ пружа подршку за велики број структура података, било примитивних, подржаних кроз сам језик, било подржаних кроз стандардну библиотеку (STL).

По начину како су имплементиране, библиотечке структуре података (каже се и **контејнери**, енгл. containers) се могу груписати на следећи начин.

- У групу секвенцијалних контејнера спадају `array`, `string`, `vector`, `list`, `forward_list` и `deque`.
- У групу адаптора контејнера спадају `stack`, `queue` и `priority_queue`.
- У групу асоцијативних контејнера спадају `set`, `multiset`, `map` и `multimap`.
- У групу неуређених асоцијативних контејнера спадају `unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap`.

Тесно повезани са контејнерима су **итератори** који су уопштење показивача и представљају објекте који показују на одређено место (на одређен елемент) унутар контејнера или евентуално на део меморије непосредно иза елемената контејнера (такав је специјални итератор који се добија методом `end`). Итератори се могу *дереференцирати* применом оператора `*`, што значи да на основу итератора можемо очитати или изменити елемент контејнера на који тај итератор показује. Уз то, итератори се могу померати (у зависности од контејнера, само у једном или у оба смера) и поредити (поређење једнакости и различитости итератора операторима `==` и `!=` је увек подржано, а понекад је могуће упоредити да ли је неки оператор испред или другог, операторима `<`, `<=`, `>` и `>=`) . Многе библиотечке функције које се примењују над елементима контејнера захтевају пар итератора који ограничава део контејнера на који ће се функција примењивати (најчешће се наводи пар итератора добијених методама `begin` који `end` ограничавају целокупан контејнер тј. све његове елементе).

Секвенцијални контејнери се користе за складиштење серија елемената. Елементи се увек складиште секвенцијално, један иза другог, па сви секвенцијални контејнери представљају одређена уопштења примитивних, статички алоцираних низова. Карактерише их могућност обиласка свих елемената редом (понекад у оба правца), као и могућност индексног приступа елементу на основу његове позиције (додуше, та операција не мора увек да буде константне сложености). Са изузетком `array`, који представља само танак омотач око примитивног статичког низа и чија димензија мора бити позната приликом превођења програма, сви остали секвенци-

јални контејнери су динамички алоцирани и допуштено је и уметање и брисање елемената (и на почетак и на средину и на крај), при чему сложеност тих операција зависи од одабира секвенцијалног контејнера (стога их треба користити веома обазриво). Иако имају доста сличан интерфејс, секвенцијални контејнери су имплементирани на различите начине, па им се разликује сложеност извођења одређених операција. Сваки од њих има одређене предности и мане и сценарије у којима је пожељно и сценарије у којима га није пожељно користити.

Адаптори контејнера само представљају слој изнад неког од постојећих секвенцијалних контејнера и пружају апстрактни интерфејс изнад имплементације секвенцијалног контејнера, имплементирајући функције тог интерфејса коришћењем секвенцијалног контејнера за складиштење података. Адаптори имају свој подразумевани секвенцијални контејнер, који се може променити приликом декларисања променљивих.

- `stack` имплементира функције стека (LIFO) где се елементи могу додавати и уклањати са једног краја.
- `queue` имплементира функције реда (FIFO) где се елементи додају на један, а узимају са другог краја.
- `priority_queue` имплементира функције реда са приоритетом где се елементи додају у произвољном редоследу, а ваде у нерастућем редоследу вредности (из реда се увек може прочитати и извадити само највећи елемент реда).

Асоцијативни контејнери подразумевају да се уметање, претрага и брисање елемената врши на основу њихове вредности, а не на основу позиције на којој се налазе — приступ елементима је на основу кључа, а не на основу индекса тј. позиције. Код секвенцијалних контејнера однос елемената се природно успоставља на основу њиховог положаја унутар контејнера (без обзира на вредности). На пример, у низу је могуће и да се неки мањи елемент налази испред већег и да се неки већи елемент налази испред мањег. Са друге стране положај елемената у асоцијативним контејнерима се успоставља искључиво на основу њихове вредности (на пример, у уређеном скупу мањи елементи се увек налазе испред већих).

Основни асоцијативни контејнери су *скупови* и *мапе* тј. *речници*. Скупови одговарају скуповима у математици и пружају ефикасно додавање и избацивање елемената, као и проверу да неки елемент припада скупу. Мапе тј. речници чувају коначна пресликавања у којима се неким кључевима (елементима неког домена) придружују неке вредности (елементи неког кодомена).

На основу начина како су имплементирани, асоцијативни контејнери се деле на *уређене* и *неуређене*. Уређени асоцијативни контејнери корисницима нуде одређене додатне функционалности (на пример, испис свих елемената у растућем редоследу вредности или проналажење најмањег елемената који је већи од дате вредности), али по цену да су понекад мало спорији од неуређених.

Уређени асоцијативни контејнери су следећи:

- `set<T>` – скуп елемената типа `T`
- `multiset<T>` – мултискуп елемената типа `T` (допуштена су вишеструка појављивања елемената)
- `map<K, V>` — пресликавање кључева типа `K` у вредности типа `V`
- `multimap<K, V>` — пресликавање кључева типа `K` у вредности типа `V` при чему се сваки оригинал може сликати у више различити слика.

Сви они претпостављају да се њихови елементи (тј. кључеви у случају мапа и мултимапа) могу поредити (релацијским оператором `<`). Обично су имплементирани помоћу самобалансирајућих уређених бинарних дрвета и карактерише их логаритамска сложеност основних операција. Укратко, складиштње елемената у дрвету подразумева да су елементи складиштени у чворовима који осим релевантних вредности чувају и показиваче на лева и десна поддрвета. Уређена бинарна дрвета подразумевају да се је вредност у сваком чвору већа или једнака од вредности у свим чворовима левог и строго мања од вредности од свих чворова у десном поддрвету, што значи да се се уметање и претрага остварују веома слично поступку бинарне претраге (поређењем са вредношћу у корену и затим преласком на лево или десно поддрво). Стога сложеност зависи од висине бинарног дрвета, која у случају да је дрво балансирано (да се висина левог и десног поддрвета не разликују превише), логаритамски зависи од укупног броја чворова у дрвету.

Неуређени асоцијативни контејнери су следећи:

- `unordered_set<T>`
- `unordered_multiset<T>`
- `unordered_map<K, V>`
- `unordered_multimap<K, V>`

Ообично су имплементирани помоћу *хеш-табела* и карактерише их амортизована константна сложеност основних операција. Неуређени контејнери претпостављају да постоји дефинисана функција *хеш-функција* `hash` која елементе скупа тј. кључеве мапе слика у целобројне вредности (тзв. хеш кодове) које одређују позицију унутар низа вредности на којој елемент треба да се нађе, ако је елемент скупа тј. кључ у мапи. Могуће је и да наступе *колизије* тј. да више елемената има исте хеш-кодове тј. да се хеш-функцијом сликају на исту позицију. Постоји неколико начина разрешавања колизија (најједноставнија подразумева да се на тој позицији чува листа свих елемената који су хеш-функцијом придружени тој позицији).

4.1 Скупови и мапе (речници)

4.1.1 Скупови

У програмима често имамо потребе да одржавамо скуп елемената (без дупликата), у који ефикасно можемо да додајемо елементе, из кога ефикасно можемо да избацујемо елементе и за који ефикасно можемо да проверавамо да ли је нека задата вредност елемент скупа. Савремени програмски језици у својим библиотекама пружају структуре података које нуде баш ове операције.

У језику C++ скуп је подржан кроз две класе: `set<T>` и `unordered_set<T>`, где је `T` тип елемената скупа (за њихово коришћење је потребно укључити заглавље `<set>` тј. `<unordered_set>`). Имплементација је различита (прва је заснована на балансираним бинарним дрветима, а друга на хеш таблицама), па су им временске и просторне карактеристике донекле различите.

Скупови подржавају следеће основне операције (за преглед свих операција упућујемо читаоца на документацију):

- `insert` - умеће нови елемент у скуп (ако је елемент већ у скупу, операција нема ефекта). Када се користи `set`, сложеност уметања је $O(\log k)$, где је k број елемената у скупу, а када се користи `unordered_set`, сложеност најгорег случаја је $O(k)$, док је просечна сложеност $O(1)$, при чему је амортизована цена сваког додавања у склопу већег броја узастопних додавања такође $O(1)$. Нагласимо и да константе код сложености O(1) могу бити релативно велике и да уметање не можемо сматрати јако брзом операцијом. Честа операција је додавање n елемената у скуп. Најгора сложеност је ако су сви елементи различити и износи приближно $\log 1 + \log 2 + \ldots + \log n$, за шта се може показати да је $O(n \log n)$.
- `erase` - уклања дати елемент из скупа (ако елемент не постоји у скупу, скуп се не мења). Сложеност је иста као у случају уметања.
- `find` - проверава да ли скуп садржи дати елемент и враћа итератор на њега или `end` ако је одговор негативан. Тако се провера припадности елемента `e` скупу `s` може извршити са `if (s.find(e) != s.end()) ...` Сложеност најгорег случаја ове операције ако се користи `set` је $O(\log k)$, а ако се користи `unordered_set` сложеност најгорег случаја је $O(k)$, али је просечна сложеност $O(1)$.
- `size` - враћа број елемената скупа

Могућа је и итерација кроз елементе скупа коришћењем петље облика `for (T element : skup)`, при чему се елементи колекције `set` набрајају у сортираном, а `unordered_set` у прилично насумичном редоследу (редослед је одређен хеш-функцијом која се користи и на њега се, као што само име `unordered` говори, не треба ослањати).

Уређени скупови (колекција `set`) подржавају и методе

- `lower_bound(x)` - проналази најмањи елемент скупа који је већи или једнак од дате вредности `x` и враћа итератор који указује на њега (или `end`, ако такав елемент не постоји),
- `upper_bound(x)` - проналази најмањи елемент скупа који је строго већи од дате вредности `x` и враћа итератор који указује на њега (или `end`, ако такав елемент не постоји).

Ако се у скуп стављају елементарни типови података (бројеви, ниске, ...), тада се користе подразумевани поредак тј. хеш-функција. Ово је могуће променити, али излази ван домена овог материјала. Да би се формирао скуп елемената неког типа над којим није дефинисан подразумевани поредак нити хеш-функција (најчешће скуп структура или објеката неке класе), потребно је посебно дефинисати их. Издвојићемо само случај када желимо да направимо скуп у ком су елементи уређени нерастуће (при чему на типу података постоји подразумевани неопадајући поредак). Такав скуп је најједноставније дефинисати коришћењем `set<T, greater<T>>`, где је за употребу `greater` потребно укључити заглавље `<functional>`.

4.1.1.1 Мултискупови

Скупови, као и у математици, не могу да садрже дупликате. Када се елемент који већ постоји у скупу убацује у скуп методом `insert`, скуп се не мења. Једно уопштење скупова дају *мултискупови* у којима је допуштено понављање елемената. Мултискупови су подржани библиотечком структуром `multiset<T>`, која се користи на потпуно исти начин као и `set<T>` (за њено коришћење је такође довољно укључити заглавље `<set>`).

4.1.2 Мапе (речници)

Програмски језик C++ пружа подршку за креирање мапа (речника, асоцијативних низова) који представљају колекције података у којима се кључевима неког типа придружују вредности неког типа (не обавезно истог). На пример, именима месеци (подацима типа `string`) можемо доделити број дана (податке типа `int`). Речници се представљају објектима типа `map<TipKljuca, TipVrednosti>`, дефинисаном у заглављу `<map>`. На пример,

```
map<string, int> brojDana =
{
    {”januar”, 31},
    {”februar”, 28},
    {”mart”, 31},
    ...
};
```

Приметимо да смо иницијализацију мапе извршили тако што смо навели листу парова облика `{kljuc, vrednost}`. Иницијализацију није неопходно извршити одмах током креирања, већ је вредности могуће додавати (а и читати) коришћењем индексног приступа (помоћу заграда `[]`).

```
map<string, int> brojDana;
brojDana[”januar”] = 31;
brojDana[”februar”] = 28;
brojDana[”mart”] = 31;
...
```

Мапу, дакле, можемо схватити и као низ тј. вектор у коме индекси нису обавезно из неког целобројног интервала облика $[0, n)$, већ могу бити произвољног типа.

Претрагу кључа можемо остварити методом `find` која враћа итератор на пронађени елемент тј. итератор иза краја мапе (који добијамо методом или функцијом `end`), ако елемент не постоји. На пример,

```
string mesec; cin >> mesec;
auto it = brojDana.find(mesec);
if (it != end(brojDana))
   cout << ”Broj dana: ” + *it << endl;
else
   cout << ”Mesec nije korektno unet” << endl;
```

Све елементе речника могуће је исписати коришћењем петље `for`. На пример,

```
for (auto& it : brojDana)
     cout << it.first << ”: ” << it.second << endl;
```

Алтернативно, можемо експлицитно користити итераторе

```
for (auto it = brojDana.begin(); it != brojDana.end(); it++)
     cout << it->first << ”: ” << it->second << endl;
```

Итерација се врши у сортираном редоследу кључева.

Постоји и облик неуређене мапе (`unordered_map` из истоименог заглавља), која може бити у неким ситуацијама мало бржа него уређена (сортирана) мапа, но то је обично занемариво. Кључеви сортиране мапе могу бити само они типови који се могу поредити релацијским операторима, док кључеви неуређене мапе могу бити само они типови који се могу лако претворити у број (тзв. хеш-вредност). Ниске, које ћемо најчешће користити као кључеве, задовољавају оба услова.

Задатак: Дупликати

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Скупови

Библиотеке савремених програмских језика обично нуде и колекције података за репрезентовање скупова. Једна могућност је да се користи колекција заснована на балансираном бинарном дрвету. У језику C++ таква је колекција `set`. Додавање елемента у скуп се врши методом `insert` (при том се аутоматски води рачуна да се скуп не мења додавањем елемента који већ постоји), док се број елемената одређује методом `size()`.

Анализа сложености. Са имплементацијом скупа базираном на балансираном бинарном дрвету, убацивање у скуп је обично сложености $O(\log k)$, где је k број елемената у скупу, па је укупна сложеност овог приступа највише $O(n \log n)$.

```
// ucitavamo elemente u skup
set<unsigned> a;
int n;
cin >> n;
for (int i = 0; i < n; i++) {
  unsigned x;
  cin >> x;
  a.insert(x);
}
// ispisujemo broj elemenata skupa
cout << a.size() << endl;
```

Једна могућност је да се користи колекција заснована на хеширању. У језику C++ таква је колекција `unordered_set` и она се користи на исти начин као и `set` (при чему тип елемената мора да буде такав да је за њега расположива хеш-функција).

