```
    // 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

одузимање бројевних вредности од итератора). Пошто за разлику од вектора не долази до реалокације, у листама се могу складиштити и велики објекти.

- `deque<T>` је имплементиран као специфична структура података коју ћемо називати **дек** и која се може замислити као вектор у коме се налазе показивачи на блокове (низове) елемената фиксне величине (нешто попут `vector<array<T, size>*>`). Оваква специфична имплементација омогућава ефикасно додавање и уклањање елемената са оба краја (отуда и назив `deque` тј. double ended queue), али и ефикасну итерацију кроз све елементе, у оба смера, као и ефикасан индексни приступ. Једна од основних предности у односу на `vector` је то што се због чувања показивача приликом реалокације не копирају елементи већ само показивачи на њих, тако да се у дек који се током рада програма проширује могу слободно чувати и велики објекти.

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 коришћењем вектора за складиштење података. Тип `stack<int>` подразумева заправо `stack<int, vector<int>>`, а могуће је користити и, на пример, `stack<int, forward_list<int>>` у ком се за имплементацију стека користи једноструко повезана листа.

- `queue` имплементира функције реда коришћењем дека за складиштење података. На пример, тип `queue<int>` подразумева заправо `queue<int, deque<int>>`. Уместо дека могуће је употребити и неки други контејнер (пре свега `list<int>`, док би `vector<int>` и `forward_list<int>` имали веома лошу сложеност, па их нема смисла користити).

- `priority_queue` имплементира функције реда са приоритетом коришћењем вектора за складиштење података. Тип `priority_queue<int>` заправо представља `priority_queue<int, vector<int>, less<int>>` где је `less<int>` функција која се користи за поређење елемената и прозрокује уређеност по опадајућем редоследу приоритета (на врху је елемент са највећим приоритетом). У вектору су смештени елементи специјалног дрвета који се назива *хип* и које ће бити објашњено касније.

Уређени асоцијативни контејнери су имплементирани помоћу **самобалансирајућих уређених бинарних дрвета** (обично су то **црвено-црна дрвета, RBT – red-black tree**).

- `set<T>` је имплементиран помоћу уређеног бинарног дрвета у коме се у чворовима налазе елементи скупа и у ком су у свим чворовима различите вредности.

- `multiset<T>` може бити имплементиран помоћу уређеног бинарног дрвета у коме се у чворовима налазе елементи мултискупа и у ком је могуће да постоји више чворова у коме су различите вредности или помоћу уређеног бинарног дрвета у чијим се чворовима налазе елементи мултискупа уз њихов број појављивања, без понављања елемената скупа.

- `map<K, V>` је имплементиран помоћу уређеног бинарног дрвета на основу кључева у коме се у чворовима налазе подаци о кључевима и њима придруженим вредностима и у ком су у свим чворовима различите вредности кључева.

- `multimap<K, V>` може бити имплементиран помоћу помоћу уређеног бинарног дрвета на основу кључева у коме се у чворовима налазе подаци о кључевима и њима придруженим вредностима и у ком више чворова може имати исту вредност кључа.

Неуређени асоцијативни контејнери су имплементирани помоћу **хеш-табела**. То су

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

4.6.1 Динамички низ - имплементација вектора

Вектори се имплементирају у облику динамичког низа, којем се димензија мења и који се реалоцира кад год број елемената достигне величину алоцираног простора.

Задатак: Вектор

Програм извршава три врсте наредби над низом целих бројева:

- `w p x` – *write* – у низ се на позицију `p` уписује вредност `x`.
- `r p` – *read* – на стандардни излаз се исписује вредност прочитана са позиције `p` (сматра се да се на позицијама на којима није вршен упис налазе нуле).
- `p x` – *push* – на крај низа, на позицију непосредно иза последњег уписаног елемента уписује се вредност `x`.

Написати програм који учивата ове наредбе, извршава их једну по једну и исписује резултат њиховог извршавања (водећи при том рачуна о томе да се наредбе изврше ефикасно).

Улаз: Свака линија стандардног улаза садржи једну наредбу.

Излаз: На стандардни излаз исписати резултат извршавања свих учитаних наредби.

Пример

Улаз	*Излаз*
`p 1`	`1`
`w 2 2`	`2`
`r 0`	`0`
`r 2`	`5`
`r 1`	`2`
`w 0 5`	
`r 0`	
`w 1 4`	
`w 1 2`	
`r 1`	

Решење

Библиотечка имплементација

Задатак се веома једноставно решава коришћењем библиотечке колекције `vector`.

```
ios_base::sync_with_stdio(false);
cin.tie(0);
vector<int> a;
char c;
while (cin >> c) {
  if (c == 'w') {
    int i, x;
    cin >> i >> x >> ws;
    if (i >= a.size())
      a.resize(i+1);
    a[i] = x;
  } else if (c == 'r') {
    int i;
    cin >> i >> ws;
    cout << (i < a.size() ? a[i] : 0) << '\n';
  } else if (c == 'p') {
    int x;
    cin >> x >> ws;
    a.push_back(x);
  }
}
```

Ручна имплементација

Прикажимо како се динамички низ може имплементирати када се не користи библиотечка подршка.

Основна идеја динамичког низа је то да се у старту алоцира неки претпостављени број елемената и да се, када се установи да тај број елемената није више довољан, изврши реалокација низа, тако што се алоцира нови, већи низ, затим се у тај нови низ ископирају елементи оригиналног низа и на крају се тај стари низ обрише, а

показивач преусмери ка новом низу (у језику C++ не постоји оператор који би био еквивалентан C функцији `realloc`).

Имплементација основне функционалности динамичког низа у језику C++ могла би да изгледа како је приказано у коду. Једноставности ради користимо глобалне промењиве и функције – оне се једноставно избегавају, на пример, тако што може да се користи објектнооријентисано програмирање, али то није у фокусу овог курса. Улогу показивача `NULL` из језика C има показивач `nullptr`, улогу функције `malloc` има израз `new`, а улогу функције `free` има израз `delete` (постоје озбиљне разлике између ових C функција и C++ оператора, али ни то нам у овом тренутку није у фокусу). Израз `new T[n]` користимо да бисмо алоцирали низ од `n` елемената типа `T`, при чему број елемената тог типа наводимо у угластим заградама. За разлику од `malloc` која враћа `void*`, изразом `new T[n]` добијамо показивач типа `T*`. Ослобађање низа елемената алоцираног са `new T[n]` вршимо оператором `delete[]`.

Анализа сложености. Ако се динамичка реалокација врши на основу неке геометријске стратегије за повећање величине (нпр. сваки пут повећамо број елемената дупло), додавање на крај се врши у амортизованом константном времену. Индексни приступ елементу (приступ на основу позиције) захтева константно време (исто као код обичног низа).

```
// adresa početka niza
int* a = nullptr;
// broj alociranih i broj popunjenih elemenata
int alocirano = 0, n = 0;

// čitanje vrednosti elementa na datoj poziciji
// ne proverava se pripadnost opsegu niza
int procitajElement(int i) {
  return a[i];
}

// upis vrednosti elementa na datu poziciju
// ne proverava se pripadnost opsegu niza
void postaviElement(int i, int x) {
  a[i] = x;
}

// pomoćna funkcija koja vrši realokaciju niza na dati broj elemenata
void realociraj(int m) {
  // alociramo novi niz
  int* novo_a = new int[m];
  alocirano = m;
  // ako postoji stari niz, kopiramo njegove elemente u novi
  // i brišemo stari niz
  if (a != nullptr) {
    copy_n(a, min(m, n), novo_a);
    delete[] a;
  }
  // pokazivač usmeravamo ka novom nizu
  a = novo_a;
}

void postaviVelicinu(int nn) {
  if (alocirano <= nn)
    realociraj(2 * nn + 1);
  n = nn;
}

// dodavanje datog elementa na kraj dinamičkog niza
// niz se automatski realocira ako je potrebno
void dodajNaKraj(int x) {
  if (alocirano <= n)
```

```
    realociraj(2 * alocirano + 1);
  a[n++] = x;
}

// brisanje celog niza
void obrisi() {
  delete[] a;
}
```

4.6.2 Листе

Структуре листе подразумевају да се подаци чувају у меморији у чворовима којима се поред података чувају показивачи. У зависности од тога да ли се чувају само показивачи на следећи или и на претходни елемент разликују се:

- једноструко повезане листе
- двоструко повезане листе

Под претпоставком да су познати показивачи на почетак и на крај листе, једноструко повезане листе допуштају додавање на почетак и на крај, као и брисање са почетка у времену $O(1)$, док брисање елемента са краја захтева време $O(n)$ (јер је потребно изменити претпоследњи чвор, до кога се може доћи само поновним обиласком ниске од почетка). Уметање и брисање следећег елемента у односу на елемент на који указује познати показивач се може извршити у времену $O(1)$. Ипак, проналажење позиције на коју треба уметнути неки елемент често захтева пролаз кроз листу и захтева време $O(n)$. Уметање елемента тако да претходи елементу на који указује познати показивач, као и брисање елемента на који указује познати показивач захтева проналазак и измену претходног чвора, што захтева нови пролазак кроз листу и време $O(n)$.

Приступ елементу на датој позицији (индексни приступ) захтева време $O(n)$.

Под претпоставком да су познати показивачи на почетак и на крај листе, двоструко повезане листе допуштају и додавање и брисање и са почетка и са краја листе у времену $O(1)$. Уметање и брисање било испред, било иза елемента на који указује познати показивач врши се у времену $O(1)$. Остале операције се извршавају у истом времену као и код једноструко повезаних листа. Мане двоструко повезаних у односу на једноструко повезане листе су то што због чувања показивача на претходне елементе захтевају више меморије и појединачне операције могу бити мало спорије јер се захтева ажурирање више показивача. Предности су то што омогућују ефикасније извршавање неких операција (пре свега брисања са краја и итерација уназад).

Имплементација се често може учинити једноставнијом (па и донекле ефикаснијом) ако се уведе вештачки чвор (“сентинела”) којим се обезбеђује да је листа увек непразна и да показивач који се налази испред стварног почетка или иза стварног краја нема вредност NULL, већ указује на неки чвор листе (у овом случају, на сентинелу), тако да се његов претходни и следећи елемент могу добити на исти начин као и када показивач указује на неки регуларан чвор листе.

Употреба листа је све ређа и ређа на савременим системима. Алокација и деалокација појединачних чворова листе је скупа операција и у реалним имплементацијама потребно је користити неки специфичан механизам управљања меморијом (на пример, већи број допустивих чворова алоцирати одједном, а онда у листу уланчавати један по један од тих чворова). Додатни проблем је то што су чворови често раштркани по меморији, па су промашаји кеш меморије много чешћи него у случају рада са структурама у којима су подаци у меморији смештени повезано (низовима и структурама у којима се користе већи блокови повезаних бајтова). Такође, због чувања показивача листе захтевају много више меморије него низови.

Предности листа у односу на низове и декове наступају пре свега у ситуацијама у којима се у листама чувају велики подаци. Тада се током реалокације динамичких низова копирају велике количине података, што може бити неефикасно, па је корисније употребити листе код којих се подаци не реалоцирају.

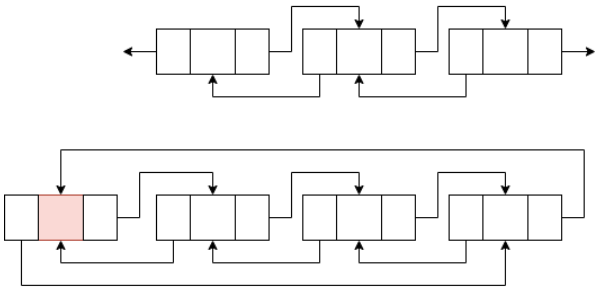
Задатак: Линијски едитор

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

Задатак се може решити тако што се ручно имплементира двоструко повезана листа (пошто се курсор може кретати у оба смера, јасно је да листа мора бити двоструко повезана). Сваки чвор двоструко-повезане листе чува вредност (у нашем случају карактер), показивач на претходни и показивач на следећи чвор. Имплементација такве листе је једноставнија ако се уведе специјални чвор (понекад се назива *сентинела*) који истовремено чува позицију почетног и крајњег чвора двоструко повезане листе (његов показивач на следећи чвор указује на стварни почетак, а показивач на претходни чвор указује на стварни крај листе, при чему оба показују на сентинелу ако је листа празна). Вредност уписана у сентинелу се никада не користи.



Слика 4.1: Двоструко повезана листа са NULL показивачима и са сентинелом

Постојање сентинеле значајно олакшава имплементације процедура за уметање и брисања елемената листе јер нема потребе испитивати специјалне случајеве да ли је неки показивач NULL.

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

```
struct cvor {
  char c;
  cvor *prethodni, *sledeci;
};

cvor* napravi_cvor(char c, cvor* p = nullptr, cvor* s = nullptr) {
  cvor* novi = new cvor();
  // potrebna je provera da li je alokacija uspela
  novi->c = c;
  novi->prethodni = p;
  novi->sledeci = s;
  return novi;
}

// brise cvor na koji ukazuje pokazivac cv (pretpostavljamo da to nije
// sentinela)
void obrisi(cvor* cv) {
  cv->prethodni->sledeci = cv->sledeci;
  cv->sledeci->prethodni = cv->prethodni;
  delete cv;
}

// ubacuje novi cvor sa karakterom c ispred cvora na koji ukazuje
// pokazivac cv
void ubaci_ispred(cvor* cv, char c) {
  cvor* novi = napravi_cvor(c, cv->prethodni, cv);
  novi->prethodni->sledeci = novi;
  novi->sledeci->prethodni = novi;
}

// ispisuje celu listu
void ispisi(cvor* sentinela) {
```

```
  for (cvor* cv = sentinela->sledeci; cv != sentinela; cv = cv->sledeci)
    cout << cv->c;
  cout << endl;
}

int main() {
  string naredbe;
  cin >> naredbe;
  int i = 0;

  // vestacki ubacen "prazan" cvor koji oznacava poziciju ispred
  // pocetka tj. iza kraja liste
  cvor* sentinela = napravi_cvor('_');
  sentinela->prethodni = sentinela->sledeci = sentinela;

  // tekuca pozicija kursora
  cvor* kursor = sentinela;

  while (i < naredbe.size()) {
    char naredba = naredbe[i++];
    if (naredba == '<') {
      if (kursor->prethodni != sentinela)
        kursor = kursor->prethodni;
    } else if (naredba == '>') {
      if (kursor != sentinela)
        kursor = kursor->sledeci;
    } else if (naredba == 'i') {
      char c = naredbe[i++];
      ubaci_ispred(kursor, c);
    } else if (naredba == 'b') {
      if (kursor->prethodni != sentinela)
        obrisi(kursor->prethodni);
    } else if (naredba == 'd') {
      if (kursor != sentinela) {
        cvor* sl = kursor->sledeci;
        obrisi(kursor);
        kursor = sl;
      }
    }
  }
  ispisi(sentinela);

  return 0;
}
```

4.6.3 Имплементација стека

Постоји неколико начина да се имплементира стек. Они обично подразумевају да се користи неки облик низа или једноструко повезане листе (при чему се елементи морају додавати на почетак и скидати са почетка листе). Имплементација треба да узме у обзир да ли је број елемената који се истовремено могу наћи на стеку ограничен (тада се може користити обичан статички низ) или није (тада је потребно користити неки низ који се динамички може ширити или листу).

Стек `stack` у језику C++ подразумевано користи вектор (колекцију `vector`).

Задатак: Историја веб-прегледача

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

Пошто је број сајтова ограничен текстом задатка, стек се може имплементирати коришћењем статичког низа.

```
// istorija pregledaca - rucno implementiran stek koji moze da cuva
// najvise 1000 adresa (to je dato u tekstu zadatka)
string istorija[1000];
// pokazivac na naredno slobodno mesto na steku
int sp = 0;

string linija;
while (getline(cin, linija)) {
  if (linija == "back") {
    if (sp > 0)
      sp--;
    if (sp > 0)
      cout << istorija[sp-1] << endl;
    else
      cout << "-" << endl;
  }
  else {
    cout << linija << endl;
    istorija[sp++] = linija;
  }
}
```

Стек може бити имплементиран коришћењем једноструко повезане листе.

```
// cvor jednostruko povezane liste
struct cvor {
  string s;
  cvor* sledeci;
};

cvor* napravi_cvor(const string& s, cvor* sledeci) {
  cvor* novi = new cvor();
  // trebalo bi proveriti da li je alokacija uspela
  novi->s = s;
  novi->sledeci = sledeci;
  return novi;
}

// dodavanje elementa na pocetak
cvor* dodaj(cvor* pocetak, const string& s) {
  return napravi_cvor(s, pocetak);
}

// brisanje elementa sa pocetka
cvor* obrisi(cvor* pocetak) {
  cvor* sledeci = pocetak->sledeci;
  delete pocetak;
  return sledeci;
}

// brisanje cele liste
void obrisi_listu(cvor* pocetak) {
  while (pocetak != nullptr) {
    cvor* sledeci = pocetak->sledeci;
    delete pocetak;
    pocetak = sledeci;
```

```
  }
}

int main() {
  // istorija pregledaca - stek implementiran preko liste
  cvor* pocetak = nullptr;

  string linija;
  while (getline(cin, linija)) {
    if (linija == "back") {
      if (pocetak != nullptr)
        pocetak = obrisi(pocetak);
      if (pocetak != nullptr)
        cout << pocetak->s << endl;
      else
        cout << "-" << endl;
    }
    else {
      cout << linija << endl;
      pocetak = dodaj(pocetak, linija);
    }
  }

  obrisi_listu(pocetak);
  return 0;
}
```

4.6.4 Имплементација реда

Постоји неколико начина да се имплементира ред. Они обично подразумевају да се користи неки облик низа или листе. Имплементација треба да узме у обзир да ли је број елемената који се истовремено могу наћи на стеку ограничен (тада се може користити обичан статички низ у који се елементи могу смештати у круг) или није (тада је потребно користити неки низ који се динамички може ширити или листу). Ред `queue` у језику C++ подразумевано користи дек тј. `deque` (ред са два краја имплементира преко специјализоване структуре података дек).

Задатак: Последњих k линија

Напиши програм који исписује k последњих линија учитаних са стандардног улаза.

Улаз: Са стандардног улаза се учитава број k ($1 \leq k \leq 100$), а затим једна по једна линија текста (њих највише 10^6).

Излаз: На стандардни излаз исписати последњих k линија (претпоставити да је увек учитано барем k линија).

Пример

Улаз	*Излаз*

`2`
`ispisati`
`poslednjih k`
`linija`
	`poslednjih k`
	`linija`

Решење

Пошто знамо максимални капацитет реда (максимални број елемената који се у неком тренутку налазе у реду), ред је могуће имплементирати коришћењем низа у који се елементи смештају кружно (када се низ попуни до краја, смештање елемената се наставља од почетка). Одржаваћемо две позиције тј. два показивача: на први елемент у реду и на прву слободну позицију (иза последњег елемента у реду). На почетку ће обе бити иницијализоване на нулу. Када год су те две једнаке, знамо да је ред празан. Да бисмо разликовали ситуацију потпуно попуњеног реда и празног реда, низ ћемо алоцирати тако да може да чува један елемент више од максималног капацитета. Тада ћемо потпуно попуњен ред моћи да препознамо тако што ће се слободна позиција налазити непосредно испред првог елемента у реду (специјално, могуће је да први елемент буде на

позицији 0, а да је последња слободна позиција последња позиција у низу). Додавање вршимо на слободну позицију и повећавамо је за 1 (наравно, по модулу дужине низа). Скидање са почетка вршимо тако што позицију првог елемента у реду увећамо за 1 (поново, по модулу дужине низа). Број елемената у реду можемо израчунати као разлику између два показивача (опет рачунату по модулу дужине низа, коришћењем везе $(k - p) \bmod n = (k - p + n) \bmod n$, да би се избегло рачунање остатка при дељењу негативних бројева).

Анализа сложености. Сложеност свих операција је $O(1)$, а меморијска сложеност је $O(k)$, где је k максимални капацитет.

Напомена. Приметимо да се на овај начин може имплементирати и ред са два краја, ако му је познат максимални капацитет.

```
vector<string> red;
int poc = 0, kraj = 0;

bool pun() {
  return (kraj + 1) % red.size() == poc;
}

bool prazan() {
  return kraj == poc;
}

void dodaj(const string& s) {
  red[kraj] = s;
  kraj = (kraj + 1) % red.size();
}

string& pocetak() {
  return red[poc];
}

void skini() {
  poc = (poc + 1) % red.size();
}

int velicina() {
  return (kraj - poc + red.size()) % red.size();
}
```

Ред је могуће имплементирати и ручно, коришћењем једноструко повезане листе. Чувају се показивач на први и последњи елемент (оба имају вредност NULL када је ред празан). Пошто уклањање последњег елемента једноструко повезане листе није могуће извршити ефикасно, чак и када је познат показивач на последњи елемент, додавање се врши на крај реда, а читање и скидање елемената са почетка. Број елемената одржавамо у посебној променљивој.

Анализа сложености. Сложеност свих операција је $O(1), а меморијска сложеност је $O(k)$, где је k максимални капацитет.

Напомена. Имплементација реда са два краја захтева коришћење двоструко повезане листе. Наиме, уклањање последњег елемента једноструко повезане листе није могуће извршити у времену $O(1)$, јер је потребна измена претпоследњег елемента.