Анализа сложености. Сложеност најгорег случаја додавања у скуп заснован на хеширању је $O(k)$, где је k број елемената у скупу, али је амортизована цена једног додавања у склопу већег броја додавања једнака $O(1)$. Зато је укупна сложеност алгоритма једнака $O(n)$ (уз релативно велики константни фактор који уметање у овакав скуп носи).

```
// ucitavamo elemente u skup
unordered_set<unsigned> a;
int n;
cin >> n;
for (int i = 0; i < n; i++) {
  unsigned x;
  cin >> x;
  a.insert(x);
}
// ispisujemo broj elemenata skupa
cout << a.size() << endl;
```

Задатак: Сортирање бројева

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Сортирање уз помоћ мултискупа

Сортирање се може извршити коришћењем мултискупа. Користи се алгоритам *сортирања уметањем* (енгл. *insertion sort*), једино што уместо уметања елемената у низ врши уметање елемената у мултискуп. На крају

се испишу сви елементи мултискупа редом (они се чувају у сортираном редоследу). Пошто се мултискуп имплементира коришћењем сортираног бинарног дрвета, овај се алгоритам назива и *сортирањем коришћењем дрвета* (енгл. *tree sort*).

У језику C++ можемо користити библиотечку колекцију `multiset`.

Анализа сложености. Пошто се уметање у мултиску извршава у сложености $O(\log k)$, где је k број елемената у мултискупу, а испис у времену $O(k)$, сложеност овог поступка сортирања је $O(n \log n)$. Заузеће меморије је $O(n)$, уз, додуше, мало већи константни фактор него када се елементи користе у низ. Додатно, елементи нису поређани један уз други у меморији, што мало успорава приступ.

```
int n;
cin >> n;
multiset<int> a;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  a.insert(x);
}
for (int x : a)
  cout << x << endl;
```

Види другачија решења овог задатка.

Задатак: Број различитих дужина дужи

Дато је N парова тачака које представљају крајеве дужи у простору. Исписати колико различитих дужина дужи се појављује у задатом скупу дужи.

Улаз: У првом реду улаза налази се природан број N ($N \leq 50000$) који представља број дужи. У следећих N редова следи опис тих N дужи са 6 целих бројева ($-10^9 \leq X_1, Y_1, Z_1, X_2, Y_2, Z_2 \leq 10^9$) одвојених празним местима, који редом представљају крајеве сваке дужи.

Излаз: У једини ред излаза потребно је исписати колико различитих дужина се појављује у задатом скупу дужи.

Пример 1

Улаз	*Излаз*
7 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 1 1 0 0 0 0 0 1 1 0 0 0 1 1 1	3

Решење

Скуп дужина дужи

Задатак можемо решити и помоћу библиотечких структура података. У језику C++ можемо употребити једну од две имплементације скупа (`set` или `unordered_set`). Квадрате дужина дужи убацујемо у скуп и на крају очитавамо број елемената тог скупа. Јако је важно нагласити да проналажење самих дужина коренивањем, у облику реалних бројева, не би дало добро решење. Наиме, услед грешака у запису тј. заокруживању реалних бројева, дешава се да се једнаке вредности могу протумачити као различите, а различите вредности као једнаке. Стога реалне бројеве никада не би требало чувати као елементе скупа нити као кључеве у мапи.

Анализа сложености. Ако претпоставимо да је операција убацивања елемента у скуп који садржи k елемената сложености $O(k)$, укупна сложеност овог алгоритма је $O(n \log n)$.

```
// skup svih duzina duzi
unordered_set<long long> duzine;
```

```
// ucitavamo koordinate temena duzi i ubacujemo njihove duzine u skup
int n;
cin >> n;
for (int i = 0; i < n; i++) {
  int x1, y1, z1, x2, y2, z2;
  cin >> x1 >> y1 >> z1 >> x2 >> y2 >> z2;
  duzine.insert((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) + (z1-z2)*(z1-z2));
}

// ispisujemo broj elemenata skupa
cout << duzine.size() << endl;
```

Задатак: Својство 132

Низ $a_0, a_1, \ldots, a_{n-1}$ задовољава 132-својство ако постоји тројка индекса $0 \leq i < j < k < n$, тако да је $a_i < a_k < a_j$. Напиши програм који испитује да ли низ задовољава 132-својство.

Улаз: Са стандардног улаза се учитава број елемената низа n ($3 \leq n \leq 10^5$), а затим и n елемената низа (раздвојених размаком).

Излаз: На стандардни излаз исписати `da` или `ne` у зависности од тога да ли низ задовољава 132-својство или не.

Пример 1

Улаз
```
4
1 2 3 4
```
Излаз
```
ne
```

Пример 2

Улаз
```
4
3 1 4 2
```
Излаз
```
da
```

Објашњење

На пример, елементи $1, 4, 2$ задовољавају 132-својство.

Пример 3

Улаз

```
7
9 11 8 9 10 7 9
```

Излаз

```
da
```

Објашњење

На пример, елементи $9, 11, 10$ задовољавају 132-својство.

Решење

Највећи елемент десно од a_j строго мањи од a_j – скуп

За сваки интервал (a_i, a_j), где је a_i минимум префикса $a_0, \ldots, a_j$ треба проверити да ли у десном делу низа, иза позиције j, постоји елемент a_k који припада том интервалу. Постоје два приступа да се то ефикасно урадимо:

- Потребно и довољно да десно од a_j постоји елемент који је строго мањи од a_j и да за највећи елемент десно од a_j који је строго мањи од a_j важи да је строго већи од a_i.
- Потребно је и довољно да десно од a_j постоји елемент који је строго већи од a_i и да за најмањи елемент десно од a_j који је строго већи од a_i важи да је строго мањи од a_j.

Илуструјмо решење засновано на првом приступу. Тврђење важи, јер ако је највећи елемент десно од a_j који је строго мањи од a_j мањи или једнак од a_i, онда су сви елементи десно од a_j строго мањи од a_j мањи или једнаки a_i и ниједан не припада интервалу (a_i, a_j).

Једно ефикасно решење можемо добити коришћењем структуре података у којој ћемо чувати све елементе десно од текућег елемента a_j, а која нам омогућава да међу њима ефикасно пронађемо највећи елемент који је строго мањи од дате вредности a_j. У језику C++ можемо користити библиотечку колекцију скуп (`set`) уређен опадајући и њену методу `upper_bound` (ефикасна метода `upper_bound`, која ради у логаритамској сложености, постоји код уређених скупова `set`, али не и код неуређених скупова `unordered_set`).

Елементе a_j можемо обилазити здесна налево, за сваког проверавати да ли је средишњи елемент неке 132-тројке и ако није, додавати га у скуп (јер ће он бити десно од свих елемената које ћемо у наредним итерацијама обрађивати). Обиласком слева надесно требало би елементе избацивати из скупа, што је мало неелегантније (мада је временска сложеност иста).

Обилазак низа здесна налево мало компликује проналажење минимума префикса (који се израчунавају инкрементално, слева надесно, међутим, њих можемо одредити у посебном пролазу на самом почетку и сместити у помоћни низ.

Анализа сложености. Додавање елемената у скуп који садржи m елемената, као и одређивање највећег елемента већег од дате вредности су сложености $O(\log m)$. Пошто се и додавање и претрага врше n пута, сложеност овог приступа је $O(n \log n)$ (израчунавање максимума префикса које се врши у фази претпроцесирања је сложености $O(n)$).

```
bool svojstvo132(const vector<int>& a) {
  int n = a.size();
  // niz minimuma prefiksa - na poziciji i nalazi se minimum prefiksa
  // niza na pozicijama [0, i]
  vector<int> minP(n);
  minP[0] = a[0];
  for (int i = 1; i < n; i++)
    minP[i] = min(minP[i-1], a[i]);

  // elementi desno od aj
  set<int, greater<int>> elementi_desno;
  elementi_desno.insert(a[n-1]);

  // za svaki element aj proveravamo da li moze biti sredisnji u nekoj
  // 132-trojci
  for (int j = n-2; j > 0; j--) {
    // analiziramo interval (ai, aj) i proveravamo da li postoji ak
    // desno od j koji mu pripada
    int ai = minP[j], aj = a[j];
    // interval je prazan
    if (ai == aj)
      continue;
    // trazimo najveci element desno od aj koji je strogo manji od aj
    auto najveci_desno_manji_od_aj = elementi_desno.upper_bound(aj);
    // ako on postoji i strogo je veci od ai, pronadjena je 132-trojka
    if (najveci_desno_manji_od_aj != elementi_desno.end() &&
        *najveci_desno_manji_od_aj > ai)
      return true;

    // najveci element desno od aj koji je manji od aj je manji ili
    // jednak ai, pa aj ne moze biti sredisnji element trojke

    // aj ce biti desno od elemenata u narednim iteracijama
    elementi_desno.insert(aj);
  }
  // nije pronadjena 132-trojka
  return false;
}
```

Задатак: Фреквенција знака

Написати програм који чита једну реч текста коју чине само велика слова енглесне абецеде и исписује слово који се најчешће појављује и колико пута се појављује. Ако се више слова најчешће појављује, исписује се слово које се пре појавило у речи.

Улаз: У једној линији стандардног улаза налази се једна реч текста са не више од 20 слова.

Излаз: У првој линији стандарног излаза приказати слово које се најчешће појављује, а у другој линији стандардног улаза исписати и колико пута се појављује.

Пример 1		**Пример 2**	
Улаз	*Излаз*	*Улаз*	*Излаз*
`POPOKATEPETL`	`P` `3`	`BACACB`	`B` `2`

Решење

Кључни део задатака је да се за свако велико слово енглеског алфабета изброји колико се пута појављује у датој речи. Потребно је, дакле, увести 26 различитих бројача - по један за свако велико слово енглеског алфабета. Јасно је да увођење 26 различитих променљивих не долази у обзир. Потребна је структура података која пресликава дати карактер у број његових појављивања.

Најбоља таква структура у овом задатку је низ бројача у којем се на позицији нула чува број појављивања слова `A`, на позицији један слова `B` итд., све до позиције 25 на којој се налази број појављивања слова `Z`. Централно питање је како на основу карактера одредити позицију његовог бројача у низу. За то се користи чињеница да су карактерима придружени нумерички кодови (ASCII у језику C++) и то на основу редоследа карактера у енглеском алфабету. Редни број карактера је зато могуће одредити одузимањем кода карактера `A` од кода тог карактера (на пример, кôд карактера `D` је 68, док је код карактера `A` 65, одузимањем се добија 3, што значи да се бројач карактера `D` налази на месту број 3 у низу).

Задатак можемо решавати тако што ћемо два пута проћи кроз низ унетих слова тј. реч. У првом пролазу бројач појављивања сваког великог слова на које наиђемо увећавамо за 1. Након тога (или паралелно са тим) одређујемо и максималан број појављивања неког слова. За то користимо уобичајене начине за одређивање максимума низа.

У другом пролазу кроз низ унетих слова тј. реч тражимо прво слово речи чији је број појава једнак већ нађеном највећем броју појава неког слова. Када га нађемо исписујемо га и прекидамо петљу (на пример, наредбом `break`). Приметимо да овде заправо вршимо једноставну линеарну претрагу.

```
string rec;
cin >> rec;

int brojPojavljivanja[26] = {0};
for (char c : rec)
  brojPojavljivanja[c - 'A']++;

int maxPojavljivanja = 0;
for (int i = 0; i < 26; i++)
  if (brojPojavljivanja[i] > maxPojavljivanja)
    maxPojavljivanja = brojPojavljivanja[i];

for (char c : rec)
  if (brojPojavljivanja[c - 'A'] == maxPojavljivanja) {
    cout << c << endl
         << maxPojavljivanja << endl;
    break;
  }
```

Пресликавање карактера у њихов број појављивања могуће је остварити и библиотечким структурама података које представљају тзв. асоцијативне низове (мапе, речнике). У језику C++ бисмо могли употребити `map<char, int>` или `unordered_map<char, int>`. Ипак, ове структуре података су примереније за ситуације у којима није тако једноставно на основу кључа добити њихову нумеричку вредност и када је скуп допустивих кључева шири.

```
// rec koja se analizira
string rec;
cin >> rec;

// broj pojavljivanja svakog slova od A do Z
map<char, int> brojPojavljivanja;
// uvecavamo broj pojavljivanja svakog slova iz reci
for (char c : rec)
  brojPojavljivanja[c]++;

// odredjujemo najveci broj pojavljivanja slova
int maxPojavljivanja = 0;
for (auto it = brojPojavljivanja.begin(); it != brojPojavljivanja.end(); it++)
  if (it->second > maxPojavljivanja)
    maxPojavljivanja = it->second;

// ispisujemo slovo koje se prvo pojavljuje u reci sa tim brojem
// pojavljivanja
for (char c : rec)
  if (brojPojavljivanja[c] == maxPojavljivanja) {
    cout << c << endl
         << maxPojavljivanja << endl;
    break;
  }
```

Задатак: Фреквенције речи

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Бројање речи

Централно место у задатку је избројати појављивања сваке речи која се јавила на улазу. Овај задатак је донекле сличан задатку Фреквенција знака, једино што се уместо појединачних карактера броје речи. За разлику од ситуације када смо на основу ASCII кода карактера могли одредити његову позицију и тако број појављивања сваког карактера чувати у низу бројача, овај пут то није једноставно и у ефикасном решењу је потребно користити напредније структуре података.

У језику C++ пресликавање речи у њен број појављивања можемо реализовати или типом `map<string, int>` (који је заснован на претраживачком стаблу) или `unordered_map<string, int>` (који је заснован на хеш-таблици).

Учитавамо реч по реч док не дођемо до краја улаза. За сваку реч увећавамо њен број појављивања у мапи тј. у речнику.

Изразом `brojPojavljivanja[rec]` се приступа броју појављивања речи. У језику C++ се изразом `brojPojavljivanja[rec]++` може увећати број појављивања било да реч постоји у мапи или не (ако не постоји, овим се број појављивања поставља на 1).

Једном када је познат број појављивања сваке речи, тада је могуће одредити тражену реч са најмањим бројем појављивања. Пошто се у случају више речи са истим бројем појављивања тражи она која је лексикографски најмања, користимо лексикографско поређење (хијерархијско поређење на основу више критеријума).

```
string s;
map<string, int> brojPojavljivanja;
while (cin >> s)
  brojPojavljivanja[s]++;

int max = 0; string maxRec;
for (auto it = brojPojavljivanja.begin();
```

```
     it != brojPojavljivanja.end(); it++)
  if (it->second > max ||
      (it->second == max && it->first < maxRec)) {
    maxRec = it->first;
    max = it->second;
  }

cout << maxRec << " " << max << endl;
```

Задатак: Сегмент датог збира у низу целих бројева

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Ефикасна претрага збирова префикса

Изражавање збира сегмента у облику разлике збира два префикса нам даје могућност да стигнемо до ефикаснијег решења. Та техника је објашњена, на пример, у задатку Највећи збир префикса. Проблем можемо формулисати и овако. За сваки збир b_{j+1} префикса $[0, j+1)$ потребно је да пронађемо да ли постоји збир b_i префикса $[0, i)$ за неко $i < j$ таква да је $b_{j+1} - b_i = z$, где је z тражени збир, тј. да се провери да ли се међу збировима претходних префикса налази вредност $b_i = b_{j+1} - z$. Ако се та претрага врши линеарно, долазимо до имплементације веома сличне претходној, која испитује сваки пар елемената $i < j$. Пошто међу елементима низа може бити и негативних, збирови префикса нису сортирани и не можемо применити ни бинарну претрагу. Остаје нам, међутим, могућност да у некој структури података која омогућава ефикасно претраживање чувамо све збирове префикса за индексе $i < j$. Ако алгоритам организујемо тако да j увећавамо од 0 до $n-1$, тада се на крају сваког корака у ту структуру може додати и збир текућег сегмента (b_{j+1}). Структура треба да реализује претрагу по кључу, тако да је најбоље употребити асоцијативни низ (мапу тј. речник). У језику C++ то може бити `map` или `unordered_map`.