```
struct cvor {
  string s;
  cvor* sledeci;
};

cvor* napravi_cvor(const string& s) {
  cvor* novi = new cvor();
  // trebalo bi proveriti da li je alokacija uspela
  novi->s = s;
```

```
  novi->sledeci = nullptr;
  return novi;
}

cvor* pocetak = nullptr;
cvor* kraj = nullptr;
int velicina = 0;

void dodaj(const string& s) {
  if (kraj == nullptr)
    pocetak = kraj = napravi_cvor(s);
  else {
    kraj->sledeci = napravi_cvor(s);
    kraj = kraj->sledeci;
  }
  velicina++;
}

bool prazan() {
  return velicina == 0;
}

const string& pocetni() {
  return pocetak->s;
}

void ukloni() {
  cvor* sledeci = pocetak->sledeci;
  delete pocetak;
  pocetak = sledeci;
  if (pocetak == nullptr)
    kraj = nullptr;
  velicina--;
}
```

4.6.5 Имплементација дека

Као што смо видели у поглављу о применама структура података, једна веома корисна структура података је ред са два краја која комбинује функционалност стека и реда.

Ако се за имплементацију користе једноструко повезане листе, тада се у времену $O(1)$ може вршити убацивање на почетак и брисање са почетка, као и убацивање на крај (под претпоставком да одржавамо показивач на крај). Брисање са краја је операција сложености $O(n)$, чак и када се чува показивач на последњи елемент (јер не можемо да пронађемо показивач на претпоследњи елемент).

Ако се за имплементацију користе двоструко повезане листе, тада се убацивање и брисање и са почетка и са краја може извршити у времену $O(1)$. Исто је и са уметањем елемента у средину (када се зна његова позиција). Међутим, индексни приступ (приступ члану на датој позицији) у најгорем случају захтева време $O(n)$. Једини могући начин претраге је линеарна претрага, чак и када су елементи у реду сортирани.

У наставку ћемо приказати структуру података **дек** (енгл. deque) која се често користи за имплементацију редова са два краја. Слично као двоструко повезане листе, дек омогућава додавање и на почетак и на крај у времену $O(1)$ (додуше амортизованом). Важна предност у односу на двоструко повезане листе је то што је индексни приступ могућ у времену $O(1)$. Овим је омогућена и бинарна претрага, што може некада бити веома важно. Додавање и брисање елемената са средине није могуће извршити ефикасно (те операције захтевају време $O(n)$).

Прикажимо и једну могућу имплементацију дека.

Задатак: Дек

Програм подржава следеће наредбе које се извршавају над низом целих бројева.

- `F x` — додавање елемента `x` на почетак низа
- `f` — уклањање и испис елемента са почетка низа на стандардни излаз
- `B x` — додавање елемента `x` на крај низа
- `b` — уклањање и испис елемента са краја низа на стандардни излаз
- `r p` — испис елемента на позицији `p` на стандардни излаз (позиција је између 0 и тренутног броја елемената низа, умањеног за 1).

Улаз: Стандардни улаз садржи низ наредби, сваку у посебном реду.

Излаз: На стандардни излаз исписати ефекат извшавања наредби.

Пример

Улаз	*Излаз*
`F 1`	`2`
`B 3`	`1`
`F 2`	`3`
`B 4`	`4`
`r 0`	`2`
`r 1`	`1`
`r 2`	`4`
`r 3`	`3`
`f`	
`r 0`	
`b`	
`r 1`	

Решење

Задатак се једноставно може решити помоћу библиотечке имплементације дека.

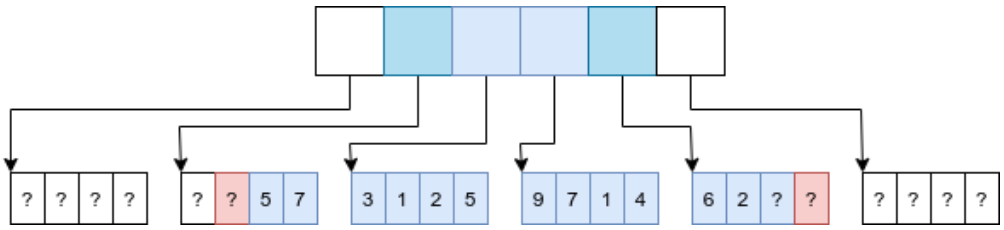
```
#include <iostream>
#include <deque>

using namespace std;

int main() {
  deque<int> d;
  char c;
  while (cin >> c) {
    if (c == 'F') {
      int x;
      cin >> x >> ws;
      d.push_front(x);
    } else if (c == 'B') {
      int x;
      cin >> x >> ws;
      d.push_back(x);
    } else if (c == 'f') {
      cout << d.front() << endl;
      d.pop_front();
    } else if (c == 'b') {
      cout << d.back() << endl;
      d.pop_back();
    } else if (c == 'r') {
      int p;
      cin >> p >> ws;
      cout << d[p] << endl;
    }
  }
```

```
  return 0;
}
```

Дек можемо замислити као низ сегмената исте, фиксне величине. Сваки сегмент је структура која садржи низ елемената (било статички, било динамички алоциран) који представља неки део реда. На пример, ред чији су елементи 5, 7, 3, 1, 2, 5, 9, 7, 1, 4, 6, 2, може бити организован у 6 сегмената од по 4 елемента (у пракси је величина појединачних сегмената обично већа).



Слика 4.2: Пример имплементације дека

Дек чува низ показивача на појединачне сегменте. Два сегмента су карактеристична: леви сегмент у ком се налази почетак реда и десни сегмент у ком се налази крај реда. Оба могу бити само делимично попуњени. У сваком сегменту се чува прва слободна позиција на коју се може додати наредни елемент. Посебно се одржавају позиције та два сегмента. У празном деку леви и десни сегмент су суседни и празни (текући елемент левог сегмента је његов десни крај, а десног сегмента је његов леви крај).

Додавање елемента на почетак је веома једноставно ако леви сегмент није попуњен до краја. Елемент се само додаје на прву слободну позицију и она се помера налево. Када је леви сегмент потпуно попуњен, прелази се на попуњавање претходног сегмента, ако он постоји. Ако не постоји, онда се врши реалокација дека и проширује се његов број сегмената (обично је нови број сегмената неколико пута већи него полазни, како би се наредне реалокације дешавале све ређе и ређе и како би се обезбедило амортизовано константно време додавања). Приликом реалокације врши се само проширивање низа показивача на сегменте, а не самих сегмената, што је веома значајно ако се у деку чувају већи објекти (они се приликом алокације не копирају, а не копирају се ни сами сегменти). Приликом реалокације, показивачи на постојеће сегменте се у проширеном низу смештају на средину, а лево и десно од њих се смештају показивачи на ново алоциране сегменте који су иницијално празни. Реалокацијом се добијају сегменти лево од текућег потпуно попуњеног сегмента и додавање на почетак се врши у њих. Додавање на десни крај тече потпуно аналогно.

Брисање са левог краја се врши слично, уклањањем елемента из левог сегмента и преласком на наредни сегмент ако се након брисања леви сегмент потпуно испразнио. Наравно, елементи сегмента се не уклањају физички, већ се само мењају индекси који указују на наредну слободну позицију. Брисање са десног краја је аналогно.

Индексни приступ елементу је могуће извршити у времену $O(1)$, тако што се прво одреди ком сегменту припада тражени елемент, а онда се прочита одговарајући елемент из тог сегмента. Једноставан трик је да се тражени индекс увећа за број празних елемената на левом крају левог сегмента. Тада се позиција сегмента у ком се елемент налази може једноставно израчунати као збир позиције левог сегмента и целобројног количника индекса i и величине једног сегмента, док се позиција унутар тог сегмента одређује као остатак у том дељењу.

Имплементација дека садржи велики број техничких детаља, па је приказујемо само информативно (читаоца упућујемо да покуша да самостално имплементира дек).

```
#include <iostream>

using namespace std;

// jedan segment
struct segment {
  // niz podataka
  int* podaci;
  // broj popunjenih elemenata niza
  int popunjeno;
```

```
  // pozicija u nizu na koju se moze ubaciti naredni levi element
  int levi;
  // pozicija u nizu na koju se moze ubaciti naredni desni element
  int desni;
};

// alokacija segmenta
segment* alocirajSegment(int velicina) {
  // alociramo novi segment
  segment* novi = new segment();
  // alociramo njegove podatke
  novi->podaci = new int[velicina];
  // nijedan podatak još nije upisan
  novi->popunjeno = 0;
  // trenutno nepopunjeni deo
  novi->levi = velicina - 1;
  novi->desni = 0;
  return novi;
}

// brisanje segmenta
void obrisiSegment(segment* s) {
  // brisemo podatke u segmentu
  delete[] s->podaci;
  // brisemo segment
  delete s;
}

struct dek {
  // niz pokazivača na segmente
  segment** segmenti;
  // broj segmenata u nizu
  int brojSegmenata;
  // velicina svakog segmenta
  int velicinaSegmenata;
  // pozicije na kojima se nalaze pokazivaci na krajnji levi
  // i krajnji desni segment (oba mogu biti polupopunjena)
  int levo, desno;
  int brojElemenata;
};


// vrši se realociranje deka tako da ima dati broj segmenata
void realocirajDek(dek& d, int brojSegmenata) {
  // pravimo novi niz pokazivača na segmente
  segment** noviSegmenti = new segment*[brojSegmenata];
  // postojeće pokazivače na segmente ćemo smestiti negde oko
  // sredine novog niza
  int uvecanje = (brojSegmenata - d.brojSegmenata) / 2;
  // ako je postojao stari niz pokazivača na segmente
  if (d.segmenti != nullptr)
    // kopiramo te pokazivače u novi niz pokazivača, krenuvši od
    // pozicije uvecanje
    for (int i = 0; i < d.brojSegmenata; i++)
      noviSegmenti[uvecanje + i] = d.segmenti[i];
  // alociramo nove segmente na početku i na kraju novog niza
  for (int i = 0; i < uvecanje; i++)
    noviSegmenti[i] = nullptr;
```

```
  for (int i = uvecanje + d.brojSegmenata; i < brojSegmenata; i++)
    noviSegmenti[i] = nullptr;

  // brišemo stari niz pokazivača na segmente
  delete[] d.segmenti;
  // preusmeravamo pokazivač na niz pokazivača
  d.segmenti = noviSegmenti;
  // ažuriramo broj segmenata
  d.brojSegmenata = brojSegmenata;
  // ažuriramo granice popunjenog dela niza
  d.levo += uvecanje;
  d.desno += uvecanje;
}


// vraća referencu na i-ti element u deku
int& iti(const dek& d, int i) {
  // trenutni broj elemenata u krajnjem levom segmentu
  int uLevom = d.segmenti[d.levo]->popunjeno;
  // pomeramo indeks tako da je pozicija 0 na početku krajnjeg levog
  // segmenta
  i += d.velicinaSegmenata - uLevom;
  // određujemo segment u kom se nalazi traženi element
  segment* s = d.segmenti[d.levo + i / d.velicinaSegmenata];
  // čitamo element iz tog segmenta
  return s->podaci[i % d.velicinaSegmenata];
}

// dodajemo element na početak deka
void dodajNaPocetak(dek& d, int x) {
  // ako je levi segment jos nealociran, alociramo ga
  if (d.segmenti[d.levo] == nullptr)
    d.segmenti[d.levo] = alocirajSegment(d.velicinaSegmenata);

  // ako je levi segment potpuno popunjen с леве стране
  if (d.segmenti[d.levo]->levi < 0) {
    // ako levo od njega nema više alociranih segmenata
    if (d.levo == 0)
      // realociramo dek i time alociramo nove segmente
      realocirajDek(d, d.brojSegmenata * 2);
    // prelazimo u prethodni segment
    d.levo--;
  }
  // segment u koji se može upisati element
  segment *s = d.segmenti[d.levo];

  // upisujemo element na tekuću slobodnu poziciju i pomeramo se
  // nalevo
  s->podaci[s->levi--] = x;
  // uvećavamo broj popunjenih elemenata
  s->popunjeno++;
  d.brojElemenata++;
}

// dodajemo element na kraj deka
void dodajNaKraj(dek& d, int x) {
  // ako je desni segment jos nealociran, alociramo ga
  if (d.segmenti[d.desno] == nullptr)
```

```
    d.segmenti[d.desno] = alocirajSegment(d.velicinaSegmenata);

  // ako je desni segment potpuno popunjen sa desne strane
  if (d.segmenti[d.desno]->desno >= d.velicinaSegmenata) {
    // ako desno od njega nema više alociranih segmenata
    if (d.desno == d.brojSegmenata - 1)
      // realociramo dek i time alociramo nove segmente
      realocirajDek(d, d.brojSegmenata * 2);
    // prelazimo na naredni segment
    d.desno++;
  }
  // segment u koji se može upisati element
  segment *s = d.segmenti[d.desno];

  // upisujemo element na tekuću slobodnu poziciju i pomeramo se
  // nalevo
  s->podaci[s->desni++] = x;
  // uvećavamo broj popunjenih elemenata
  s->popunjeno++;
  d.brojElemenata++;
}

// uklanjamo element sa pocetka deka
int ukloniSaPocetka(dek& d) {
  if (d.segmenti[d.levo]->levi >= d.velicinaSegmenata) {
    d.levo++;
    d.segmenti[d.levo]->levi = 0;
  }
  segment *s = d.segmenti[d.levo];
  s->popunjeno--;
  d.brojElemenata--;
  return s->podaci[++s->levi];
}


// uklanjamo element sa kraja deka
int ukloniSaKraja(dek& d) {
  if (d.segmenti[d.desno]->desni < 0) {
    d.desno--;
    d.segmenti[d.desno]->desni = d.velicinaSegmenata;
  }
  segment *s = d.segmenti[d.desno];
  s->popunjeno--;
  d.brojElemenata--;
  return s->podaci[--s->desni];
}

void ispisiDek(dek& d) {
  for (int i = 0; i < d.brojElemenata; i++)
    cout << iti(d, i) << " ";
  cout << endl;
}


// brisanje deka
void obrisiDek(dek& d) {
  // brisemo sve segmente
  for (int i = 0; i < d.brojSegmenata; i++)
```

```
    obrisiSegment(d.segmenti[i]);
  // brisemo niz pokazivaca na segmente
  delete[] d.segmenti;
}

void inicijalizujDek(dek& d, int brojElemenata, int velicinaSegmenata = 3) {
  d.velicinaSegmenata = velicinaSegmenata;
  d.segmenti = nullptr;
  d.brojElemenata = 0;
  d.brojSegmenata = 0;
  d.levo = -1;
  d.desno = 0;

  realocirajDek(d, brojElemenata);
}


int main() {
  // gradimo prazan dek i zatim ga inicijalizujemo tako da inicijalno
  // moze da primi 10 elemenata
  dek d;
  inicijalizujDek(d, 10);

  char c;
  while (cin >> c) {
    if (c == 'F') {
      int x;
      cin >> x >> ws;
      dodajNaPocetak(d, x);
    } else if (c == 'B') {
      int x;
      cin >> x >> ws;
      dodajNaKraj(d, x);
    } else if (c == 'f') {
      cout << ukloniSaPocetka(d) << endl;
    } else if (c == 'b') {
      cout << ukloniSaKraja(d) << endl;
    } else if (c == 'r') {
      int p;
      cin >> p >> ws;
      cout << iti(d, p) << endl;
    }
    ispisiDek(d);
  }

  // brisemo dek
  obrisiDek(d);

  return 0;
}
```

4.6.6 Бинарно дрво претраге - имплементација скупова и мапа

Дрвета су структуре података које се користе за представљање хијерархијских односа између делова (на пример, израза, система директоријума и датотека, организације елемената унутар HTML странице, синтаксе програма унутар преводиоца и слично). Бинарно дрво је рекурзивно дефинисани тип података: или је празно или садржи неки податак и лево и десно поддрво.

Уобичајени начин за представљање дрвета у језику C++ је преко чворова увезаних помоћу показивача.

```
struct cvor {
  int x;
  cvor *levo, *desno;
};
```

У случају да се дрвета користе за представљање скупова или мултискупова у чворовима се чува само један податак (елемент скупа). Ако се користе за представљање мапа или мултимапа, у чворовима се чувају два податка: кључ и вредност придружена кључу.

```
struct cvor {
  int k, v;
  cvor *levo, *desno;
}
```

Пошто је дрво рекурзивно-дефинисана структура података, најлакше је функције које оперишу са дрветима реализовати рекурзивно. У неким ситуацијама је могуће релативно лако елиминисати рекурзију, док је у неким другим ситуацијама имплементирање нерекурзивних операција компликовано (и захтева коришћење стека). Већина функција које ћемо ми имплементирати ће бити рекурзивна.

У наставку ћемо се бавити употребом дрвета за имплементацију асоцијативних контејнера (скупова, мултискупова, мапа и мултимапа). За то се користи посебна организација вредности у бинарном дрвету.

Дрво је **уређено бинарно дрво** (тј. **бинарно дрво претраге**, engl. binary search tree), ако је празно или ако је његово лево и десно поддрво уређено и ако је чвор у корену већи од свих чворова у левом поддрвету и мањи од свих чворова у десном поддрвету. Оваква дефиниција подразумева да у дрвету нема дупликата, што је случај у обичним скуповима и мапама. У мултискуповима и мултимапама је дозвољено постојање дупликата у дрвету и тада се дефиниција проширује тако што се допушта да је чвор у корену већи или једнак од свих чворова у десном поддрвету.

Нагласимо да није довољно проверити да је вредност у сваком чвору већа од вредности у корену левог поддрвета и вредности у корену десног поддрвета! На пример, то важи у наредном дрвету, али оно није претраживачко, јер је елемент 4 који је већи од елемента 3 завршио лево од њега.

```
    3
  2   5
1   4
```

4.6.6.1 Балансирано бинарно дрво

Мана класичних уређених бинарних дрвета је то што ако нису балансирана операције претраге, уметања и брисања могу захтевати линеарно време у односу на број чворова у дрвету. Балансирана бинарна дрвета гарантују да се то не може догодити и да је временска сложеност најгорег случаја ових операција логаритамска у односу на број чворова у дрвету. Две најчешће коришћене врсте балансираних бинарних дрвета.

- Адељсон-Вељски Ландисова дрвета (енгл. AVL tree)
- Црвено-црна дрвета (енгл. Red-black tree, RBT)

Њихов детаљан опис превазилази домен овог материјала.

Задатак: Мапа

Над речником који слика ниске карактера дужине највише 10 карактера у целе бројеве се могу вршити следеће наредбе:

- `w k v` — *write* — кључу `k` се придружује вредност `v` (ако је кључу `k` већ додељена нека вредност, она се занемарује)
- `r k` — *read* — на стандардни излаз се исписује вредност придружена кључу `k`. Ако кључу није придружена вредност, исписује се `-`.

Написати програм који чита и извршава низ оваквих наредби (наравно, водити рачуна о ефикасности).

Улаз: Свака линија стандардног улаза садржи једну наредбу.

Излаз: На стандардни излаз исписати резултат извршавања свих наредби.

Пример

Улаз	*Излаз*
`w pera 5`	`4`
`w ana 4`	`-`
`r ana`	`3`
`r mika`	`5`
`w mika 3`	
`r mika`	
`r pera`	

Решење

Задатак можемо решити преко ручно имплементираног уређеног бинарног дрвета. Једноставности ради, нећемо користити балансирано бинарно дрво (мада се тиме нарушава сложеност најгорег случаја, када се кључеви јављају у релативно сортираном редоследу).

```
#include <iostream>

using namespace std;

struct cvor {
  string kljuc;
  int vrednost;
  cvor *levo, *desno;
};

cvor* napravi_cvor(const string& kljuc, int vrednost) {
  cvor* novi = new cvor();
  // potrebno je proveriti da li je alokacija cvora uspela
  novi->kljuc = kljuc;
  novi->vrednost = vrednost;
  novi->levo = novi->desno = nullptr;
  return novi;
}

cvor* ubaci(cvor* koren, const string& kljuc, int vrednost) {
  if (koren == nullptr)
    return napravi_cvor(kljuc, vrednost);
  if (kljuc < koren->kljuc)
    koren->levo = ubaci(koren->levo, kljuc, vrednost);
  else if (kljuc > koren->kljuc)
    koren->desno = ubaci(koren->desno, kljuc, vrednost);
  else
    koren->vrednost = vrednost;
  return koren;
}

cvor* pronadji(cvor* koren, const string& kljuc) {
  if (koren == nullptr)
    return nullptr;
  if (kljuc < koren->kljuc)
    return pronadji(koren->levo, kljuc);
  else if (kljuc > koren->kljuc)
    return pronadji(koren->desno, kljuc);
  return koren;
}

bool pronadji(cvor* koren, const string& kljuc, int& vrednost) {
  cvor* c = pronadji(koren, kljuc);
  if (c == nullptr)
```

```
    return false;
  vrednost = c->vrednost;
  return true;
}

void obrisi_drvo(cvor* koren) {
  if (koren != nullptr) {
    obrisi_drvo(koren->levo);
    obrisi_drvo(koren->desno);
    delete koren;
  }
}

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);

  cvor* koren = nullptr;

  char c;
  while (cin >> c) {
    if (c == 'w') {
      string k; int v;
      cin >> k >> v >> ws;
      koren = ubaci(koren, k, v);
    } else if (c == 'r') {
      string k;
      cin >> k >> ws;
      int v;
      if (pronadji(koren, k, v))
        cout << v << '\n';
      else
        cout << "-" << endl;
    }
  }

  obrisi_drvo(koren);

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Скуп

Над скупом елемената могу се вршити следеће наредбе:

- `i x` — *insert* — уметање вредности `x` у скуп (ако се она већ налази у скупу, скуп се не мења)
- `e x` — *erase* — брисање вредности `x` из скупа елемената (ако се она не налази у скупу, скуп се не мења)
- `m` — *min* — на стандардни излаз исписује се минимални елемент скупа или `-` ако је скуп празан
- `M` — *max* — на стандардни излаз исписује се максимални елемент скупа или `-` ако је скуп празан
- `n x` — *next* — на стандардни излаз исписује се следбеник елемента `x` тј. најмањи број у скупу који је строго већи од `x` (ако такав елемент не постоји, исписује се `-`)

Написати програм који чита и извршава низ оваквих наредби (наравно, водити рачуна о ефикасности).

Улаз: Свака линија стандардног улаза садржи једну наредбу.

Излаз: На стандардни излаз исписати резултат извршавања свих наредби.