Пошто се у задатку тражи само одређивање броја сегмената са датим збиром, кључеви могу бити збирови префикса, а вредност придружена сваком кључу може бити број префикса са тим збиром. Да се тражила само провера да ли постоји сегмент са датим збиром, могли смо уместо асоцијативног низа (мапе, речника) чувати само скуп раније виђених вредности збирова префикса, а да су се експлицитно тражили сви префикси, онда бисмо сваки кључ пресликавали у низ вредности i таквих да је b_i једнако том кључу.

Анализа сложености. Ако рачунамо да ће претрага бити реализована у $O(\log n)$ (што је најчешће случај ако се користе структуре података засноване на бинарним стаблима), тада ће укупна сложеност ове имплементације бити $O(n \log n)$. Напоменимо да смо добитак на ефикасности платили додатном меморијом коју смо ангажовали, међутим, у овом сценарију није потребно памтити учитан низ, тако да меморијска сложеност неће бити значајно повећана.

```
// ucitavamo trazeni zbir
int trazeniZbir;
cin >> trazeniZbir;

// zbir prefiksa
int zbirPrefiksa = 0;

// broj segmenata sa trazenim zbirom
int broj = 0;

// broj pojavljivanja svakog vidjenog zbira prefiksa
map<int, int> zbiroviPrefiksa;
// zbir pocetnog praznog prefiksa je 0 i on se za sada pojavio
// jednom
zbiroviPrefiksa[0] = 1;

// ucitavamo elemente niza niz
int n;
```

```
cin >> n;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  // prosirujemo prefiks tekucim elementom
  zbirPrefiksa += x;

  // trazimo broj pojavljivanja vrednosti zbirPrefiksa - trazeniZbir
  // i azuriramo broj pronadjenih segmenata
  auto it = zbiroviPrefiksa.find(zbirPrefiksa - trazeniZbir);
  if (it != zbiroviPrefiksa.end())
    broj += it->second;

  // povecavamo broj pojavljivanja trenutnog zbira
  zbiroviPrefiksa[zbirPrefiksa]++;
}

cout << broj << endl;
```

Задатак: Број сегмената са различитим елементима

Дат је низ природних бројева дужине n. Написати програм којим се одређује колико има сегмената у датом низу чији су сви елементи различити. Сегмент низа чине узастопни елементи низа (њих бар 2).

Улаз: Прва линија стандардног улаза садржи природан број n ($2 \leq n \leq 50000$), број елемената низа. У свакој од n наредних линија стандардног улаза, налази по један члан низа.

Излаз: На стандардном излазу приказати у једној линији број сегмената датог низа чији су сви елементи различити.

Пример

Улаз	*Излаз*
5 1 2 2 3 6	4

Решење

Максимални сегменти за фиксирани крај

Једно прилично елегантно решење се заснива на томе да за сваку позицију анализирамо све сегменте којима је крај управо на тој позицији, а којима су сви елементи различити. Приметимо да ако неки сегмент има то својство, онда и сви сегменти иза њега (његови суфикси) такође имају то својство, док ако неки сегмент нема то својство, онда то својство нема ни један сегмент испред њега (његов префикс). Зато је довољно да пронађемо најдужи могући сегмент који се завршава на позицији $kraj$ и којем су сви елементи различити. Ако је то сегмент $[pocetak, kraj]$ онда ће такви бити и сегменти $[pocetak+1, kraj]$, ..., $[kraj-1, kraj]$. Њих је укупно $kraj - pocetak$.

Остаје питање како одредити позицију $pocetak$. Претпоставимо да већ знамо решење за претходну позицију краја, тј. претпоставимо да знамо да је $[pocetak, kraj-1]$, најдужи сегмент који има све различите елементе и који се завршава на позицији $kraj - 1$ (у старту можемо и $pocetak$ иницијализовати на нулу, а $kraj$ на јединицу, знајући да $[0, 0]$ не садржи дупликате и најдужи је такав који се завршава на позицији нула). Ако се елемент на позицији $kraj$ не садржи у том сегменту, онда је сегмент $[pocetak, kraj]$ наш тражени. Ако се садржи, онда је он сигурно једини дупликат у сегменту $[pocetak, kraj]$. Ако претпоставимо да се елемент на позицији $kraj$ у том сегменту јавља и на позицији p, тада је сегмент који тражимо $[p + 1, kraj]$, зато што сви сегменти који почињу од позиције $pocetak$, па све до позиције p садрже исти тај дупликат. Сегмент $[p+1, kraj]$ не садржи дупликате и најдужи је такав сегмент, тако да у тој ситуацији $pocetak$ треба поставити на вредност $p + 1$.

На крају, остаје питање како утврдити да ли се a_{kraj} јавља у сегменту $[pocetak, kraj-1]$ и ако се јавља, како одредити позицију p на којој се јавља. Директно решење подразумева линеарну претрагу сегмента приликом сваког проширења низа, што би знатно деградирало сложеност целог алгоритма. Боље решење је да се чува асоцијативни низ (мапа, речник) у којем се елементи низа из сегмента позиција $[pocetak, kraj-1]$ пресликавају у њихове позиције. Тада се једноставном претрагом мапе тј. речника (чија је сложеност константна или највише логаритамска) утврђује да ли се нови крајњи елемент јавља у претходном сегменту и на исти начин се одређује и његова позиција. Рецимо и да се приликом померања почетка на позицију $p+1$ сегмент скраћује, што треба да се ослика и у мапи - зато је тада потребно из мапе уклонити све елементе који се јављају у низу, на позицијама од $pocetak$, па закључно са p.

Приметимо одређену сличност овог алгоритма са оним приказаним у задатку Најкраћа подниска која садржи све дате карактере, где смо такође анализирали сегменте са фиксираним крајем и ослањали се на познато решење у којем је крај био једну позицију пре текуће.

Анализа сложености. И почетак и крај се само увећавају (никада се не умањују), што значи да се укупно изврши највише $O(n)$ корака у којима се врши претрага мапе, што значи да је сложеност највише $O(n \log n)$ – ако се користи мапа заснована на хеширању, сложеност је $O(n)$, уз могуће мало веће заузеће меморије.

```
// broj nepraznih segmenata niza sa svim razlicitim elementima
int brojSegmenataSaRazlicitimElementima(const vector<int>& a) {
  // broj elemenata niza
  int n = a.size();

  // ukupan broj segmenata niza ciji su svi elementi razliciti
  int broj = 0;

  // za svaku poziciju kraj zelimo da pronadjemo najduzi segment
  // oblika [pocetak, kraj] koji ima sve razlicite elemente

  // za svaki element u tekucem segmentu [pocetak, kraj] pamtimo
  // poziciju na kojoj se pojavljuje
  unordered_map<int, int> prethodno_pojavljivanje;

  int pocetak = 0;
  for (int kraj = 0; kraj < n; kraj++) {
    // karakter a[kraj] se vec javio u segmentu [pocetak, kraj-1]?
    if (prethodno_pojavljivanje.find(a[kraj]) != prethodno_pojavljivanje.end()) {
      // nijedan segment koji se zavrsava na poziciji kraj, a pocinje
      // pre ranijeg pojavljivanja elementa a[kraj] ne moze da ima sve
      // razlicite elemente, pa zato razmatramo samo segmente koji se
      // zavrsavaju na poziciji kraj i pocinju iza pozicije tog
      // prethodnog pojavljivanja - najduzi takav pocinje na prvoj
      // poziciji iza te pozicije
      int novi_pocetak = prethodno_pojavljivanje[a[kraj]] + 1;
      // brisemo iz segmenta sve elemente od starog do ispred novog pocetka
      // i mapu uskladjujemo sa time
      for (int i = pocetak; i < novi_pocetak; i++)
        prethodno_pojavljivanje.erase(a[i]);
      // pomeramo pocetak
      pocetak = novi_pocetak;
    }
    // prosirujemo segment elementom a[kraj], pa pamtimo poziciju
    // njegovog pojavljivanja
    prethodno_pojavljivanje[a[kraj]] = kraj;

    // [pocetak, kraj] sadrzi sve razlicite elemente i on je najduzi
    // takav koji se zavrsava na poziciji kraj
    // sigurno su takvi i [pocetak+1, kraj], ..., [kraj-1, kraj]
    // njih ima (kraj - pocetak) i taj broj dodajemo na ukupan broj
    // trazenih segmenata
```

```
    broj += kraj - pocetak;
  }

  // vracamo ukupan broj pronadjenih segmenata
  return broj;
}
```

4.2 Стек

Стек представља колекцију података у коју се подаци додају по LIFO (енгл. last-in-first out) принципу - елемент се може додати и скинути само на врха стека.

У језику C++ стек се реализује класом `stack<T>` где `T` представља тип елемената на стеку. За њено коришћењем потребно је укључити заглавље `<stack>`. Подржане су следеће методе (све сложености $O(1)$):

- `push` - поставља дати елемент на врх стека
- `pop` - скида елемент са врха стека (под претпоставком да стек није празан). Нагласимо да је ова метода типа `void` и да не враћа уклоњени елемент.
- `top` - очитава елемент на врху стека (под претпоставком да стек није празан)
- `empty` - проверава да ли је стек празан
- `size` - враћа број елемената на стеку.

Ако се на стеку чувају уређени парови или н-торке, тада се уместо методе `push`, може користити метода `emplace`, којој се само редом наводе елементи пара тј. н-торке (није потребно посебно позивати функцију за креирање пара тј. н-торке, што је неопходно када се користи `push`).

Стек у језику C++ је заправо само адаптер око неке колекције података (подразумеано вектора) који корисника тера да поштује правила приступа стеку и спречава да направи операцију која над стеком није допуштена (попут приступа неком елементу испод врха).

Задатак: Линије у обратном редоследу

Напиши програм који исписује све линије које се учитају са стандардног улаза у обратном редоследу од редоследа учитавања.

Улаз: Са стандардног улаза се учитавају линије текста, све до краја улаза.

Излаз: На стандардни излаз исписати учитане линије у обратном редоследу.

Пример

Улаз	*Излаз*
`zdravo`	`dan`
`svete`	`dobar`
`dobar`	`svete`
`dan`	`zdravo`

Решење

Једно од могућих решења је да се све учитане линије сместе на стек, а да се затим испишу узимајући једну по једну са стека. Пошто стек функционише по принципу LIFO (last in first out, тј. онај који последњи уђе, први излази), редослед ће бити обрнут (најкасније додата линија биће прва скинута и исписана, док ће прва постављена линија бити скинута и исписана последња).

```
stack<string> s;
string linija;
while (getline(cin, linija))
  s.push(linija);
while (!s.empty()) {
  cout << s.top() << endl;
  s.pop();
}
```

Задатак: Историја веб-прегледача

Прегледач веба памти историју посећених сајтова и корисник има могућност да се враћа унатраг на сајтове које је раније посетио. Написати програм који симулира историју прегледача тако што се учитавају адресе посећених сајтова (свака у посебном реду), а када се учита ред у коме пише `back` прегледач се враћа на последњу посећену страницу. Ако се наредбом `back` вратимо на почетну страницу, исписати `-`. Ако се на почетној страници изда наредба `back`, остаје се на почетној страници. Програм треба да испише све сајтове које је корисник посетио.

Улаз: Са стандардног улаза се учитавају веб-адресе, свака у посебној линији, њих највише 1000.

Излаз: На стандардни излаз исписати редом сајтове који се посећују.

Пример

Улаз	*Излаз*
`http://www.google.com`	`http://www.google.com`
`http://www.rts.rs`	`http://www.rts.rs`
`back`	`http://www.google.com`
`http://www.petlja.org`	`http://www.petlja.org`
`http://www.matf.bg.ac.rs`	`http://www.matf.bg.ac.rs`
`back`	`http://www.petlja.org`
`back`	`http://www.google.com`
`back`	`-`
`back`	`-`

Решење

Историја прегледача се понаша као стек јер се елементи само могу скидати и постављати на један крај историје (као последње посећени) и са тог краја се могу и скидати.

```
stack<string> istorija;
string linija;
while (getline(cin, linija)) {
  if (linija == "back") {
    if (!istorija.empty())
      istorija.pop();
    if (!istorija.empty())
      cout << istorija.top() << endl;
    else
      cout << "-" << endl;
  }
  else {
    cout << linija << endl;
    istorija.push(linija);
  }
}
```

Види другачија решења овог задатка.

Задатак: Push-pop реконструкција

Током рада са стеком укупно n пута је извршена операција `push` којом се нека вредност поставља на врх стека и укупно n пута је извршена операција `pop` којом је елемент скинут са врха стека. Ако је познат низ бројева који су редом били аргументи операције `push` и низ бројева који су редом добијани као резултат операције `pop`, напиши програм који одређује редослед операција `push` и `pop`.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^5$), а затим два низа од по n бројева раздвојених размацима. Претпоставити да су и у једном и у другом низу сви елементи различити.

Излаз: На стандардни излаз исписати редослед операција `push` и `pop` или `-`, ако такав редослед операција није могуће пронаћи за задате низове.

Пример 1

Улаз	*Излаз*
5	push
1 2 3 4 5	push
5 4 3 2 1	push
	push
	push
	pop
	pop
	pop
	pop
	pop

Пример 2

Улаз	*Излаз*
5	push
1 2 3 4 5	push
3 2 5 4 1	push
	pop
	pop
	push
	push
	pop
	pop
	pop

Пример 3

Улаз	*Излаз*
5	-
1 2 3 4 5	
5 4 3 1 2	

Решење

Задатак ћемо решити симулирањем операција са стеком. Пролазимо са два показивача кроз низове push и pop.