Пример

Улаз	*Излаз*
i 3	3
i 5	5
i 4	4
m	5
M	8
e 3	-
m	
i 8	
n 4	
n 7	
n 8	

Решење

Задатак се лако решава коришћењем библиотечке имплементације уређеног скупа. У језику C++ могуће је користити тип `set<int>` (али не и `unordered_set<int>`).

```
#include <iostream>
#include <set>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  char c;
  set<int> s;
  while (cin >> c) {
    if (c == 'i') {
      int x;
      cin >> x >> ws;
      s.insert(x);
    } else if (c == 'e') {
      int x;
      cin >> x >> ws;
      s.erase(x);
    } else if (c == 'm') {
      if (!s.empty())
        cout << *s.begin() << '\n';
      else
        cout << "-" << '\n';
    } else if (c == 'M') {
      if (!s.empty())
        cout << *prev(s.end()) << '\n';
      else
        cout << "-" << '\n';
    } else if (c == 'n') {
      int x;
      cin >> x >> ws;
      auto it = s.upper_bound(x);
      if (it != s.end())
        cout << *it << '\n';
      else
        cout << "-" << '\n';
    }
  }
  return 0;
}
```

Задатак можемо решити и ручном имплементацијом скупа коришћењем претраживачког (уређеног) бинарног

дрвета.

Уметање у бинарно дрво претраге се дефинише веома једноставном рекурзивном конструкцијом. Уметање се увек врши на место листа. Вредност која се умеће се пореди са вредношћу у корену, ако је мања од ње умеће се лево, а ако је већа од ње умеће се десно. Ако је једнака вредности у корену, разликују се случај у ком се дупликати допуштају (тада дрво представља скуп) и случај у ком се не допуштају (тада дрво представља мултискуп). Ако се дупликати не допуштају, тада се приликом детекције да се вредност већ налази у корену дрвета, уметање просто прекида (у супротном би се уметање наставило у левом поддрвету).

Брисање је донекле компликованије реализовати него уметање. Можемо га реализовати наредним рекурзивним алгоритмом.

- ако је елемент који се брише мањи од корена, он се брише из левог поддрвета
- ако је елемент који се брише већи од корена, он се брише из десног поддрвета
- ако је једнак корену, брише се корен
  - ако десно поддрво не постоји, резултат је лево поддрво
  - у супротном се брише чвор који садржи најмањи елемент из десног поддрвета и та вредност се уписује у корен. Ова операције се може једноставно извести засебном рекурзивном конструкцијом.

Приликом брисања корена, у корен је могао бити доведен и највећи елемент левог поддрвета.

Минимум се одређује веома једноставно, као крајњи леви потомак корена (у овом случају није потребно користити рекурзивну имплементацију). Максимум се одређује аналогно.

Следбеник датог броја се лако може одредити рекурзивном конструкцијом.

- Ако је дрво празно, следбеник не постоји (у имплементацији је могуће вратити показивач NULL).
- Ако је вредност у корену мања или једнака од датог броја, тада се следбеник рекурзивно одређује у десном поддрвету (јер он мора бити строго већи од датог броја, а вредности лево од корена и у њему нису такве). Наравно, ако десно дрво не постоји, биће враћен показивач NULL.
- У супротном се следбеник рекурзивно одређује у левом поддрвету. Ако он постоји, то је коначан резултат, а ако не постоји, тада је следбеник корен.

Ову функцију је могуће имплементирати и нерекурзивно.

Анализа сложености. Под претпоставком да је дрво балансирано, сложеност свих описаних операција је $O(\log n)$, где је n број чворова у дрвету. Наиме, у свим случајевима се анализира или само лево или само десно поддрво, па је једначина којом се описује број корака $T(n) = T(n/2) + O(1)$.

```
#include <iostream>

using namespace std;

struct cvor {
  int x;
  cvor *levo, *desno;
};

cvor* napraviCvor(int x) {
  cvor* novi = new cvor();
  // potrebna je provera da li je alokacija uspela
  novi->levo = novi->desno = nullptr;
  novi->x = x;
  return novi;
}

cvor* ubaci(cvor* drvo, int x) {
  if (drvo == nullptr)
    return napraviCvor(x);
```

```
  if (x < drvo->x)
    drvo->levo = ubaci(drvo->levo, x);
  else if (x > drvo->x)
    drvo->desno = ubaci(drvo->desno, x);
  return drvo;
}

int minimum(cvor* drvo) {
  while (drvo->levo != nullptr)
    drvo = drvo->levo;
  return drvo->x;
}

int maksimum(cvor* drvo) {
  while (drvo->desno != nullptr)
    drvo = drvo->desno;
  return drvo->x;
}

// brisanje date vrednosti
// Iz datog drveta se ukljanja čvor sa najmanjom vrednošću
cvor* obrisiMin(cvor* drvo, int& x) {
   // iz praznog drveta nema šta da se briše
   if (drvo == nullptr) return nullptr;
   // ako je levo poddrvo prazno
   if (drvo->levo == nullptr) {
      // brišemo drvo i vraćamo desno poddrvo
      cvor* desno = drvo->desno;
      x = drvo->x;
      delete drvo;
      return desno;
    }
    // u suprotnom brišemo najmanji element levog poddrveta
    drvo->levo = obrisiMin(drvo->levo, x);
    return drvo;
}

// Iz datog drveta ukljanja dati čvor sa datom vrednošću
cvor* obrisi(cvor* drvo, int x) {
  // iz praznog drveta nema šta da se briše
  if (drvo == nullptr)
    return nullptr;
  // čvor se nalazi levo, pa se tamo i briše
  if (x < drvo->x)
    drvo->levo = obrisi(drvo->levo, x);
  // čvor se nalazi desno, pa se tamo i briše
  else if (x > drvo->x)
    drvo->desno = obrisi(drvo->desno, x);
  // čvor se nalazi u korenu
  else {
    if (drvo->desno == nullptr) {
      // ako je desno poddrvo prazno brišemo drvo i vraćamo levo poddrvo
      cvor* levo = drvo->levo;
      delete drvo;
      return levo;
    } else {
      // desno poddrvo nije prazno, pa brišemo najmanji čvor iz
      // njega i vrednost iz tog čvora upisujemo u drvo
```

```
      int min;
      drvo->desno = obrisiMin(drvo->desno, min);
      drvo->x = min;
    }
  }
  // fizički čvor u kome je drvo se nije promenio
  return drvo;
}


cvor* sledbenik(cvor* drvo, int x) {
  if (drvo == nullptr)
    return nullptr;
  if (drvo->x <= x)
    return sledbenik(drvo->desno, x);
  else {
    cvor* sl = sledbenik(drvo->levo, x);
    if (sl != nullptr)
      return sl;
    return drvo;
  }
}

// brisanje celog drveta
void obrisi(cvor* drvo) {
   if (drvo != nullptr) {
      obrisi(drvo->levo);
      obrisi(drvo->desno);
      delete drvo;
   }
}

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);

  char c;
  cvor* drvo = nullptr;

  while (cin >> c) {
    if (c == 'i') {
      int x;
      cin >> x >> ws;
      drvo = ubaci(drvo, x);
    } else if (c == 'e') {
      int x;
      cin >> x >> ws;
      drvo = obrisi(drvo, x);
    } else if (c == 'm') {
      if (drvo != nullptr)
        cout << minimum(drvo) << '\n';
      else
        cout << "-" << '\n';
    } else if (c == 'M') {
      if (drvo != nullptr)
        cout << maksimum(drvo) << '\n';
      else
        cout << "-" << '\n';
    } else if (c == 'n') {
```

```
      int x;
      cin >> x >> ws;
      cvor* s = sledbenik(drvo, x);
      if (s != nullptr)
        cout << s->x << '\n';
      else
        cout << "-" << '\n';
    }
  }
  obrisi(drvo);
  return 0;
}
```

4.6.7 Хип - имплементација реда са приоритетом

Претраживачко бинарно дрво се може употребити за имплементацију редова са приоритетом, уз релативно добре перформансе, под претпоставком да је дрво балансирано (AVL или RBT). Подсетимо се, основне операције су уметање елемента у ред, читање елемента са највећим приоритетом и његово избацивање. Уметање произвољне вредности се врши у логаритамској сложености, максимални вредност се може пронаћи у логаритамској сложености спуштајући се надесно докле год је то могуће и тај чвор (лист) је могуће једноставно уклонити, поново у логаритамској сложености. Додатно, могуће је претражити дрво, па и обрисати произвољну вредност у логаритамској сложености.

Ако се задржимо само на три основне функционалности, могућа је другачија имплементација која има боље перформансе него претраживачко дрво. Поново се користи бинарно дрво, али овај пут организовану у структуру података која се назива **хип** (енгл. heap).

Макс-хип (енгл. max-heap) је бинарно дрво које задовољава услов да је сваки ниво осим евентуално последњег потпуно попуњен, као и да је вредност сваког чвора већа или једнака од вредности у његовој деци. Мин-хип се дефинише аналогно, једино што се захтева да је вредност сваког чвора мања или једнака од вредности у његовој деци. Макс-хип омогућава веома ефикасно одређивање максималног елемента у себи (он се увек налази у корену), а видећемо и веома ефикасно његово уклањање. Пошто је и операција уметања новог елемента у хип ефикасна, ова структура података је веома добар кандидат за имплементацију реда са приоритетом.

Задатак: Хип

Програм препознаје следеће наредбе које се извршавају над колекцијом података (у којој се могу јавити и дупликати):

- `i x` — *insert* — у колекцију бројева умеће се вредност `x`
- `m` — *maximum* — на стандардни излаз се исписује највећи елемент који се тренутно налази у колекцији
- `r` — *remove* — из колекције се избацује највећи елемент

Напиши програм који чита и извршава низ оваквих наредби (наравно, водити рачуна о ефикасности).

Улаз: Свака линија стандардног улаза садржи једну наредбу.

Излаз: На стандардни излаз се исписује резултат извршавања свих наредби.

Пример

Улаз	*Излаз*
`i 3`	`7`
`i 7`	`8`
`i 4`	`4`
`m`	
`i 8`	
`m`	
`r`	
`r`	
`m`	

Решење

Описане наредбе су основне три наредбе реда са приоритетом.

Задатак се веома једноставно може решити коришћењем библиотечке имплементације реда са приоритетом.

```
#include <iostream>
#include <queue>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  priority_queue<int> pq;
  char c;
  while (cin >> c) {
    if (c == 'i') {
      int x;
      cin >> x >> ws;
      pq.push(x);
    } else if (c == 'm') {
      cout << pq.top() << '\n';
    } else if (c == 'r') {
      pq.pop();
    }
  }
  return 0;
}
```

Задатак се може решити употребом уређеног мултискупа (који се може имплементирати коришћењем уређеног бинарног дрвета). У језику C++ најједноставније је употребити библиотечку имплементацију `multiset`.

Анализа сложености. Уматање, брисање и проналажење најмањег елемента у мултискупу извршавају се у времену $O(\log k)$, где је k број елемената у мултискупу, па је укупна сложеност алгоритма $O(n \log n)$, где је n број наредби које се извршавају.

Анализа сложености. Уметање, брисање и претрага уређеног речника се извршава у времену $O(\log k)$, где је k број елемената у мултискупу. Нажалост, проналажење најмањег елемента се извршава у времену $O(k)$, па је стога, сложеност најгорег случаја алгоритма $O(n^2)$, где је n број наредби које се извршавају.

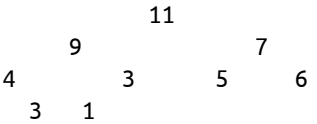
```
#include <iostream>
#include <set>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  multiset<int> pq;
  char c;
  while (cin >> c) {
    if (c == 'i') {
      int x;
      cin >> x >> ws;
      pq.insert(x);
    } else if (c == 'm') {
      cout << *(prev(pq.end())) << '\n';
    } else if (c == 'r') {
      pq.erase(prev(pq.end()));
    }
  }
  return 0;
}
```

Размотримо наредну имплементацију макс-хипа. Дрво смештамо у низ. Корен на позицију 0, наредна два

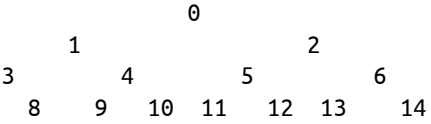
елемента на позиције 1 и 2, затим наредне елементе на позиције 3, 4, 5 и 6 итд. На пример, хип



представљамо низом

```
11 9 7 4 3 5 6 2 3 1
```

Захтевамо да се хип попуњава редом и су сви нивои осим евентуално последњег комплетно попуњени, а да су у последњем нивоу попуњени само почетни елементи. Напишимо позиције у низу и њихов распоред у бинарном дрвету.

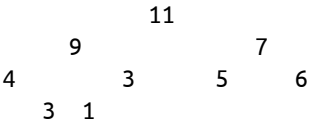


Лако је уочити да ако се корен налази на позицији i, онда се његово лево дете налази на позицији $2i + 1$, а десно дете на позицији $2i + 2$, док му се родитељ налази на позицији $\left\lfloor \frac{i-1}{2} \right\rfloor$.

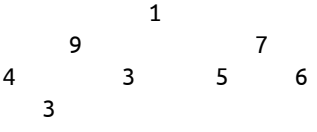
Највећи елемент се увек налази у корену, тј. на позицији 0 (што се лако може доказати индукцијом имајући у виду да је вредност у сваком чвору већа од вредности његовој у деци). Време потребно за ово је очигледно $O(1)$.

Размотримо како бисмо могли реализовати операцију уклањања највећег елемента из макс-хипа. Пошто се он налази у корену, а дрво мора бити потпуно попуњено (осим евентуално последњег нивоа) на место избаченог елемента је најједноставније уписати последњи елемент из хипа (најдешњи елемент последњег нивоа). Овим је задовољен услов за распоред елемената у дрвету, али је својство хипа могуће нарушено, јер тај елемент не мора бити већи од свих осталих (и обично није). На срећу, поправку можемо извршити релативно једноставно. Потребно је да упоредимо вредност у корену са вредношћу његове деце (ако постоје). Ако је вредност у корену већа или једнака од тих вредности, онда корен задовољава услов макс-хипа и процедура може да се заврши (јер за све остале чворове знамо да задовољавају тај услов, јер је избацивање кренуло од исправног хипа). У супротном, мењамо вредност у корену са већом од вредности његове деце (тј. са вредношћу његовог детета, ако има само једно дете). Након тога корен задовољава услов макс-хипа, и преостаје једино да проверимо (и евентуално поправимо) оно поддрво у чијем је корену завршила вредност корена. Ово је проблем истог облика, само мање димензије у односу на полазни и лако се, дакле, решава индуктивно-рекурзивном конструкцијом.

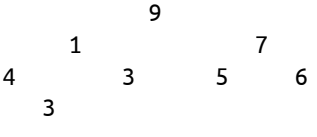
Прикажимо рад овог алгоритма на једном примеру.



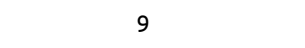
Избацујемо елемент са врха и на његово месту премештамо последњи елемент.



Мењамо вредност у корену, са већом од вредности његова два детета.



Исти поступак примењујемо на лево поддрво. Размењујемо вредност у корену са већом од вредности његова два детета.



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

Поступак се још једном примењује на поддрво са кореном 1.

```
             9
        4             7
   3        3      5     6
 2     1
```

Након овога, добијено дрво представља макс-хип.

Број корака померања наниже у најгорем случају одговара висини дрвета. Пошто у потпуном дрвету висине може да стане $2^{+1} - 1$ елемената, висина логаритамски зависи од броја елемената. Дакле, време потребно за уклањање највећег елемента из хипа у којем се налази n елемената је $O(\log n)$.

Уметање новог елемента функционише по сличном, али обрнутом принципу. Елемент је најједноставније убацити на крај низа. Ипак, могуће је да му ту није место, јер је можда већи од свог родитеља. У том случају могуће је извршити њихову размену. Након замене, поддрво T са кореном у новом чвору задовољава услов хипа и цело дрво без поддрвета T такође задовољава услов хипа. Сваким померањем новог елемента навише, остатак дрвета без поддрвета T се смањује и проблем се своди на проблем мање димензије и решава се индуктивно-рекурзивном конструкцијом.

Пошто је и у случању операције померања наниже и у случају операције померања навише рекурзија репна, она се може релативно једноставно уклонити.

Још један начин да имплементирамо померање наниже је да одредимо позицију већег од два детета и онда то дете упоредимо са родитељем.

```
#include <iostream>
#include <vector>

using namespace std;

int roditelj(int i) {
  return (i-1) / 2;
}

int levoDete(int i) {
  return 2*i + 1;
}

int desnoDete(int i) {
  return 2*i + 2;
}

int najveci(const vector<int>& hip) {
  return hip[0];
}

// element na poziciji k se pomera naniže, razmenom sa svojom decom
// sve dok se ne zadovolji uslov hipa
void pomeriNanize(vector<int>& hip, int k) {
  // pozicija najvećeg čvora (razmatrajući roditelja i njegovu decu)
  int najveci = k;
  int levo = levoDete(k), desno = desnoDete(k);
  if (levo < hip.size() && hip[levo] > hip[najveci])
    najveci = levo;
  if (desno < hip.size() && hip[desno] > hip[najveci])
    najveci = desno;
  // ako roditelj nije najveći
```

```
  if (najveci != k) {
    // menjamo roditelja i veće dete
    swap(hip[najveci], hip[k]);
    // rekurzivno obrađujemo veće dete
    pomeriNanize(hip, najveci);
  }
}

// izbacivanje najvećeg elementa iz hipa
void izbaciNajveci(vector<int>& hip) {
  // poslednji element izbacujemo iz hipa i
  // upisujemo ga na početnu poziciju
  hip[0] = hip.back();
  hip.pop_back();
  // pomeramo početni element naniže dok se ne
  // zadovolji uslov hipa
  pomeriNanize(hip, 0);
}

// element na poziciji k se pomera naviše, razmenom sa svojim
// roditeljem, sve dok se ne zadovolji uslov hipa
void pomeriNavise(vector<int>& hip, int k) {
  // pozicija roditelja čvora k
  int r = roditelj(k);
  // ako čvor k nije koren i ako je veći od roditelja
  if (k > 0 && hip[k] > hip[r]) {
     // razmenjujemo ga sa njegovim roditeljem
     swap(hip[k], hip[r]);
     // pomeramo roditelja navise
     pomeriNavise(hip, r);
  }
}

// ubacuje se element x u hip
void ubaci(vector<int>& hip, int x) {
  // element dodajemo na kraj
  hip.push_back(x);
  // pomeramo ga naviše dok se ne zadovolji uslov hipa
  pomeriNavise(hip, hip.size() - 1);
}

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  vector<int> hip;
  char c;
  while (cin >> c) {
    if (c == 'i') {
      int x;
      cin >> x >> ws;
      ubaci(hip, x);
    } else if (c == 'm') {
      cout << najveci(hip)  << '\n';
    } else if (c == 'r') {
      izbaciNajveci(hip);
    }
  }
  return 0;
}
```

Рекурзија у функцијама за померање елемената навише и наниже је репна, па се лако елиминише.

```
// element na poziciji k se pomera naniže, razmenom sa svojom decom
// sve dok se ne zadovolji uslov hipa
void pomeriNanize(vector<int>& hip, int k) {
  while (true) {
    // pozicija najvećeg čvora
    // (razmatrajući roditelja i njegovu decu)
    int najveci = k;
    int levo = levoDete(k), desno = desnoDete(k);
    if (levo < hip.size() && hip[levo] > hip[najveci])
      najveci = levo;
    if (desno < hip.size() && hip[desno] > hip[najveci])
      najveci = desno;
    // ako je roditelj najveći, uslov hipa je zadovoljen
    if (najveci == k)
      break;
    // menjamo roditelja i veće dete
    swap(hip[najveci], hip[k]);
    // obrađujemo veće dete
    k = najveci;
  }
}

// element na poziciji k se pomera naviše, razmenom sa svojim
// roditeljem, sve dok se ne zadovolji uslov hipa
void pomeriNavise(vector<int>& hip, int k) {
  int r = roditelj(k);
  while (k > 0 && hip[k] > hip[r]) {
     swap(hip[k], hip[r]);
     k = r;
     r = roditelj(k);
  }
}
```

Још један начин да имплементирамо померање наниже је да одредимо позицију већег од два детета и онда то дете упоредимо са родитељем.

```
// element na poziciji k se pomera naniže, razmenom sa svojom decom
// sve dok se ne zadovolji uslov hipa
void pomeriNanize(vector<int>& hip, int k) {
  int roditelj = k;
  // pretpostavljamo da je levo dete vece
  int veceDete = levoDete(k);
  // dok god čvor ima dece
  while (veceDete < hip.size()) {
    // poredimo da li je desno dete vece od levog
    int desno = veceDete + 1;
    if (desno < hip.size() && hip[desno] > hip[veceDete])
       veceDete = desno;
    // ako je roditelj veci ili jednak od oba deteta,
    // uslov hipa je zadovoljen
    if (hip[roditelj] >= hip[veceDete])
      break;
    // menjamo roditelja i vece dete
    swap(hip[veceDete], hip[roditelj]);
    // nastavljamo obradu od veceg deteta
    roditelj = veceDete;
    veceDete = levoDete(roditelj);
  }
```

```
}
```

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 бројева може бити реализовано директном имплементацијом алгоритма *heap sort* у којој је ред са приоритетом реализован помоћу ручно имплементираног хипа.

Интересанто, исти низ (или вектор) се може користити и за смештање полазног низа и за смештање хипа и за смештање сортираног низа (елементи хипа се могу сместити у почетни део низа, док се сортирани део може сместити у крајњи део низа). Тиме се штеди меморијски простор, међутим, потребно је мало прилагодити функције за рад са хипом. Наиме, функција за померање наниже поред низа или вектора у коме су смештени елементи, мора као параметар да прими и број елемената низа или вектора који представља хип (јер стварна димензија може бити и већа, ако се простор иза хипа користи за смештање елемената сортираног низа). На основу тог броја је лако установити који елементи низа припадају, а који не припадају хипу. Поређење са `hip.size()` се мора заменити поређењем са тим бројем.