- Ако је текући елемент на врху стека којег одржавамо једнак текућем елементу у низу pop, тада вршимо операцију pop. Заиста, то је једина могућност јер ако бисмо извршили операцију push уместо pop, на врху стека би се појавио неки елемент различит од тог који је тренутно на врху и он би морао бити уклоњен пре овог који је тренутно на врху, што значи да наредна операција pop не би вратила елемент који је тренутно на реду у низу pop.
- Ако текући елемент на врху стека којег одржавамо није једнак текућем елементу у низу pop или ако је стек празан, тада проверавамо да ли постоји још елемената у низу push.
  - Ако постоји још елемената, вршимо операцију push и наредни елемент низа push постављамо на стек. Заиста, то је једина могућност, јер ако ако је стек празан, pop није могућ, а ако стек није празан и ако бисмо извршили операцију pop уместо push, тада би резултат примене операције pop био различит од онога наметнутог учитаним низом pop.
  - Ако је низ push празан, тада решење не постоји. Заиста, на основу претходне дискусије знамо да није могућа ни операција pop, нити је могућа операција push.

```
// ucitavamo nizove vrednosti push i pop
int n;
cin >> n;
vector<int> push(n);
// ...
vector<int> pop(n);
// ...
// stek ciji rad simuliramo
stack<int> stek;
// niz naredbi push i pop (potrebno za ispis resenja)
vector<string> naredbe;
naredbe.reserve(2*n);
// da li postoji resenje
bool moze = true;
// dok ne obradimo kompletno oba niza
int push_i = 0, pop_i = 0;
while (push_i < n || pop_i < n) {
  if (!stek.empty() && stek.top() == pop[pop_i]) {
    // ako se na vrhu steka nalazi naredni element koga treba
    // skinuti, skidamo ga
    naredbe.push_back("pop");
    stek.pop();
    pop_i++;
  } else if (push_i < n) {
    // u suprotnom, ako postoji naredni element koji treba
    // postaviti na stek, postavljamo ga
    naredbe.push_back("push");
```

```
    stek.push(push[push_i]);
    push_i++;
  } else {
    // posto ne mozemo ni postaviti ni skinuti element sa steka,
    // resenje ne postoji
    moze = false;
    break;
  }
}

// ispisujemo rezultat
if (moze)
  for (const string& s : naredbe)
    cout << s << endl;
else
  cout << "-" << endl;
```

Задатак: Линијски едитор

Едитор омогућава куцање једне линије текста. На почетку је линија празна и курсор се налази на почетку. Корисник може да куца слова, помера курсор лево и десно и брише слово (испред или иза курсора). Курсор никада не може да испадне ван граница текста (када се притисне тастер лево док је курсор на почетку или десно док је курсор на крају текста, он се не помера).

Улаз: Са стандардног улаза се уноси ниска карактера која описује акције корисника. Акције су следеће:

- `iX` – корисник је откуцао карактер `X` (insert `X`)
- `<` – корисник је притиснуо тастер лево
- `>` – корисник је притиснуо тастер десно
- `b` – корисник је притиснуо тастер `backspace` за брисање карактера иза курсора
- `d` – корисник је притиснуо тастер `delete` за брисање карактера испред курсора

Излаз: На стандардни излаз исписати линију текста добијену извршавањем свих команди.

Пример 1

Улаз *Излаз*

```
iaib<bic>>    cb
```

Објашњење

```
Текст           Наредба   Значење
|               ia        куцање карактера a
a|              ib        куцање карактера b
ab|             <         тастер лево
a|b             b         тастер за брисање карактера иза курсора
|b              ic        куцање карактера c
c|b             >         тастер десно
cb|             >         тастер десно
cb|
```

Пример 2

Улаз

```
izidiriaivio<<<<<dbib<<<i!>>>>>i.
```

Излаз

```
!bravo.
```

Решење

Решење грубом силом одржава ниску карактера у коју умеће и из које брише карактере. У језику C++ позицију курсора је најједноставније одржавати у виду итератора (он, на пример, може да указује на карактер

који је непосредно иза позиције курсора, тј. на `end` када је курсор на крају ниске). Уметање се може вршити методом `insert`, а брисање методом `erase`.

Анализа сложености. Сложеност операција брисања и уметања карактера је линеарна у односу на дужину ниске, па је укупна сложеност оваквог приступа у најгорем случају квадратна (најгори случај је ако се курсор често помера тако да се уметање и брисање врше негде близу почетка или бар средине ниске).

```
string str = "";

string naredbe;
cin >> naredbe;
int i = 0;
auto it = begin(str);
while (i < naredbe.size()) {
  char naredba = naredbe[i++];
  if (naredba == '<') {
    if (it > begin(str))
      it--;
  } else if (naredba == '>') {
    if (it < end(str))
      it++;
  } else if (naredba == 'i') {
    char c = naredbe[i++];
    it = str.insert(it, c);
    it++;
  } else if (naredba == 'b') {
    if (it > begin(str)) {
      it--;
      it = str.erase(it);
    }
  } else if (naredba == 'd') {
    if (it < end(str))
      it = str.erase(it);
  }
}

cout << str << endl;
```

Ефикасније решење се може добити ако се уместо ниске чува листа карактера. У језику C++ постоје две колекције реализоване помоћу повезаних листа. Колекција `forward_list<T>` представља једноструко повезане листе, док колекција `list<T>` представља двоструко повезане листе. За њихово коришћење је потребно укључити заглавља `<forward_list>` тј. `<list>`. Основне методе за рад са листама су следеће (све су сложености $O(1)$):

- `begin()` - враћа итератор на почетак листе
- `end()` - враћа итератор на крај листе
- `insert(it, x)` - умеће елемент испред датог итератора и враћа итератор на елемент који је уметнут
- `erase(it)` - брише елемент на који указује дати итератор и враћа итератор иза обрисаног елемента (што може бити и `end`, ако је обрисан последњи елемент).

На итератор код колекције `forward_list` могуће је примењивати оператор `++`, чиме се он помера на наредни елемент листе, а код колекције `list` и оператор `--`, чиме се он помера на претходни елемент листе. И ове операције су сложености $O(1)$.

Позицију курсора у овом задатку је најједноставније одржавати у виду итератора (он, на пример, може да указује на карактер који је непосредно иза позиције курсора, тј. на `end` када је курсор на крају ниске). Пошто се курсор помера у оба смера, потребно је користити колекцију `list<char>`, а не `forward_list<char>`. Уметање се онда може вршити методом `insert`, а брисање методом `erase`.

Анализа сложености. Пошто су код листе операције уметања и брисања, као и померања итератора за једно место константне сложености, овај алгоритам је линеарне сложености.

```
string naredbe;
cin >> naredbe;
int i = 0;
list<char> str;
auto it = begin(str);
while (i < naredbe.size()) {
  char naredba = naredbe[i++];
  if (naredba == '<') {
    if (it != begin(str))
      it--;
  } else if (naredba == '>') {
    if (it != end(str))
      it++;
  } else if (naredba == 'i') {
    char c = naredbe[i++];
    it = str.insert(it, c);
    it++;
  } else if (naredba == 'b') {
    if (it != begin(str)) {
      it--;
      it = str.erase(it);
    }
  } else if (naredba == 'd') {
    if (it != end(str))
      it = str.erase(it);
  }
}
cout << string(begin(str), end(str)) << endl;
```

Могуће је и веома једноставно и елегантно решење коришћењем два стека. У једном стеку се чувају карактери лево од тренутне позиције курсора (тако да се на врху налази карактер најближи курсору), а у другом карактери десно од тренутне позиције курсора (тако да се на врху поново налази карактер најближи курсору). Куцање карактера се тада реализује уметањем карактера на леви стек, померање налево се реализује пребацивањем карактера са врха левог, на врх десног стека, померање надесно ради обратно, док се брисање карактера иза курсора своди на скидање елемента са врха левог, а испред курсора на скидање елемената са врха десног стека. Ако је неки од стекова празан, операције скидања елемента са њега се просто прескачу.

Анализа сложености. Пошто су операције скидања и додавања елемената на врх стека константне сложености, укупна сложеност овог алгоритма је линеарна.

```
string stekUString(stack<char>& stek) {
  string str = "";
  while (!stek.empty()) {
    str += stek.top();
    stek.pop();
  }
  return str;
}


int main() {
  stack<char> levo, desno;
  string naredbe;
  cin >> naredbe;
  int i = 0;
  while (i < naredbe.size()) {
    char naredba = naredbe[i++];
    if (naredba == '<') {
      if (!levo.empty()) {
        desno.push(levo.top());
```

```
        levo.pop();
      }
    } else if (naredba == '>') {
      if (!desno.empty()) {
        levo.push(desno.top());
        desno.pop();
      }
    } else if (naredba == 'i') {
      levo.push(naredbe[i++]);
    } else if (naredba == 'b') {
      if (!levo.empty())
        levo.pop();
    } else if (naredba == 'd') {
      if (!desno.empty())
        desno.pop();
    }
  }
  string str = stekUString(levo);
  reverse(begin(str), end(str));
  str += stekUString(desno);
  cout << str << endl;
  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Сортирање бројева

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Нерекурзивно брзо сортирање

Једна веома важна употреба стека је за реализацију рекурзије. Током извршавања рекурзивних функција, на стек се смештају вредности локалних променљивих и аргумената сваког активног позива функције. Рекурзију увек можемо уклонити и уместо системског стека можемо ручно одржавати стек са тим подацима. Прикажимо ову технику уклањања рекурзије на примеру нерекурзивне имплементације алгоритма брзог сортирања. Нагласимо да је код алгоритма QuickSort дубина рекурзије мала (она логаритамски зависи од броја елемената низа), па се њеном елиминацијом не добија ништа значајно.

На стеку ћемо чувати аргументе рекурзивних позива функције сортирања. На почетку је то пар индекса $(0, n1)$. Главна петља се извршава све док се стек не испразни и у њој се обрађује пар индекса који се скида са врха стека. Уместо рекукрзивних позива њихове ћемо аргументе постављати на врх стека и чекати да они буду обрађени у некој од наредних итерација петље. Приметимо да се аргументи другог рекурзивног позива обрађују тек када се у потпуности реши потпроблем који одговара првом рекурзивном позиву, што одговара понашању функције када је заиста имплементирана рекурзивно.

Напомена. Рецимо и да је ова техника општа и да се рекурзија увек може елиминисати на овај начин. За разлику од тога, елиминисање специфичних облика рекурзије (попут, на пример, репне) није увек применљиво, али када јесте, доводи до боље меморијске (па и временске) ефикасности јер се не користи стек.

```
void quick_sort(vector<int>& a) {
  // duzina niza
  int n = a.size();
  // stek na kome čuvamo argumente rekurzivnih poziva
  stack<pair<int, int>> sortirati;
  // sortiranje kreće od obrade celog niza tj. pozicija (0, n-1)
  sortirati.emplace(0, n - 1);
  while (!sortirati.empty()) {
    // skidamo par (l, d) sa vrha steka
```

```
    auto p = sortirati.top();
    int l = p.first, d = p.second;
    sortirati.pop();
    // obrađujemo par (l, d) na isti način kao u rekurzivnoj implementaciji
    if (d - l < 1)
      continue;
    int k = l;
    for (int i = l+1; i <= d; i++)
      if (a[i] < a[l])
        swap(a[++k], a[i]);
    swap(a[k], a[l]);
    // umesto rekurzivnih poziva njihove argumente
    // postavljamo na stek
    sortirati.emplace(k+1, d);
    sortirati.emplace(l, k-1);
  }
}
```

Види другачија решења овог задатка.

Задатак: Вредност постфиксног израза

Префиксна нотација се понекад назива и пољска нотација, а постфиксна нотација се понекад назива и обратна пољска нотација (енгл. reverse polish notation, RPN) у част логичара Јана Лукашијевича који ју је изумео. Она подразумева да се бинарни оператори уместо између операнада записују након њих. На пример, уместо `3 + 5`, писаћемо `3 5 +`. Напиши програм који одређује вредност постфиксно записаног израза.

Улаз: Са стандардног улаза се учитава постфиксно записан израз који садржи једноцифрене бројеве и операторе `+` и `*` (без размака).

Излаз: На стандардни излаз исписати вредност учитаног израза.

Пример 1

Улаз	*Излаз*
`12+3*`	`9`

Објашњење

Постфиксно је записан израз `(1+2)*3`.

Пример 2

Улаз

```
11+2*345+*+
```

Излаз

```
31
```

Објашњење

Постфиксно је записан израз `(1+1)*2+3*(4+5)`.

Решење

Велика предност постфиксно записаних израза је то што им се вредност веома једноставно израчунава уз помоћ стека. Када наиђемо на број постављамо га на врх стека. Када наиђемо на оператор, скидамо две вредности са врха стека, примењујемо на њих одговарајућу операцију и резултат постављамо на врх стека. Вредност целог израза се на крају налази на врху стека.

Пример. Нпр. за израз `34+5*` на стек постављамо `3`, затим `4`, након тога те две вредности уклањамо и постављамо `7`, затим постављамо `5` и на крају уклањамо `7` и `5` и мењамо их са `35`.

```
bool jeOperator(char c) {
  return c == '+' || c == '*';
}
```

```
int primeniOperator(char op, int op1, int op2) {
  int v = 0;
  switch(op) {
     case '+': v = op1 + op2; break;
     case '*': v = op1 * op2; break;
  }
  return v;
}

int vrednost(const string& izraz) {
   stack<int> st;
   for (char c : izraz) {
      if (isdigit(c))
        st.push(c - '0');
      else if (jeOperator(c)) {
         int op2 = st.top(); st.pop();
         int op1 = st.top(); st.pop();
         st.push(primeniOperator(c, op1, op2));
      }
   }
   return st.top();
}
```

Задатак: Превођење потпуно заграђеног израза у постфиксни облик

Написати програм који исправан инфиксни аритметички израз који има заграде око сваке примене бинарног оператора преводи у постфиксни облик. Једноставности ради претпоставити да су сви операнди једноцифрени бројеви и да се јављају само операције сабирања и множења.

Улаз: Једина линија стандардног улаза садржи исправан, потпуно заграђен израз.

Излаз: На стандардни излаз исписати тражени постфиксни облик.

Пример

Улаз	*Излаз*
`(((3*5)+(7+(2*1)))*4)`	`35*721*++4*`

Решење

Чињеница да је израз потпуно заграђен олакшава израчунавање, јер нема потребе да водимо рачуна о приоритету и асоцијативности оператора. Такви изрази се описују наредном, веома једноставном граматиком.

```
<izraz> :: <cifra>
<izraz> :: '(' <izraz> '+' <izraz> ')'
<izraz> :: '(' <izraz> '*' <izraz> ')'
```

Један начин да се приступи решавању проблема је да се примени индуктивно-рекурзивни приступ. Обрада структурираног улаза рекурзивним функцијама се назива *рекурзивни спуст* и детаљно се изучава у курсевима превођења програмских језика. Дефинишемо рекурзивну функцију чији је задатак да преведе део ниске који представља исправан инфиксни израз. Он може бити или број, када је превођење тривијално јер се он само препише на излаз или израз у заградама. У овом другом случају читамо отворену заграду, затим рекурзивним позивом преводимо први операнд, након тога читамо оператор, затим рекурзивним позивом преводимо други операнд, након тога читамо затворену заграду и исписујемо оператор који смо прочитали (он бива исписан непосредно након превода својих операнада).

Променљива `i` мења своју вредност кроз рекурзивне позиве. Стога ћемо је преносити по референци тако да представља и улазну и излазну величину функције. Задатак функције је да прочита израз који почиње на позицији i, да га преведе у постфиксни облик и да променљиву `i` промени тако да њена нова вредност i' указује на позицију ниске непосредно након израза који је преведен.

```
// Prevodi deo izraza od pozicije i u postfiksni oblik i rezultat
// nadovezuje na nisku postfiks. Po zavrsetku rada funkcije,
```

```
// promenljiva i ukazuje na poziciju iza prevedenog izraza.
void prevedi(const string& izraz, int& i, string& postfiks) {
  if (isdigit(izraz[i]))
    postfiks += izraz[i++];
  else {
    // preskačemo otvorenu zagradu
    i++;
    // prevodimo prvi operand
    prevedi(izraz, i, postfiks);
    // pamtimo operator
    char op = izraz[i++];
    // prevodimo drugi operand
    prevedi(izraz, i, postfiks);
    // preskačemo zatvorenu zagradu
    i++;
    // ispisujemo upamćeni operator
    postfiks += op;
  }
}

// prevodi potpuno zagradjen izraz u postfiksni oblik
string prevedi(const string& izraz) {
  string postfiks = "";
  int i = 0;
  prevedi(izraz, i, postfiks);
  return postfiks;
}
```

Решење техником рекурзивног спуста се може прерадити тако да се добије нерекурзивна имплементација. Да бисмо се ослободили рекурзије, потребно је да употребимо стек. Кључна опаска је да се у стек оквиру функције, пре рекурзивног позива за превођење другог операнда памти оператор. Ово нам сугерише да нам је за нерекурзивну имплементацију неопходно да одржавамо стек на који ћемо смештати операторе. Када наиђемо на број преписујемо га на излаз, када наиђемо на оператор стављамо га на стек, а када наиђемо на затворену заграду скидамо и исписујемо оператор са врха стека.

Пример. Размотримо, на пример, израз `((3+4)*(5+2))`

- Први карактер је отворена заграда коју прескачемо.
- Наредни карактер је отворена заграда коју прескачемо.
- Наредни карактер је цифра, коју исписујемо (до сада је исписано `3`).
- Наредни карактер је оператор `+`, који постављамо на стек. Стек је сада `+`.
- Наредни карактер је цифра, коју исписујемо (до сада је исписано `34`).
- Наредни карактер је затворена заграда, па исписујемо оператор са врха стека (до сада је исписано `34+`). Стек је сада празан.
- Наредни карактер је оператор `*` који постављамо на стек. Стек је сада `*`.
- Наредни карактер је отворена заграда коју прескачемо.
- Наредни карактер је цифра, коју исписујемо (до сада је исписано `34+5`)
- Наредни карактер је оператор `+` који постављамо на стек. Стек је сада `*+`.
- Наредни карактер је цифра, коју исписујемо (до сада је исписано `34+52`)
- Наредни карактер је затворена заграда, па исписујемо оператор са врха стека (до сада је исписано `34+52+`). Стек је сада `*`.
- Наредни карактер је затворена заграда, па исписујемо оператор са врха стека (до сада је исписано `34+52+*`). Стек је сада празан.

```
bool jeOperator(char c) {
  return c == '+' || c == '*';
}

string prevedi(const string& izraz) {
  string postfiks;
  stack<char> operatori;
  for (char c : izraz) {
    if (isdigit(c))
      postfiks += c;
    else if (c == ')') {
      postfiks += operatori.top();
      operatori.pop();
    } else if (jeOperator(c))
      operatori.push(c);
  }
  return postfiks;
}
```

Задатак: Вредност израза

Написати програм којим се израчунавају и приказују вредности датих аритметичких израза. Сваки израз је исправно задат, састоји се од природних бројева и операција +, -, * и / (целобројно дељење). Коришћењем проширене Бекусове нотације (EBNF), синтаксу израза можемо описати на следећи начин:

```
<izraz>  ::= <term> {<operacija1> <term>}
<term>   ::= <faktor> {<operacija2> <faktor>}
<faktor> ::= <broj> | '(' <izraz> ')'
<broj>   ::= <cifra> {<cifra>}
<cifra> ::= '0' | '1' | ... | '9'
<operacija1> ::= '+' | '-'
<operacija2> ::= '*' | '/'
```

Улаз: У свакој линији стандарног улаза налази се исправан израз (израз не садржи размаке).

Излаз: Свака линија стандардног излаза садржи редом вредности израза датих на стандардном улазу, свака вредност у посебној линији. Ако израз није дефинисан, због дељења 0, приказати поруку `deljenje nulom`.

Пример

Улаз	*Излаз*
`1+2*3-4`	`3`
`2*3-5*(100-8*12)`	`-14`
`123-43*(12-3*5)/(17-35/2)`	`deljenje nulom`
`12/5+2`	`4`

Решење

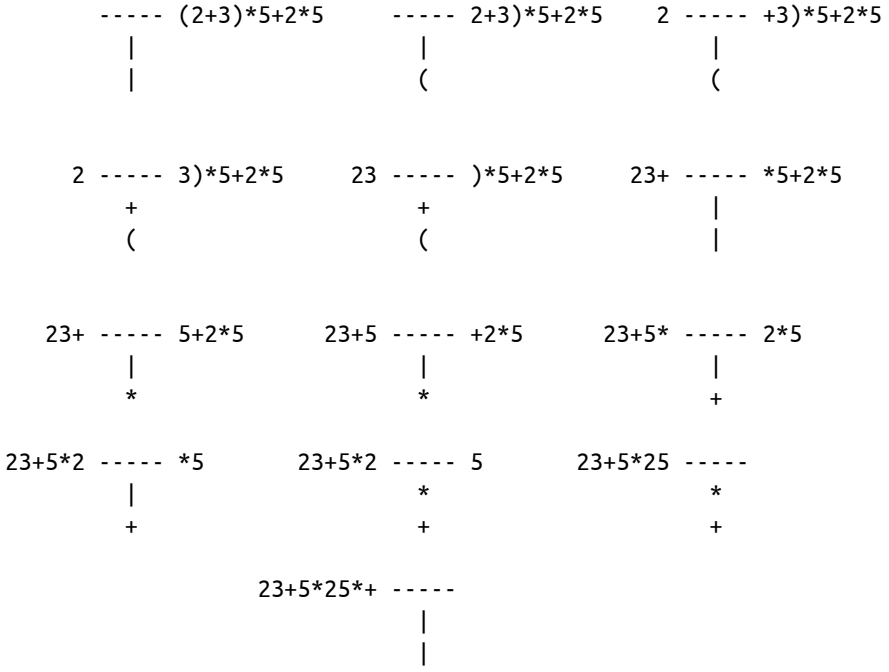
Израчунавање вредности израза је опет комбинација превођења у постфиксни облик и израчунавања вредности постфиксног израза.

Проблем се решава слично као код потпуно заграђених израза, али овај пут се мора обраћати пажња на приоритет и асоцијативност оператора. Решење се може направити рекурзивним спустом, али се тиме нећемо бавити у овом курсу. Кључна дилема је шта радити у ситуацији када се прочита `op2` у изразу облика `i1 op1 i2 op2 i3` где су `i1`, `i2` и `i3` три израза (било броја било израза у заградама), а `op1` и `op2` два оператора. У том тренутку на излазу ће се налазити израз `i1` преведен у постфиксни облик и иза њега израз `i2` преведен у постфиксни облик, док ће се оператор `op1` налазити на врху стека оператора. Уколико `op1` има већи приоритет од оператора `op2` или уколико им је приоритет исти, али је асоцијативност лева, тада је потребно прво израчунавати израз `i1 op1 i2` тиме што се оператор `op1` са врха стека пребаци на излаз. У супротном (ако `op2` има већи приоритет или ако је приоритет исти, а асоцијативност десна) оператор `op1` остаје на стеку и изнад њега се поставља оператор `op2`.

Ово је један од многих алгоритама које је извео Едсгер Дејкстра и назива се на енглеском језику *Shunting yard algortihm*, што би се могло слободно превести као алгоритам сортирања железничких вагона. Замислимо да

израз треба да пређе са једног на други крај пруге. На прузи се налази споредни колосек (пруга је у облику слова T и споредни колосек је усправна црта). Делови израза прелазе са десног на леви крај (замислимо да иду по горњој ивици слова T). Бројеви увек прелазе директно. Оператори се увек задржавају на споредном колосеку, али тако да се пре него што оператор уђе на споредни колосек са њега на излаз пребацују сви оператори који су вишег приоритета у односу на текући или имају исти приоритет као текући а лево су асоцијативни. И отворене заграде се постављају на споредни колосек, а када наиђе затворена заграда са споредног колосека се уклањају сви оператори до отворене заграде. Када се исцрпи цео израз на десној страни, сви оператори са споредног колосека се пребацују на леву страну. Јасно је да споредни колосек има понашање стека, тако да имплементацију можемо направити коришћењем стека на који ћемо стављати операторе.

Пример. Илуструјмо ову железничку аналогију једним примером.




```
// provera da li je karakter aritmeticki operator
bool jeOperator(char c) {
  return c == '+' || c == '-' || c == '*' || c == '/';
}

// prioritet datog operatora
int prioritet(char c) {
  if (c == '+' || c == '-')
    return 1;
  else if (c == '*' || c == '/')
    return 2;
  // greska
  return 0;
}

// primenjuje datu operaciju na dve vrednosti na vrhu steka,
// zamenjujući ih sa rezultatom primene te operacije
// vracamo informaciju o tome da li operator uspesno primenjen ili je
// doslo do deljenja nulom
bool primeni(stack<char>& operatori, stack<int>& vrednosti) {
  // operator se nalazi na vrhu steka operatora
  char op = operatori.top(); operatori.pop();
  // operandi se nalaze na vrhu steka operatora
  int op2 = vrednosti.top(); vrednosti.pop();
  int op1 = vrednosti.top(); vrednosti.pop();
```

```
  // izracunavamo vrednost izraza
  int v = 0;
  if (op == '+') v = op1 + op2;
  else if (op == '-') v = op1 - op2;
  else if (op == '*') v = op1 * op2;
  else if (op == '/') {
    // deljenje nulom
    if (op2 == 0)
      return false;
    v = op1 / op2;
  }
  // postavljamo ga na stek operatora
  vrednosti.push(v);
  // operator je uspesno primenjen
  return true;
}


// izracunavamo vrednost izraza
// vracamo informaciju o tome da li je vrednost uspesno izracunata ili
// je doslo do deljenja nulom
bool vrednost(const string& izraz, int& v) {
  stack<int> vrednosti;
  stack<char> operatori;

  // analiziramo sve karaktere u ulaznom izrazu
  int i = 0;
  while (i < izraz.length()) {
    if (isdigit(izraz[i])) {
      // brojevne konstante postavljamo na stek
      v = izraz[i] - '0';
      i++;
      while (i < izraz.length() && isdigit(izraz[i]))
        v = 10 * v + (izraz[i++] - '0');
      vrednosti.push(v);
    } else if (izraz[i] == '(') {
      // otvorene zagrade postavljamo na stek
      operatori.push('(');
      i++;
    } else if (izraz[i] == ')') {
      // izračunavamo vrednost izraza u zagradi
      while (operatori.top() != '(')
        if (!primeni(operatori, vrednosti))
          return false;
      // uklanjamo otvorenu zagradu
      operatori.pop();
      i++;
    } else if (jeOperator(izraz[i])) {
       // obrađujemo sve prethodne operatore višeg prioriteta
       while (!operatori.empty() && jeOperator(operatori.top()) &&
              prioritet(operatori.top()) >= prioritet(izraz[i]))
         if (!primeni(operatori, vrednosti))
           return false;
       // stavljamo operator na stek
       operatori.push(izraz[i]);
       i++;
    }
  }
```

```
  // izračunavamo sve preostale operacije
  while (!operatori.empty())
    if (!primeni(operatori, vrednosti))
      return false;

  // vrednost izraza se nalazi na vrhu steka
  v = vrednosti.top();
  return true;
}
```

4.3 Ред

Ред представља колекцију података у коју се подаци додају по FIFO (енгл. first-in-first out) принципу – елемент се увек узима са почетка, а додаје на крај реда.

У језику C++ ред се реализује класом `queue<T>` где `T` представља тип елемената на стеку. За њено коришћењем потребно је укључити заглавље `<stack>`. Подржане су следеће методе (све сложености $O(1)$):

- `push` - поставља дати елемент на крај реда
- `pop` - скида елемент са почетка реда (под претпоставком да ред није празан). Нагласимо да је ова метода типа `void` и да не враћа уклоњени елемент.
- `front` - очитава елемент на почетку реда (под претпоставком да ред није празан)
- `empty` - проверава да ли је ред празан
- `size` - враћа број елемената у реду

Ако се у реду чувају уређени парови или н-торке, тада се уместо методе `push`, може користити метода `emplace`, којој се само редом наводе елементи пара тј. н-торке (није потребно посебно позивати функцију за креирање пара тј. н-торке, што је неопходно када се користи `push`).

Ред у језику C++ је заправо само адаптер око неке колекције података (подразумеано реда са два краја) који корисника тера да поштује правила приступа реду и спречава да направи операцију која над редом није допуштена (попут приступа неком елементу који није на почетку).

4.3.1 Ред са два краја

Уопштење представља ред са два краја који допушта да се елементи и додају и узимају са било ког краја реда (та структура података заправо комбинује и функционалност стека и функционалност реда).

У језику C++ могуће је користити структуру `deque<T>`. За њено коришћење потребно је укључити заглавље `<deque>`. Подржане су следеће операције (све сложености $O(1)$).

- `push_front` - додавање елемента на почетак
- `push_back` - додавање елемента на крај
- `front` - читање елемента са почетка (под претпоставком да ред није празан)
- `back` - читање елемента са краја (под претпоставком да ред није празан)
- `pop_front` - уклањање елемента са почетка (под претпоставком да ред није празан). Нагласимо да је ова метода типа `void` и да не враћа уклоњени елемент.
- `pop_back` - уклањање елемента са краја (под претпоставком да ред није празан). Нагласимо да је ова метода типа `void` и да не враћа уклоњени елемент.
- `empty` - провера да ли је ред празан
- `size` - број елемената у реду

Интересантно, захваљујући специфичном начину имплементације, ова структура података подржава и оператор индексног приступа којим се елемент на датој позицији може прочитати или изменити у времену $O(1)$.

Задатак: Сегмент највећег просека

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Приметимо да је проблем налажења сегмента дужине k чији је просек највећи, еквивалентан проблему налажења сегмента дужине k највећег збира. Просек сегмента добијамо дељењем суме сегмента са дужином сегмента, која је у овом задатку константа и износи k, тако да је просек највећи када је збир највећи.

Смањивање употребљене меморије коришћењем реда

Претходно решење је временски ефикасно, али се користи превише меморије. Наиме, није неопходно у меморији истовремено чувати све елементе низа, већ је довољно чувати само елементе текућег сегмента (ово може бити битно у случајевима када је меморија критичан ресурс и када је k знатно мање од n). Приликом учитавања сваког новог елемента почетни елемент тренутног сегмента се уклања, а последњи елемент се додаје на крај сегмента. Ово указује на то да је за чување текућег сегмента погодна структура података ред, која нам омогућава да додајемо елементе на крај и уклањамо елементе са почетка (први елемент који је додат први бива и уклоњен из реда). У језику C++ можемо употребити колекцију `queue`. Елементе у ред додајемо методом `push`, уклањамо их методом `pop` (она не враћа вредност). Елемент на почетку реда се може очитати методом `front`.

Анализа сложености. У реду се истовремено чува највише k елемената, па је меморијска сложеност $O(k)$, што може бити доста мање него $O(n)$. Пошто се сваки елемент највише једном убацује и највише једном избацује из реда, а убацивање и избацивање из реда је операција константне сложености, укупна временска сложеност алгоритма је $O(n)$.