Један начин да се од низа формира хип (у истом меморијском простору) је да се крене од празног хипа и да се један по један елемент убацује у хип. Овај начин се назива **формирање хипа наниже** или **Вилијамсов метод**.

```
// formira hip od elemenata niza a dužine n
// hip se smešta u istom memorijskom prostoru kao i niz
void formirajHip(int a[], int n) {
  // ubacujemo jedan po jedan element u hip i pomeramo ga naviše
  for (int i = 1; i < n; i++)
     pomeriNavise(a, i);
}
```

Инваријанта спољне петље (тј. индуктивна хипотеза) је то да елементи на позицијама $[0, i)$ чине макс-хип. Пошто знамо да операција померања елемента навише исправно умеће последњи елемент у постојећи макс-хип, лако је доказати да инваријанта остаје одржана. На крају петље је $i = n$, што значи да су сви елементи низа (елементи на позицијама $[0, n)$) сложени у макс-хип. Додатно, размене не мењају мултискуп елемената низа, па је мултискуп елемената у хипу једнак мултискупу елемената полазног низа.

Сложеност најгорег случаја формирања хипа једнака је $\Theta(n \log n)$. Наиме, сложеност операције уметања тј. померања елемента навише у хипу који има k елемената једнака је $O(\log k)$, па је укупна сложеност формирања хипа асимптотски једнака збиру свих логаритама од 1 до n, што је, како смо раније видели $\Theta(n \log n)$.

Други, ефикаснији начин формирања хипа назива се **формирање хипа навише** или **Флојдов метод**. Идеја је да се елементи оригиналног низа обилазе од позади и да се сваки елемент уметне у хип чији корен представља, тако што се спусти наниже кроз хип. Индуктивна хипотеза у овом приступу је то да су сви елементи из интервала (i, n) корени исправних макс-хипова. На пример, ако је дат низ

```
7 2 8 9 4 6 5 1 3
```

сви елементи осим прва два чине коренове исправних хипова. Заиста, у низу је смештено следеће дрво.

```
          7
      2       8
   9    4  6   5
  1   3
```

Елементи од 4, 6, 5, 1 и 3 немају наследнике и тривијално представљају исправне једночлане хипове. И у општем случају, сви елементи у другој половини низа представљају листове и за њих је ова инваријанта тривијално испуњена. Елемент 9 је већи од оба своја сина, па је корен исправног макс-хипа. Слично, елемент 8 је већи од оба своја сина, па је и он корен исправног макс-хипа.

Поставља се питање како проширити инваријанту. Елемент на позицији i не мора да буде већи од своје деце, али знамо да су оба његова детета корени исправних хипова. Стога је само потребно спустити елемент са врха на његово место, што је операција коју смо већ разматрали приликом операције брисања елемента из хипа.

Дакле, сваки елемент од оног на позицији $\lfloor \frac{n}{2} \rfloor$, па оназад до оног на позицији 0 се спушта наниже колико је потребно тако да хип коме је корен на тој позицији буде исправан.

У примеру, се елемент 2 мења са својим већим сином (то је елемент 9).

```
          7
       9     8
    2    4  6   5
  1   3
```

```
7 9 8 2 4 6 5 1 3
```

Након тога се мења са елементом 3 (он је сада већи син елемента 2).q

```
          7
       9     8
    3    4  6   5
  1   2
```

```
7 9 8 3 4 6 5 1 2
```

На крају, потребно је спустити још елемент 7. Он се мења са елементом 9 и пошто је након тога већи од своја оба детета, поступак се завршава.

```
          9
       7     8
    3    4  6   5
  1   2
```

```
9 7 8 3 4 6 5 1 2
```

```
// svi elementu u nizu a na pozicijama [0, i) zadovoljavaju uslov hipa
// pomera se element na poziciji i navise tako da nakon toga svi elementi
// na pozicijama [0, i] zadovoljavaju uslov hipa
void pomeri_gore(vector<int>& a, int i) {
  while (i > 0) {
    // roditelj cvora na poziciji i
    int roditelj = (i - 1) / 2;
    // ako element nije veci od svog roditelja, postupak je zavrsen
    if (a[i] <= a[roditelj])
      break;
    // inace razmenjujemo element sa svojim roditeljem
    swap(a[i], a[roditelj]);
    // i postupak nastavljamo od pozicije roditelja
    i = roditelj;
  }
}


// svi elementi u nizu a na pozicijama od 0 do n-1 zadovoljavaju uslov
// hipa osim eventualno elementa na poziciji i koji moze biti manji od
// svojih potomaka (ali je svakako manji ili jednak svom roditelju,
// ako roditelj postoji) - element se pomera nanize, tako da se na
// pozicijama 0 do n-1 dobije ispravan hip
void pomeri_dole(vector<int>& a, int i, int n) {
  while (true) {
    // pozicija sa kojom treba zameniti element na poziciji i
    // vrednost i govori da element ne treba menjati
    int menjam = i;
    // pozicija levog i desnog potomka cvora i
    int levi = 2 * i + 1;
    int desni = 2 * i + 2;
    // ako element na poziciji i ima levog potomka i ako je manji od njega
```

```
    // trebalo bi da se zameni sa levim
    if (levi < n && a[i] < a[levi])
      menjam = levi;
    // medjutim, ako postoji i desni potomak i ako je on jos manji
    // trebalo bi da se zameni sa njim
    if (desni < n && a[menjam] < a[desni])
      menjam = desni;

    // ako je menjam ostalo na vrednosti i znaci da nije manje ni od
    // jednog potomka i posao je zavrsen
    if (menjam == i)
      break;

    // u suprotnom menjamo element sa svojim potomkom
    swap(a[i], a[menjam]);
    // i nastavljamo postupak popravke od tog potomka
    i = menjam;
  }
}

// od niza pravi hip
void napravi_hip(vector<int>& a) {
  // za sve pozicije od 1 do a.size()-1
  for (int i = 1; i < a.size(); i++)
    // na pozicijama [0, i) je vec napravljen hip
    // element na poziciji i jedini ne mora da zadovoljava uslov
    // pa ga pomeramo navise tako da dobijemo hip na pozicijama [0, i]
    pomeri_gore(a, i);
}

// od niza u kojem se nalazi hip pravi sortirani niz
void sortiraj_hip(vector<int>& a) {
  // za sve pozicije od a.size()-1 unazad do 1
  for (int k = a.size() - 1; k > 0; k--) {
    // najveci element u hipu razmenjujemo sa elementom na poziciji n
    swap(a[0], a[k]);
    // element na vrhu ne mora biti na svom mestu u hipu koji ima k
    // elemenata pa ga pomeramo na dole tako da tih prvih k elemenata
    // cine ispravan hip
    pomeri_dole(a, 0, k);
  }
}

void heap_sort(vector<int>& a) {
  // Od niza pravi hip - binarno drvo u cijem se svakom cvoru nalazi
  // element koji je veci ili jednak od svojih potomaka. Drvo je u niz
  // smesteno po nivoima. Na primer:
  //        13
  //    8        10           13 8 10 4 2 5
  // 4     2   5
  napravi_hip(a);
  // od niza u kojem se nalazi hip pravi sortirani niz
  sortiraj_hip(a);

  // moze i uz pomoc biblioteckih funkcija
  // make_heap(a.begin(), a.end());
  // sort_heap(a.begin(), a.end());
}
```

Оценимо сложеност овог алгоритма. Прво, веома је згодно то што је једна половина елемената низа већ на свом месту. Спуштање елемената низ хип има сложеност $O(\log n)$, па је овде поново у питању неки збир логаритамских сложености и на први поглед може се помислити да ће и овде сложеност бити $\Theta(n \log n)$ – она ће свакако бити $(n \log n)$, али ћемо показати да ће бити нижа тј. $\Theta(n)$. Наиме, није сваки наредни логаритам у збиру за један већи од претходног, како је то био случај приликом померања елемената навише. Ако бисмо разматрали хип у коме су сви нивои потпуно попуњени, постојала би једна позиција са које би се елемент спуштао целом висином хипа, две позиције са којих би се елемент спуштао висином која је за један мања, четири елемента са којих је висина за 2 мања и тако даље. Укупан број корака за хип висине i би био $i + 2(i-1) + 4(i-2) + \ldots 2^{i-1} \cdot 1$ тј.

$$\sum_{k=0}^{i-1} 2^k (i-k).$$

До овог резултата можемо доћи и разматрањем следеће рекурентне једначине. Број корака померања наниже одговара збиру висина свих чворова у дрвету. Нека је $H(i)$ збир свих висина потпуног бинарног дрвета висине i. Тада је $H(i) = 2H(i-1) + i$, јер се потпуно бинарно дрво састоји од два потпуна бинарна дрвета висине $i-1$ и корена висине i. Важи и $H(0) = 0$. Размотајмо ову рекурентну једначину.

$$\begin{aligned}
H(i) &= 2H(i-1) + i \\
&= 2(2H(i-2) + (i-1)) + i = 4H(i-2) + 2(i-1) + i \\
&= 4(2H(i-3) + (i-2)) + 2(i-1) + i = 8H(i-3) + 4(i-2) + 2(i-1) + i \\
&= \ldots \\
&= 2^i H(0) + \sum_{k=0}^{i-1} 2^k (i-k) = \sum_{k=0}^{i-1} 2^k (i-k) \\
&= i \sum_{k=0}^{i-1} 2^k - \sum_{k=0}^{i-1} k 2^k = i(2^i - 1) - ((i-1)-1)2^{(i-1)+1} - 2 = 2^{i+1} - i - 2
\end{aligned}$$

Последње важи јер знамо да је

$$\sum_{k=0}^{n} 2^k = 2^{n+1} - 1, \quad \sum_{k=0}^{n} k 2^k = (n-1)2^{n+1} + 2.$$

Прво следи на основу формуле за збир геометријског реда $\sum_{k=0}^{n} x^k$, а друго њеним диференцирањем и множењем са x.

Постоји и лакши начин да се дође до истог резултата. Наиме, једначину $H(i) = 2H(i-1)+i$ можемо записати као $H(i) + i = 2(H(i-1) + (i-1)) + 2$ и онда увести смену $G(i) = H(i) + i$. Зато је $G(i) = 2G(i-1) + 2$ и $G(0) = 0$. Одмотавањем сада добијамо

$$\begin{aligned}
G(i) &= 2G(i-1) + 2 \\
&= 2(2G(i-2) + 2) + 2 = 4G(i-2) + 2 \cdot 2 + 2 \\
&= 4(2G(i-3) + 2) + 2 \cdot 2 + 2 = 8G(i-3) + 4 \cdot 2 + 2 \cdot 2 + 2 \\
&= \ldots \\
&= 2^i G(0) + 2 \sum_{k=0}^{i-1} 2^k = 2^{i+1} - 2
\end{aligned}$$

Зато је $H(i) = G(i) - i = 2^{i+1} - i - 2$.

Пошто је $G(i) + 2 = 2(G(i-1) + 2)$, сменом $F(i) = G(i) + 2$ би се добило да је $F(i) = 2F(i-1)$ и $F(0) = 2$, па би скоро директно следило да је $F(i) = 2^{i+1}$ и $G(i) = F(i) - 2 = 2^{i+1} - 2$.

Од свих ових комплексних извођења битан нам је само крајњи резултат, а то је да је $H(i) = 2^{i+1} - i - 2$. Пошто је за потпуно дрво број елемената низа $n = 2^i - 1$, важи да је $H(i) = 2(n+1) - i - 2 = 2n - i$ и важи да је сложеност конструкције навише $\Theta(n)$. Дакле, формирање хипа навише има асиптотски бољу сложеност него формирање хипа наниже!