```
// broj elemenata niza i duzina segmenta
int n, k;
cin >> k >> n;

// red u kojem u svakom trenutku cuvamo tekuci segment
queue<double> q;

// ucitavamo prvi segment duzine k, smestamo elemente u red i
// racunamo mu sumu
double suma = 0.0;
for (int i = 0; i < k; i++) {
  double x; cin >> x;
  q.push(x);
  suma += x;
}

// trenutna maksimalna suma segmenta i indeks njenog pocetka
int maxPocetak = 0;
double maxSuma = suma;

for (int i = 1; i <= n-k; i++) {
  // ucitavamo naredni element
  double x; cin >> x;
  // azuriramo sumu
  suma = suma - q.front() + x;
  // menjamo "najstariji" element u redu
  q.pop(); q.push(x);
  // ako je potrebno, azuriramo maksimum
  if (suma >= maxSuma) {
    maxSuma = suma;
    maxPocetak = i;
  }
}

// ispisujemo pocetak poslednjeg segmenta sa maksimalnom sumom
// (ujedno i prosekom)
cout << maxPocetak << endl;
```

Задатак: Јосифов проблем

Ђаци седе у кругу обележени бројевима од 0 до $n-1$ и играју се разбрајалице тако да у сваком бројању један ђак испадне. Бројање креће од ђака 0 и сваки m-ти ђак испада. Напиши програм који одређује који ђак ће остатати последњи.

Улаз: У првој линији стандардног улаза налази се почетни број ђака n ($1 \leq n \leq 10^5$), а у другом дужина бројалице m ($2 \leq m \leq n$).

Излаз: На стандардни излаз исписати број преосталог ђака.

Пример

Улаз	*Излаз*
8 3	6

Објашњење

Ђаци који седе у кругу на почетку и након сваког испадања су:

```
0 1 2 3 4 5 6 7
0 1 3 4 5 6 7
0 1 3 4 6 7
1 3 4 6 7
1 3 6 7
3 6 7
3 6
6
```

Решење

Структура података која допушта ефикасно избацивање елемената из средине је листа (нпр. двоструко повезана). Кружно кретање по листи можемо остварити тако што након сваког померања итератора проверимо да ли смо стигли до краја листе и ако јесмо, итератор ручно поново поставимо на почетак листе.

```
// efekat postfiksnog uvecavanja iteratora u kruznoj listi tj. efekat it++
// iterator se pomera na sledece mesto, ali se vraca polazna vrednost
list<int>::iterator uvecaj(list<int>& lista, list<int>::iterator& it) {
  list<int>::iterator polazni = it;
  it++;
  if (it == lista.end())
    it = lista.begin();
  return polazni;
}

int josif(int n, int m) {
  list<int> lista;
  for (int i = 0; i < n; i++)
    lista.emplace_back(i);

  auto it = lista.begin();
  while (lista.size() > 1) {
    for (int i = 0; i < m - 1; i++)
      uvecaj(lista, it);
    lista.erase(uvecaj(lista, it));
  }

  return *it;
}
```

Кружну листу можемо једноставно реализовати коришћењем реда. У сваком кораку бројања једног ђака са почетка реда ћемо пребацити на крај реда. Након пребацивања $m-1$ ученика, оног који је на почетку реда трајно избацујемо.

```
int josif(int n, int m) {
  queue<int> red;
  for (int i = 0; i < n; i++)
    red.push(i);
  while (red.size() > 1) {
    for (int i = 0; i < m - 1; i++) {
      red.push(red.front());
      red.pop();
    }
    red.pop();
  }
  return red.front();
}
```

Задатак: Максимална бијекција

Филмски продуцент организује вечеру на коју жели да позове глумце. Да би се глумци осећали пријатно на вечери, продуцент жели да обезбеди да је сваки глумац присутан на вечери омиљен глумац неког другог глумца присутног на вечери. Сваки од n глумаца, потенцијалних гостију, одабрао је свог омиљеног глумца из тог скупа глумаца (при чему није искључено и да је неки глумац одабрао сам себе). Напиши програм који одређује највећи подскуп тог скупа глумаца који садржи глумце које продуцент може позвати на вечеру.

Улаз: Са стандардног улаза уноси се број n ($1 \leq n \leq 50000$) који представља број глумаца који су гласали, а затим и редом редни бројеви омиљеног глумца сваког глумца (сви бројеви су између 0 и $n-1$).

Излаз: На стандардни излаз испиши највећи број глумаца који могу присуствовати вечери.

Пример

Улаз	*Излаз*
7	3
2	
0	
0	
4	
4	
3	
5	

Објашњење

На вечеру могу бити позвани глумци са бројевима $0, 2, 4$. Глумац 0 је омиљени глумац глумца 2, глумац 2 је омиљени глумац глумца 0, док је глумац 4 сам себи омиљен.

Решење

Гласови глумаца одређују функцију f дефинисану на скупу $\{0, 1, \ldots, n-1\}$. Нека је скуп S скуп глумаца који су позвани на вечеру. Да би сваки глумац био омиљен глумац неком другом глумцу из скупа S, потребно је да рестрикција функције f на скуп S буде "на" (тј. да за сваку слику постоји оригинал који се слика у ту слику). Пошто је скуп S коначан, на основу Дирихлеовог принципа, она ће уједно бити и "1-1" (за сваку слику ће постојати тачно један оригинал који се у њу слика). Заиста, ако би неки глумац на вечери био омиљен за два различита глумца, неком глумцу на вечери би недостајао глумац коме би он био омиљен. Функија која је истовремено "на" и "1-1" зове се бијекција.

Елиминација елемената

Ефикасан алгоритам можемо направити ако пронађемо потребан услов да елемент буде део скупа S. Наиме, сваки елемент скупа S мора бити слика тачно једног елемента скупа S. Ако је сваки елемент скупа X (домена функције f) слика тачно једног елемента скупа X, тада је f бијекција на скупу X. У супротном мора да постоји елемент који није слика ни једног елемента скупа X и тај елемент не може бити део скупа S. Када тај елемент уклонимо (заједно са његовом сликом), добијамо скуп који је за један елемент мањи и на који можемо применити исти поступак (суштински, имамо описан индуктивни тј. рекурзивни поступак).

Овај приступ се може једноставно имплементирати тако што за сваки елемент скупа X израчунамо број елемената који се сликају у њега. То можемо урадити коришћењем једноставног, асоцијативног низа. Слична техника је показана као у задатку Фреквенције речи.

Можемо одржавати радну листу (ред) елемената у које се не слика ни један елемент (након израчунавања броја елемената који се сликају у сваки од елемената скупа X, све бројеве за које је вредност у асоцијативном низу нула, убацујемо у радну листу). Након тога, све док се радна листа не испразни, узимамо један по један елемент из радне листе, избацујемо га из скупа X и зато смањујемо број елемената који се сликају у слику тог избаченог елемента (умањујемо вредност у асоцијативном низу). Ако се установи да се након тога вредност слике у асоцијативном низу смањила на нулу, тада се слика убацује у радну листу. Иако редослед узимања елемената из радне листе може бити потпуно произвољан, за имплементацију се најчешће користи ред (јер даје некакав осећај правичности). Решење у ком би се уместо реда користио стек било би такође исправно.

```
vector<int> ulazniStepen(n, 0);
for (int i = 0; i < n; i++)
  ulazniStepen[f[i]]++;
queue<int> q;
for (int i = 0; i < n; i++)
  if (ulazniStepen[i] == 0)
    q.push(i);
int broj_elemenata = n;
while (!q.empty()) {
  int i = q.front(); q.pop();
  broj_elemenata--;
  if (--ulazniStepen[f[i]] == 0)
    q.push(f[i]);
}
cout << broj_elemenata << endl;
```

Задатак: Сортирање - сви испред мањи или сви испред већи

Бројеви у низу су такви да за сваки елемент важи или да су сви елементи испред њега мањи од њега или да су сви елементи испред њега већи од њега. Нпр. низ $5, 8, 12, 4, 2, 13, 19, 1$ задовољава то својство. Напиши програм који у линеарној сложености сортира тај низ.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^5$), а затим n елемената низа (елементи су дати у једној линији, раздвојени размацима).

Излаз: На стандардни излаз исписати сортиране елементе низа (раздвојене размаком).

Пример

Улаз	*Излаз*
8 5 8 12 4 2 13 19 1	1 2 4 5 8 12 13 19

Решење

Ако се не узме у обзир специфичност улазних података, низ се може сортирати било којим алгоритмом за сортирање низа бројева (слично као у задатку Сортирање бројева).

Анализа сложености. Ако се користи библиотечка функција сортирања, временска сложеност овог алгоритма је $O(n \log n)$, док је меморијска сложеност $O(n)$.

```
// red sa dva kraja u koji smestamo sortiran niz
deque<int> a;

// ucitavamo prvi element i ubacujemo ga u red
int x; cin >> x;
a.push_back(x);

// ucitavamo ostale elemente
for (int i = 1; i < n; i++) {
  cin >> x;
```

```
  // element poredimo sa poslednjim u redu i dodajemo ga na pocetak
  // ili na kraj reda
  if (a.back() < x)
    a.push_back(x);
  else
    a.push_front(x);
}

// ispisujemo sve elemente iz reda
for (int x : a)
  cout << x << " ";
cout << endl;
```

Решимо задатак индуктивном конструкцијом. Претпоставимо да обрађујемо један по један по један елемент и да смо већ сортирали префикс елемената испред текућег. Ако је он већи од свих елемената испред себе у полазном низу, тада га треба додати на крај сортираног префикса, а ако је мањи од свих елемената испред себе, треба га додати на почетак сортираног префикса.

Пример. Прикажимо сортирање низа `5, 8, 12, 4, 2, 13, 19, 1`.

- На почетку је ред празан, па у њега додајемо 5 (ред постаје `5`).
- 8 је веће од последњег елемента у реду, па га додајемо на крај (ред постаје `5, 8`).
- 12 је веће од последњег елемента у реду, па га додајемо на крај (ред постаје `5, 8, 12`).
- 4 је мање од последњег елемента у реду, па га додајемо на почетак (ред постаје `4, 5, 8, 12`).
- 2 је мање од последњег елемента у реду, па га додајемо на почетак (ред постаје `2, 4, 5, 8, 12`).
- 13 је веће од последњег елемента у реду, па га додајемо на крај (ред постаје `2, 4, 5, 8, 12, 13`).
- 19 је веће од последњег елемента у реду, па га додајемо на крај (ред постаје `2, 4, 5, 8, 12, 13, 19`).
- 1 је мање од последњег елемента у реду, па га додајемо на почетак (ред постаје `1, 2, 4, 5, 8, 12, 13, 19`).

За ефикасну имплементацију потребно је користити структуру података која допушта ефикасно додавање елемената на почетак и на крај и то може бити ред са два краја или двоструко повезана листа. У језику C++ можемо употребити `deque` или `list`. У језику C# можемо употребити `LinkedList`. Да би се проверило да ли је елемент већи од свих испред себе или мањи од свих испред себе, довољно је упоредити га са било којим од тих елемената (на пример, са последњим елементом у реду). Када је ред празан, први елемент можемо убацити било на почетак, било на крај. Да бисмо избегли проверу да ли је ред празан приликом обраде сваког елемента, први елемент можемо убацити у празан ред пре него што почнемо са обрадом осталих елемената.

Анализа сложености. И меморијска и временска сложеност овог алгоритма је $O(n)$.

```
// ucitavamo niz
int n;
cin >> n;
vector<int> a(n);
for (int i = 0; i < n; i++)
  cin >> a[i];

// sortiramo niz
sort(begin(a), end(a));

// ispisujemo sortiran niz
for (int i = 0; i < n; i++)
  cout << a[i] << " ";
cout << endl;
```

4.4 Ред са приоритетом

Ред са приоритетом је врста реда у коме елементи имају на неки начин придружен приоритет, додају се у ред један по један, а увек се из реда уклања онај елемент који има највећи приоритет од свих елемената у реду.

У језику C++ ред са приоритетом се реализује класом `приориту_qyeye<T>`, где је `T` тип елемената у реду. Ред са приоритетом подржава следеће методе:

- `push` - додаје дати елемент у ред
- `pop` - уклања елемент са највећим приоритетом из реда (под претпоставком да ред није празан). Нагласимо да је ова метода типа `void` и да не враћа уклоњени елемент.
- `top` - очитава елемент са највећим приоритетом (под претпоставком да ред није празан)
- `empty` - проверава да ли је ред празан
- `size` - враћа број елемената у реду

Операције `push` и `pop` су сложености $O(\log k)$, где је k број елемената у реду, док су остале операције сложености $O(1)$.

Задатак: Сортирање бројева

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Сортирање уз помоћ реда са приоритетом

Сортирање бројева се може извршити коришћењем реда са приоритетом. Користи се алгоритам *сортирања уз помоћ хипа* (енгл. *heap sort*) који је варијација алгоритма сортирања селекцијом (енгл. *selection sort*) у којем се, подсетимо се, у сваком кораку најмањи елемент доводи на почетак низа. Алгоритам хип сорт користи чињеницу да је одређивање и уклањање најмањег елемента из реда са приоритетом прилично ефикасна операција. Стога се сортирање може реализовати тако што се сви елементи уметну у ред са приоритетом, из кога се затим проналази и уклања један по један најмањи елемент.

Анализа сложености. И убацивање елемената у ред са приоритетом и избацивање елемената из реда са приоритетом обично је сложености $O(\log k)$, где је k број елемената у реду са приоритетом). Стога је укупна сложеност овог алгоритма сортирања $O(n \log n)$.

```
// ovo je način da se u C++-u definiše red sa prioritetom u kome su
// elementi poređani u opadajućem redosledu prioriteta (ovde, vrednosti)
priority_queue<int, vector<int>, greater<int>> Q;
// učitavamo sve elemente niza i ubacujemo ih u red
int n;
cin >> n;
for (int i = 0; i < n; i++) {
  int ai;
  cin >> ai;
  Q.push(ai);
}
// vadimo jedan po jedan element iz reda i ispisujemo ga
while (!Q.empty()) {
  cout << Q.top() << endl;
  Q.pop();
}
```

Види другачија решења овог задатка.

Задатак: Збир k најбољих

Ученик је радио n задатака и за сваки задатак је добио одређени број поена. Одредити збир поена на k задатака које је најбоље урадио.

Улаз: У првој линији стандардног улаза унети природан број n ($1 \leq n \leq 10^6$) - број задатака које је ученик радио, у другој природан број k ($1 \leq k \leq n$) - број задатака које је најбоље урадио, а затим у следећих n линија број поена које је добио на задацима.

Излаз: Укупан број поена које је освојио на k најбоље оцењених задатака.

Пример

Улаз	*Излаз*
10 3 15 80 25 60 10 20 50 45 40 30	190

Решење

Модификовани алгоритам сортирања селекцијом

Једна могућа идеја која избегава сортирање елемената који су мањи од првих k је да се сортирање врши алгоритмом селекције (види задатак Сортирање бројева) који, подсетимо се, у сваком кораку на текуће место у низу доводи најмањи од преосталих елемената низа (у првом кораку се на прво место доводи највећи (или најмањи) елемент целог низа, у другом кораку се на друго место доводи највећи од свих елемената низа осим оног постављеног на прво место итд.). Приметимо да ће се након k корака на почетку низа наћи тачно k највећих елемената и ту се алгоритам може зауставити.

Анализа сложености. Пошто је за налажење најмањег елемента у низу потребно $O(n)$ операција, и то се ради k пута, временска сложеност овог алгоритма је $O(n \cdot k)$, док је меморијска сложеност $O(n)$.

```
// ucitavamo ulazne podatke
int n, k;
cin >> n >> k;
vector<int> a(n);
for (int i = 0; i < n; i++)
  cin >> a[i];

// sortiramo niz primenjujuci k rundi algoritam sortiranja
// selekcijom
for (int i = 0; i < k; i++) {
  int minPoz = i;
  for (int j = i + 1; j < n; j++)
    if (a[j] > a[minPoz])
      minPoz = j;
  swap(a[i], a[minPoz]);
}

// izracunavamo i ispisujemo zbir elemenata niza
int s = 0;
for (int i = 0; i < k; i++)
  s += a[i];
cout << s << endl;
```

Уметање

Једно решење је да се k највећих елемената низа чувају у нерастуће сортираном низу и да се примењује техника сортирања уметањем (види задатак Сортирање бројева). Ради једноставности имплементације чуваћемо низ од $k + 1$ елемената. Када учитамо сваки следећи број поена, стављаћемо га иза последњег постављеног елемента у том низу док се низ не попуни, односно на последњу позицију (ону са индексом k), и затим применом алгоритма уметања померати елемент улево док не дође на своју позицију у том низу (ако је тај елемент није међу највећих k у до сада учитаним елементма, он ће остати на позицији $k + 1$ и у следећем учитавању бити замењен). Уметање можемо реализовати тако што у привремену променљиву учитамо тај последњи

елемент низа, затим све елементе који су мањи од њега померимо за једно место удесно и на крају њега упишемо на место елемента који је последњи померен. На крају, сабирамо елементе на првих k позиција тог низа ($k+1$-ви елемент на позицији k није релевантан).

Анализа сложености. Пошто се у сваком кораку елементи могу померати k пута, а имамо укупно n корака и временска сложеност овог алгоритма је лоша и износи $O(n \cdot k)$, при чему је просторна сложеност боља и износи $O(k)$.

```
int n, k;
cin >> n >> k;

vector<int> a(k + 1);
cin >> a[0];
for (int i = 1; i < n; i++) {
  int x;
  cin >> x;
  // umece x na odgovarajuce mesto u prvih min(i, k) nerastuce
  // uredjenih elemenata tako da je niz i dalje u nerastucem poretku
  int j;
  for (j = min(i, k) - 1; j >= 0 && x > a[j]; j--)
    a[j + 1] = a[j];
  a[j + 1] = x;
}

// izracunavamo i ispisujemo zbir elemenata niza
int s = 0;
for (int i = 0; i < k; i++)
  s += a[i];

cout << s << endl;
```

Ред са приоритетом

Највећих k до сада виђених елемената низа можемо чувати у структури података која нам омогућава да пронађемо најмањи елемент у њој и да га евентуално заменимо оним који је тренутно учитан (ако је тренутно учитани елемент већи од њега). Идеална структура за то је хип тј. ред са приоритетом.

Ред са приоритетом у језику C++ можемо добити помоћу `priority_queue`. Елементе у ред можемо убацити методом `push`. Елемент који је најмањи можемо очитати методом `top` и избацити методом `pop`.

На почетку ред попуњавамо са k првих учитаних елемената, а затим сваки наредни учитани елемент поредимо са најмањим у реду и ако је већи од њега, најмањи избацујемо, а учитани елемент убацујемо.

Анализа сложености. Пошто је сложеност метода за убацивање и избацивање из реда са приоритетом логаритамска, а методе за очитавање најмањег елемента константна, временска сложеност овог алгоритма је $O(n \cdot \log(k))$, док је просторна сложеност $O(k)$.

Напомена. Приметимо да је овај алгоритам донекле сличан алгоритму сортирања уз помоћ хипа тј. алгоритма Хип-сорт (HeapSort).

```
int n, k;
cin >> n >> k;

// red sa prioritetom koji cuva k najvecih elemenata koristi se
// min-hip, koji omogucava brzo uklanjanje najmanjeg elementa
priority_queue<int, vector<int>, greater<int>> pq;

// ucitavamo prvih k elemenata i ubacujemo ih u red
for (int i = 0; i < k; i++) {
  int x;
  cin >> x;
  pq.push(x);
}
```

```
// ucitavamo preostale elemente
for (int i = k; i < n; i++) {
  int x;
  cin >> x;
  // ako je ucitani element veci od najmanjeg trenutno u redu
  // izbacujemo taj najmanji i menjamo ga ucitanim
  if (x > pq.top()) {
    pq.pop();
    pq.push(x);
  }
}

// izbacujemo elemente iz reda racunajuci njihov zbir i ispisujemo ga
int s = 0;
while (!pq.empty()) {
  s += pq.top();
  pq.pop();
}
cout << s << endl;
```

Види другачија решења овог задатка.

Задатак: К-ти највећи збир пара елемената два низа

Дата су два низа која садрже природне бројеве. Напиши програм који одређује k-ти највећи збир који се може добити када се сабере један елемент првог и један елемент другог низа.

Улаз: Са стандардног улаза се учитава број m ($1 \leq m \leq 5000$), а затим из наредног реда m елемената првог низа раздвојених размаком. Из наредног реда се учитава број n ($1 \leq n \leq 5000$), а затим из наредног реда n елемената другог низа раздвојених размаком. Елементи оба низа су природни бројеви између 0 и 10^6. На крају се учитава број k ($0 \leq k < mn$).

Излаз: На стандардни излаз збир који се налази на позицији k у низу који би се добио када би се низ свих збирова парова једног елемента првог и једног елемента другог низа сортирао нерастуће (позиције се броје од нуле).

Пример 1

Улаз	*Излаз*
3 1 5 3 3 6 4 2 4	7

Објашњење

Збирови који се могу добити, поређани од највећег ка најмањег су $5+6=11$, $5+4=9$, $3+6=9$, $5+2=7$, $3+4=7$, $1+6=7$, $3+2=5$, $1+4=5$, $1+2=3$, па се на позицији 4 налази збир 7.

Пример 2

Улаз

```
5
5 3 8 6 1
6
1 10 9 7 12 2
9
```

Излаз

```
15
```

Решење

Груба сила

Решење грубом силом подразумева да се формира низ свих збирова, да се сортира нерастуће и да се прочита елемент са позиције k.

Анализа сложености. Парова елемената има $m \cdot n$, па је меморијска сложеност квадратна и износи $O(mn)$. Временском сложеношћу доминира сортирање и она износи $O(mn \log (mn))$.

Уместо сортирања, могуће је применити и мало ефикаснији алгоритам *QuickSelect*, који проналази елемент на позицији k и без сортирања низа, но тиме решење не би значајно било унапређено. Опис тог алгоритма дат је у задатку Збир k најбољих.

```
// formiramo sve zbirove parova elemenata dva niza
vector<int> zbirovi;
zbirovi.reserve(m*n);
for (int i = 0; i < m; i++)
  for (int j = 0; j < n; j++)
    zbirovi.push_back(a[i] + b[j]);

// sortiramo niz zbirova nerastuce
sort(begin(zbirovi), end(zbirovi), greater<int>());

// ispisujemo k-ti element sortranog niza zbirova
cout << zbirovi[k] << endl;
```

Обједињавање сортираних низова

Проблем се може свести на проблем обједињавања неколико сортираних низова, који се не морају истовремено чувати у меморији. Наиме, ако сортирамо оба низа опадајуће, можемо разматрати следеће низове:

$$\begin{gathered} a_0 + b_0, a_0 + b_1, \ldots, a_0 + b_{n-1}, \\ a_1 + b_0, a_1 + b_1, \ldots, a_1 + b_{n-1}, \\ \ldots \\ a_{m-1} + b_0, a_{m-1} + b_1, \ldots, a_{m-1} + b_{n-1}. \end{gathered}$$

Сви они су сортирани нерастуће и могу се објединити коришћењем реда са приоритетом. У почетку у ред убацујемо први елемент сваке листе тј. све збирове облика $a_i + b_0$. У сваком кораку избацујемо највећи збир из реда и додајемо у ред наредни елемент листе којој он припада. Да бисмо након вађења елемента из реда могли додати наредни елемент листе којој он припада, поред вредности збира $a_i + a_j$ (на основу којих је ред уређен) у реду морамо чувати и индексе i и j. Заправо довољно је чувати само индекс j, јер се на основу збира $z = a_i + b_j$, збир $a_i + b_{j+1}$ може добити као $z - b_j + b_{j+1}$.

Анализа сложености. У реду се у сваком тренутку налази највише m елемената. Пошто се чувају и оригинални низови (да би се сортирали), меморијска сложеност је $O(m + n)$. Ажурирање реда врши се k пута. Пошто је сложеност иницијалних сортирања низова \$, сложеност једног ажурирања реда је $O(\log m)$, а важи $k < mn$, временска сложеност је $O(m \log m + n \log n + mn \log m)$. Сложеношћу јасно доминира фаза обједињавања, па се временска сложеност може проценити и само са $O(mn \log m)$.

```
// sortiramo nizove opadajuce
sort(begin(a), end(a), greater<int>());
sort(begin(b), end(b), greater<int>());

// objedinjavamo sortirane nizove
// a[i] + b[0], ..., a[i] + b[n-1], za svako i od 0 do m-1
// u redu cuvamo zbirove a[i] + b[j] i pozicije j
priority_queue<pair<int, int>> pq;

// dodajemo u red prvi element svakog niza
for (int i = 0; i < m; i++)
  pq.emplace(a[i] + b[0], 0);
```

```
// k puta azuriramo red
for (int K = 0; K < k; K++) {
  // skidamo element sa vrha reda
  int j, z;
  tie(z, j) = pq.top(); pq.pop();
  // ako lista u kojoj je bio skinuti elemnt nije ispraznjena,
  // u red dodajemo njen naredni element
  if (j + 1 < n)
    pq.emplace(z - b[j] + b[j+1], j+1);
}

// element na poziciji k je trenutno na pocetku reda
cout << pq.top().first << endl;
```

Задатак: Ажурирање медијане

У заводу за статистику желе да што непристрасније процене која је просечна плата. Закључили су да израчунавање аритметичке средине може дати мало искривљену слику јер неколико људи са веома високим платама могу значајно повећати просек. Зато су одлучили да уместо аритметичке средине израчунају медијану која се добија тако што се све плате поређају у неопадајући низ и онда се узме средишњи елемент тог низа. Ако у низу има паран број елемената, онда се за медијану узима аритметичка средина два средишња елемента. На пример, ако медијана низа бројева 1, 2, 4, 7, 9 је 4 (јер је он средишњи), а низа бројева 1, 2, 4, 5, 7, 9 је 4.5 (јер је то аритметичка средина бројева 4 и 5 који су средишњи елементи). Подаци о платама пристижу у завод, а софтвер мора да може да у сваком тренутку да податак о медијани до тада унетих плата.

Улаз: Са стандардног улаза се уносе линије, све до краја стандардног улаза. Линија или садржи слово `d` и затим износ плате раздвојен размаком (цео број), што значи да се уноси податак о новој плати или садржи слово `m` што значи да је потребно на стандардни излаз исписати податак о медијани до тада унетих плата. Прва линија сигурно садржи `d`.

Излаз: На стандардном излазу су исписане тражене медијане, свака у посебном реду, заокружене на једну децималу.

Пример

Улаз	*Излаз*
`d 5`	`6.0`
`d 7`	`6.5`
`d 6`	
`m`	
`d 8`	
`m`	

Решење

Два хипа

Ефикасно решење се може добити ако у сваком тренутку у једној (рећи ћемо левој) колекцији чувамо све елементе који су мањи од средишњег, а у другој (рећи ћемо десној) све оне који су већи или једнаки средишњем (ако постоји паран број елемената, те две колекције треба да садрже исти број елемената, а ако постоји непаран број елемената, десна колекција може да садржи један елемент више). Ако има непаран број елемената, тада је медијана једнака најмањем елементу десне колекције, а у супротном је једнака аритметичкој средини између највећег елемента леве и најмањег елемента десне колекције. Сваки нови елемент се пореди са најмањим елементом десне колекције и ако је мањи или једнак њему убацује се у леву колекцију, а ако је већи од њега, убацује се у десну колекцију. Тада се проверава да ли се средина променила. Ако се десило да лева колекција има више елемената од десне (што не допуштамо), највећи елемент леве колекције треба да пребацимо у десну. Ако се десило да у десној колекцији има два елемента више него у левој, тада најмањи елемент десне колекције пребацујемо у леву. Дакле, лева колекција треба да буде таква да лако можемо да пронађемо и избацимо њен највећи елемент, а десна да буде таква да лако можемо да пронађемо и избацимо њен најмањи елемент, при чему обе колекције морају да подрже ефикасно убацивање произвољних елемената. Јасно је да те колекције треба да буду хипови (у језику C++ можемо употребити `priority_queue`) у

којима се најмања тј. највећа вредност може очитати у константном времену, уклонити у логаритамском, исто колико је потребно и да се уметне нови елемент.

Напомена. Идеја да се уместо у једне подаци чувају у две структуре података често може довести до ефикасног решења. На пример, уместо повезане листе у коју се елементи убацију на тренутну позицију итератора који се помера по листи, могу се користити два стека, као што је приказано у задатку Линијски едитор.

```
priority_queue<int, vector<int>, greater<int>> veci_od_sredine;
priority_queue<int, vector<int>, less<int>> manji_od_sredine;

double medijana() {
  if (manji_od_sredine.size() == veci_od_sredine.size())
    return (manji_od_sredine.top() + veci_od_sredine.top()) / 2.0;
  else
    return veci_od_sredine.top();
}

void dodaj(int x) {
  if (veci_od_sredine.empty())
    veci_od_sredine.push(x);
  else {
    if (x <= veci_od_sredine.top())
      manji_od_sredine.push(x);
    else
      veci_od_sredine.push(x);
    if (manji_od_sredine.size() > veci_od_sredine.size()) {
      veci_od_sredine.push(manji_od_sredine.top());
      manji_od_sredine.pop();
    } else if (veci_od_sredine.size() > manji_od_sredine.size() + 1) {
      manji_od_sredine.push(veci_od_sredine.top());
      veci_od_sredine.pop();
    }
  }
}
```

4.5 Апстрактне структуре података

Структуре података карактерише њихов апстрактни интерфејс тј. операције које треба да подрже (и допуштена сложеност тих операција). У наредних неколико примера биће приказано како се могу реализовати неке структуре на основу задатог интерфејса. У многим случајевима довољно је на инвентиван начин употребити неку библиотечку структуру података или искомбиновати неколико библиотечких структура података.

Задатак: Мапа са увећањем

Мапа тј. речник је структура података која пресликава задате целобројне кључеве у задате целобројне вредности. Допуштени су следеће операције над мапом.

- `r k` — *read* — испис вредности придружене кључу на стандардни излаз (ако том кључу није додељена вредност, потребно је исписати 0).
- `w k x` — *write* — кључу `k` придружује се вредност `x` (ако је раније била придружена нека друга вредност, она се занемарује).
- `e k` — *erase* — брише се вредност придружена кључу `k`.
- `i k x` — *increment* — вредност придружена кључу `k` се увећава за `x` (ако кључу `k` раније није била придружена вредност, њему се додељује вредност `x`).
- `a x` — *increment all* — све вредности које су придружене кључевима се увећавају за `x`.

Написати програм који чита низ овако описаних операција и спроводи их.

Улаз: Са стандардног улаза се учитавају операције, свака у посебном реду, њих највише 10^5.

Излаз: На стандардни излаз исписати резултат извршавања учитаних операција (оно што се исписује операцијама `r k`).

Пример

Улаз	*Излаз*
`w 0 2`	`2`
`w 1 1`	`3`
`r 0`	`5`
`i 1 2`	`6`
`r 1`	`0`
`a 3`	
`w 2 0`	
`r 0`	
`r 1`	
`r 2`	

Решење

Груба сила

Основу решења представља коришће библиотечког асоцијативног низа (мапе тј. речника). Једина операција која није директно подржана мапама је увећање свих елемената за дату вредност. Ако се она имплементира грубом силом, потребан је обилазак свих кључева у мапи, што је прилично неефикасно.

Анализа сложености. Сложеност операције увећања свих вредности је линеарна у односу на број кључева у мапи, па је укупна сложеност квадратна у односу на број операција (најгори случај може бити када прва половина наредби упише све различите кључеве у мапу, а друга половина наредби буде увећање свих вредности).

```
map<int, int> m;

int vrednost(int k) {
   auto it = m.find(k);
   return it == m.end() ? 0 : it->second;
}

void umetni(int k, int v) {
   m[k] = v;
}

void obrisi(int k) {
   m.erase(k);
}

void uvecaj(int k, int v) {
   m[k] += v;
}

void uvecajSve(int v) {
  for (auto& p : m)
    p.second += v;
}

int main() {
  ios_base::sync_with_stdio(0); cin.tie(0);
  char c;
  while (cin >> c) {
    if (c == 'r') {
      int k;
      cin >> k >> ws;
      cout << vrednost(k) << '\n';
```

```
    } else if (c == 'w') {
      int k, x;
      cin >> k >> x >> ws;
      umetni(k, x);
    } else if (c == 'e') {
      int k;
      cin >> k >> ws;
      obrisi(k);
    } else if (c == 'i') {
      int k, x;
      cin >> k >> x >> ws;
      uvecaj(k, x);
    } else if (c == 'a') {
      int x;
      cin >> x >> ws;
      uvecajSve(x);
    }
  }
  return 0;
}
```

Чување збира свих увећања

Да би се та операција могла ефикасније имплементирати уз мапу чувамо и засебан цео број који представља збир свих увећања. Уместо да повећавамо све вредности у мапи, повећаваћемо само тај број. Пре исписа вредности придружене кључу увећа ћемо је за тај број, а при придруживању вредности кључу умањићемо је за тај број.

Анализа сложености. У зависности од тога да ли се користи уређени или неуређени асоцијативни низ, сложеност свих операција је или $O(\log n)$ или амортизована константна. Укупна сложеност је зато у најгорем случају квазилинеарна у односу на број наредби.

```
map<int, int> m;
int uvecanje = 0;

int vrednost(int k) {
   auto it = m.find(k);
   return it == m.end() ? 0 : it->second + uvecanje;
}

void umetni(int k, int v) {
   m[k] = v - uvecanje;
}

void obrisi(int k) {
   m.erase(k);
}

void uvecaj(int k, int v) {
   auto it = m.find(k);
   if (it == m.end())
     m[k] = v - uvecanje;
   else
     it->second += v;
}

void uvecajSve(int v) {
   uvecanje += v;
}
```

Задатак: Ред без дупликата

Програм извршава следеће операције над одређеном колекцијом података.

- `i x` — *insert* — ако колекција већ не садржи елемент `x` он се у њу додаје.
- `r` — *remove* — из колекције се уклања и на стандардни излаз се исписује први елемент који је у њу додат (као код класичног реда). Ако је колекција празна, исписује се `-`.

Напиши програм који учитава низ наредби и извршава их.

Улаз: Са стандардног улаза се учитава низ наредби, свака је у посебном реду.

Излаз: На стандардни излаз исписати резултат извршавања програма.

Пример

Улаз	*Излаз*
`i 1`	`1`
`i 2`	`2`
`i 1`	`3`
`i 3`	`4`
`r`	`-`
`r`	
`i 4`	
`r`	
`r`	
`r`	

Решење

Груба сила - линеарна претрага

У имплементацији можемо употребити библиотечку имплементацију структуре ред. Пре додавања елемента на крај, морамо проверити да ли елемент већ постоји у реду. У језику C++ можемо употребити структуру података `deque` и функцију `find` која примењује линеарну претрагу (њу није могуће употребити када се користи `queue`).

Анализа сложености. Због линеарне претраге реда пре додавања новог елемента, сложеност додавања је линеарна у односу на тренутни број елемената у реду, па је укупна сложеност алгоритма квадратна (најгори случај може бити када се у свим наредбама уноси нови елемент у ред, различит од свих претходних).

```
ios_base::sync_with_stdio(false); cin.tie(0);
deque<int> q;

char c;
while (cin >> c) {
  if (c == 'i') {
    int x;
    cin >> x >> ws;
    if (find(q.begin(), q.end(), x) == q.end()) {
      q.push_back(x);
    }
  } else if (c == 'r') {
    if (!q.empty()) {
      cout << q.front() << '\n';
      q.pop_front();
    } else {
      cout << "-" << '\n';
    }
  }
}
```

Чување додатног скупа

Ефикасна имплементација је могућа ако се уз ред чува додатни скуп елемената који су садржани у реду. Пре додавања елемента у ред, проверава се да ли тај елемент постоји у скупу. Ако не постоји, додаје се и у скуп и у ред. Приликом избацивања елемената, први елемент реда се избацује и из реда и из скупа.

Анализа сложености. У зависности од тога да ли се користи уређен или неуређен скуп, сложеност операција додавања, брисања и претраге елемената у скупу може бити логаритамска или амортизована константнтна. Додавање у ред и читање елемената из реда има константну сложеност. Зато је укупна сложеност алгоритма линеарна ако се користи неуређен скуп тј. квазилинеарна ако се користи уређен скуп.

```
  return 0;
}
```

Задатак: Фреквенцијски стек

Програм извршава две врсте операција са структуром података која донекле подсећа на стек.

- Операција `0 x` поставља елемент `x` на врх стека.
- Операција `1` уклања елемент који се најчешће појављује од свих елемената који су тренутно на стеку. Ако је више таквих елемената, уклања се онај који је последњи додат. Нпр. ако су на стеку елементи `1 2 2 1 3`, уклања се елемент `1` и стек постаје `1 2 2 3`.

Улаз: Са стандардног улаза се учитава број операција n ($1 \leq n \leq 10^5$), а затим n операција (свака у посебном реду).

Излаз: Опис излазних података.

Пример

Улаз	*Излаз*
12	2
0 1	1
0 2	2
0 2	3
0 1	2
0 3	1
0 2	
1	
1	
1	
1	
1	
1	

Решење

Основна идеја решења је да се уместо јединственог стека елементи чувају на више стекова организованих по фреквенцијама тј. броју појављивања елемента. Прво појављивање неког елемента стављаћемо на стек првих појављивања, друго појављивање на стек других појављивања итд. Да бисмо знали на који стек треба ставити наредно појављивање неког елемента, помоћу асоцијативног низа (речника, мапе) одржаваћемо број појављивања сваког елемента. Одржаваћемо и максималну фреквенцију. Ако се додавањем неког елемента повећава максимална фреквенција, додајемо нови стек. Приликом уклањања елемента, уклањаћемо врх последњег стека, тј. стека који одговара тој највећој фреквенцији. Ако се након тога тај стек испразни, смањујемо максималну фреквенцију.

Пример. Прикажимо рад алгоритма на примеру додавања бројева `1 2 2 1 3 2`.

- Прво се додаје број `1` на стек елемената који се појављују само једном. Максимална фреквенција постаје 1.

  `1: 1`

- Након тога се додаје број `2` на стек елемената који се појављују само једном.

  `1: 1, 2`

- Затим се додаје број 2 на стек елемената који се појављују два пута. Максимална фреквенција постаје 2.

```
1: 1, 2
2: 2
```

- Затим се додаје број 1 на стек елемената који се појављују два пута.

```
1: 1, 2
2: 2, 1
```

- Затим се додаје број 3 на стек елемената који се појављују једном.

```
1: 1, 2, 3
2: 2, 1
```

- На крају се додаје број 2 на стек елемената који се појављују три пута. Максимална фреквенција постаје 3.

```
1: 1, 2, 3
2: 2, 1
3: 2
```

Прикажимо сада уклањање елемената док се стек не испразни.

- Пошто је максимална фреквенција 3, уклања се врх стека елемената са фреквенцијом 3 и исписује се број 2. Пошто се тај стек испразнио максимална фреквенција се смањује на 2.

```
1: 1, 2, 3
2: 2, 1
```

- Пошто је максимална фреквенција 2, уклања се врх стека елемената са фреквенцијом 2 и исписује се број 1.

```
1: 1, 2, 3
2: 2
```

- Пошто је максимална фреквенција 2, уклања се врх стека елемената са фреквенцијом 2 и исписује се број 2. Пошто се тај стек испразнио максимална фреквенција се смањује на 1.

```
1: 1, 2, 3
```

- Након тога се редом скидају и исписује елементи 3, 2 и на крају 1 (тада се максимална фреквенција смањује на нулу).

```
ios_base::sync_with_stdio(false); cin.tie(0);
// frekvencija tj. broj pojavljivanja svakog elementa
map<int, int> frekv;
// niz stekova - na steku broj i čuvaju se i-ta pojavljivanja svih
// elemenata u originalnom steku
map<int, stack<int>> stekovi;
// najveca frekvencija nekog elementa u originalnom steku
int maksFrekv = 0;

// obradjujemo n naredbi
int n;
cin >> n;
for (int i = 0; i < n; i++) {
  int naredba;
  cin >> naredba;
  if (naredba == 0) {
    // ucitavamo element
    int x;
    cin >> x;
    // uvecavamo njegov broj pojavljivanja
    frekv[x]++;
```

```
    // u skladu sa tim ga postavljamo na njemu odgovarajuci stek
    stekovi[frekv[x]].push(x);
    // azuriramo najvecu frekvenciju nekog elementa
    maksFrekv = max(maksFrekv, frekv[x]);
  } else if (naredba == 1) {
    // uklanjamo element sa vrha steka elemenata koji se pojavljuju
    // najcesce
    int x = stekovi[maksFrekv].top();
    stekovi[maksFrekv].pop();
    cout << x << '\n';
    // smanjujemo njegov broj pojavljivanja
    frekv[x]--;
    // azuriramo najvecu frekvenciju nekog elementa
    if (stekovi[maksFrekv].empty())
      maksFrekv--;
  }
}
```

4.6 Имплементација структура података

Као што смо већ видели, језик C++ пружа подршку за велики број структура података, било примитивних, подржаних кроз сам језик (на пример, кориснички дефинисане структуре и уније и статички и динамички алоцирани низови), било подржаних кроз стандардну библиотеку (STL).

Подсетимо се, по начину како су имплементиране, библиотечке структуре података се могу груписати на следећи начин.

- У групу секвенцијалних структура података (контејнера) спадају `array`, `vector`, `string`, `list`, `forward_list` и `deque`.
- У групу адаптора контејнера спадају `stack`, `queue` и `priority_queue`.
- У групу асоцијативних контејнера спадају `set`, `multiset`, `map` и `multimap`.
- У групу неуређених асоцијативних контејнера спадају `unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap`.

Секвенцијални контејнери се користе за складиштење серија елемената и представљају одређена уопштења класичних низова (додајући могућности динамичког проширивања, додавања и уклањања елемената и слично). Сваки секвенцијални контејнер има своју специфичну имплементацију која одређује сложеност операција и самим тим адекватне сценарије употребе.

- `array<T, size>` представља само танак **омотач око класичног статичког низа** чија је димензија `size` позната у тренутку декларације (не може се употребити променљива) и служи само да омогући да се класични статички низови могу користити на исти начин као и други контејнери (на пример, допуштена је директна додела низа низу, поређење два низа оператором `==` и слично, што није могуће ако се користе примитивни статички низови).
- `vector<T>` је имплементиран преко **динамичког низа** који се по потреби реалоцира. Таква имплементација омогућава ефикасан индексни приступ и итерацију у оба смера, као и додавање елемената на крај (`push_back`) и скидање елемената са краја (`pop_back`). Пошто услед реалокација динамичког низа долази до честог копирања великог броја елемената, вектори који се током рада програма проширују, показују лоше перформансе када се у њима чувају велики објекти. `string` је имплементиран слично као вектор (уз додатне оптимизације и операције специфичне за податке текстуалног типа).
- `list<T>` је имплементиран преко **двоструко повезане листе**, док је `forward_list<T>` имплементиран преко **једноструко повезане листе**. Захваљујући таквој имплементацији, основна предност ових контејнера у односу на `vector` и `deque` је то што подржавају уметање и брисање елемената са било које позиције (не само на почетак или на крај) у константној сложености. Са друге стране, губи се могућност ефикасног индексног приступа. Двоструко имплементирана листа омогућава итерацију у оба смера, а једноструко повезана листа само унапред (над итераторима није могуће примењивати оператор `--` нити