Једно интуитивно објашњење ове чињенице лежи у томе да се код конструкције хипа наниже чак половина елемената (сви листови) може пењати целом висином дрвета, све до корена. Са друге стране, код конструкције навиже, корен је једини чвор који се може спустити целом висином, док се највећи број елемената дрвета јако мало помера.

Завршна фаза сортирања се спроводи вађењем елемената хипа која следи након формирања хипа. Инваријанта ове фазе биће да се у низу на позицијама $[0, i]$ налазе елементи исправно формираног хипа, а да се у делу (i, n) налазе сортирани елементи који су сви већи или једнаки од елемената који се тренутно налазе у хипу. На почетку је $i = n - 1$, па је инваријанта тривијално задовољена (елементи у интервалу $[0, n - 1]$ чине исправан хип, док је интервал $(n - 1, n)$ празан). У сваком кораку се највећи елемент хипа (елемент на позицији 0) избацује из хипа и додаје на почетак сортираног дела низа (на позицију i). Елемент са позиције i који је овом разменом завршио на врху хипа се помера наниже, све док се задовољи услов хипа. Вредност i се смањује за 1, чиме се исправно одржава граница између хипа и сортираног дела низа (хип је за један елемент краћи, а сортирани део низа је за један елемент дужи). Када се петља заврши тада је $i = 0$, па из инваријанте следи да је низ исправно сортиран (сви елементи на позицијама $(0, n)$ су сортирани и већи или једнаки елементу на позицији 0). Пошто се врше само размене елемената мултискуп елемената низа се не мења.

Види другачија решења овог задатка.

4.6.8 Хеширање и хеш-табеле

Хеширање (енгл. hashing) или директна организација података је један од основних начина за имплементацију неуређених скупова и неуређених мапа. Те структуре података подразумевају чување одређених слогова који се могу претраживати на основу неког њиховог кључног атрибута тј. кључа (код скупова се заправо чувају само кључеви, док се код мапа чувају слогови који поред кључа садрже и вредност придружену том кључу). На пример, у мапи тј. речнку кључеви могу бити ЈМБГ особа, а слогови уз ЈМБГ могу чувати и имена, презимена и још неке додатне информације о особама.

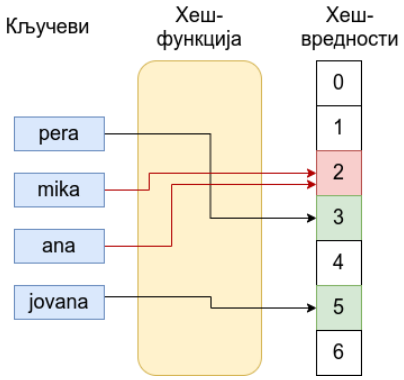
Хеширање омогућава ефикасно додавање нових слогова у структуру података (додавање елемента у скуп тј. придруживање вредности кључу у мапи), ефикасну претрага на основу кључа (проверу да ли елемент постоји у скупу тј. проналажење вредности придружене кључу у мапи), брисање слогова са датим кључем и измену вредности слогова на основу кључа (измену вредности придружене кључу у мапи). Са друге стране, за разлику од, на пример, уређених дрвета, хеширање не омогућава испис слогова у уређеном (сортираном) редоследу, нити проналажење кључа најближег датом.

Хеширање подразумева коришћење тзв. **хеш-функције** која сваки кључ пресликава у неки цео број (кажемо да је то **хеш-вредност** или **хеш-кôд** кључа). Сви подаци се чувају у **хеш-табели** – низу који на свакој позицији чува један слог (а у неким варијантама и више слогова). Хеш-вредност кључа одређује позицију у табели на којој слог са тим кључем треба да се нађе. Позиције у табели се некада називају и *слотови* (енгл. slots) или *кофе* (енгл. buckets), јер се некада и више слогова чува на једној позицији. Дакле, ако је димензија табеле n, хеш-фунцкија слика вредности из неког произвољно великог скупа кључева у релативно мали скуп позиција у табели $[0, n)$.

Најједноставнији случај хеширања је случај када се може успоставити бијекција између скупа кључева и допустивих позиција у низу. На пример, ако сваком малом слову енглеске абецеде желимо да доделимо вредност у низу од 26 елемената, хеш функција може бити `h(c) = c-'a'`, која елемент `a` слика у 0, `b` у 1 итд.

Ипак, много реалнија ситуација је да је број потенцијалних кључева доста већи од димензије табеле. Тада функција хеширања не може више бити инјективна тј. морају постојати **колизије** – ситуације у којима више кључева има исту хеш-вредност. Приликом хеширања потребно је да се реше два проблема:

- одабир хеш-функције којом се смањује вероватноћа настанка колизија
- разрешавање колизија када оне настану



Слика 4.3: Пресликавање кључева у хеш-вредности. Кључеви `mika` и `ana` су у колизији, јер се сликају у исту вредност 2

Наравно, на број колизија утиче и величина хеш-табеле, па се хеш-табеле најчешће с временом шире, када се табела у великој мери напуни и када број колизија постане превилики.

4.6.8.1 Избор хеш-функције

Хеш функција треба да:

- се израчунава брзо,
- минимизује број колизија.

Наиме, очигледно је да је пожељно да се хеш-функција израчунава у сложености $O(1)$, пожељно са малим константним фактором и да је недопустиво користити функције које се споро израчунавају, јер се израчунавање хеш-вредности врши приликом сваког уметања, сваког брисања, сваке промене и сваке претраге кључа.

Да би се смањио број колизија, пожељно је да хеш-функција кључеве слика равномерно (не и насумично) на све позиције у табели (да би се смањила могућност колизија проузрокованих нагомилавањем кључева у неким областима табеле). Многе једноставне хеш-функције не задовају својсто равномерности. На пример, ако би се за смештање ЈМБГ користио низ од 1000 елемената и ако би се хеширање ЈМБГ вршило тако што би се хеш-кôд сваког ЈМБГ одређивао на основу три цифре које одређују годину рођења (на пример, таква хеш функција би пресликала ЈМБГ 0305987810013 у позицију 987 у низу), Тада бисмо имали значајно нагомилавање на позицијама које почињу деветком тј. нулом и скоро потпуно неискоришћен остали простор у низу.

Ако се не зна унапред редослед обраде кључева, колизије је практично немогуће избећи. Наиме, до првих колизија долази веома брзо, чак иако се се резервише прилично велики меморијски простор за смештање елемената. Може се доказати да је, ако се елементи појављују у насумичном редоследу, ако се користи хеш-функција која равномерно распоређује елементе по табели и ако низ има n елемената, већ након убацивања око $\sqrt{2\ln(2)} \cdot \sqrt{n})$ тј. око $1,18 \cdot \sqrt{n}$ елемената, вероватноћа да убацивање новог елемента изазове колизију већа од 50%. Ово је у тесној вези са такозваним рођенданским парадоксом, који нам каже да је у скупу од само 23 особе вероватноћа да две особе славе рођендан истог дана већа него да све особе славе рођенда различитог дана (у скупу од 70 особа, та вероватноћа је већ преко 99,9%). На пример, ако резервишемо простор од милион (тј. 10^6 елемената), већ након убацивања тек нешто преко хиљаду и сто елемената (тј. $1,18 \cdot 10^3 = \sqrt{2\ln 2 \cdot 10^6}$), тј. већ када је попуњеност низа тек нешто изнад једног промила, имамо значајну вероватноћу да колизије крену да се јављају. Слободно можемо рећи да се колизије појављују и када је низ практично празан.

Дакле, колизије ће се сигурно стално јављати, међутим, ако је дужина низа n, а број различитих кључева m, код добрих хеш-функција вероватноћа да се на неком месту у низу појави више од m/n вредности треба да буде јако мала.

Хеширање се често изводи кроз два корака — први је да се хеш-функцијом кључу придружи произвољни природни број, а други корак је да се онда тај број преслика у неку позицију у низу, најчешће одређивањем остатка при дељењу са дужином низа n. Када је број елемената у низу одређени степен броја 2 своди на операцију издвајање одређеног броја крајњих битова у бинарном запису броја, што је веома ефикасно. Ипак, у том сценарију постоји неколико проблема.

Један потенцијални проблем у том сценарију представља то што у неким случајевима зависности између вредности кључева и дужине низа n могу створити неравномерну расподелу. На пример, ако је низ дужине 1024, и ако су из неког разлога све вредности кључева парни бројеви, половина позиција у низу ће бити неискоришћена (јер ће остатак парног броја при дељењу са 1024, одређен са последњих 10 битова броја, увек бити паран број). Тај проблем је знатно мањи ако је дужина низа прост број, али предност низова чија је дужина степен двојке је брже израчунавање хеш-функције (захваљујући простом издвајању крајњих битова).

Други проблем је то што вредност хеш-функције зависи само од крајњих битова кључа, па се информација коју кључ носи не користи у потпуности. Стога је у првој фази, пре издвајања последњих битова, потребно применити неку функцију чија ће вредност зависити од свих битова кључа. Чест начин је да се примени нека функција облика $h(x) = x \bmod p$, где је p неки прост број (такав да је $n << p << M$, где је n број елемената низа, а M укупан број различитих кључева). Још боље перформансе (али по цену споријег израчунавања) је коришћење функција $(ax + b) \bmod p$ — тиме се избегава проблем који настаје у случају када се функцијом $h(x) = x \bmod p$, хешира пуно кључева који имају исти остатак при дељењу са p.

У случајевима када је кључ сложен (низ вредности или ниска карактера) у првој фази се изврши неки облик агрегације више вредности у једну (та вредност је обично неки 32-битни или 64-битни цео број). То се обично врши применом неке операције попут сабирања или битовске ексклузивне дисјункције на појединачне елементе низа. Алтернативно, могуће је вредности низа схватити као коефицијенте полинома и затим израчунати вредност полинома (на пример, ниску карактера $c_0 \dots c_{k-1}$ у којој учествује само 26 слова енглеског алфабета је могуће превести у број израчунавањем вредности $c_0 26^{k-1} + \dots c_{k-1} 26^0$, занемарујући прекорачење).

За сваку хеш-функцију (чак и оних добрих тј. оних које кључеве равномерно распоређују по хеш-табели) може да се деси да постоји неки фамилија кључева која ће проузроковати велики број колизија и лоше перформансе. Решење је понекад *универзално хеширање* (енгл. universal hashing) које подразумева да се хеш-функција која ће се користити бира насумично из унапред задатог скупа добрих хеш функција (ово је слично насумичном избору пивота код брзог сортирања). Тада је математичко очекивање броја колизија мало, какви год да се кључеви јаве на улазу.

На крају, нагласимо да је теоријски, ако је скуп свих кључева који ће се у програму јавити, могуће одредити *савршену хеш-функцију* која ће бити инјективна и која ће сваки кључ пресликати тачно на једно месту у низу (чија величина одговара броју кључева који се јављају). Тада се колизије не би јављале, међутим, показује се да је сама конструкција такве савршене хеш-функције често рачунаски доста захтевна, па се не исплати.

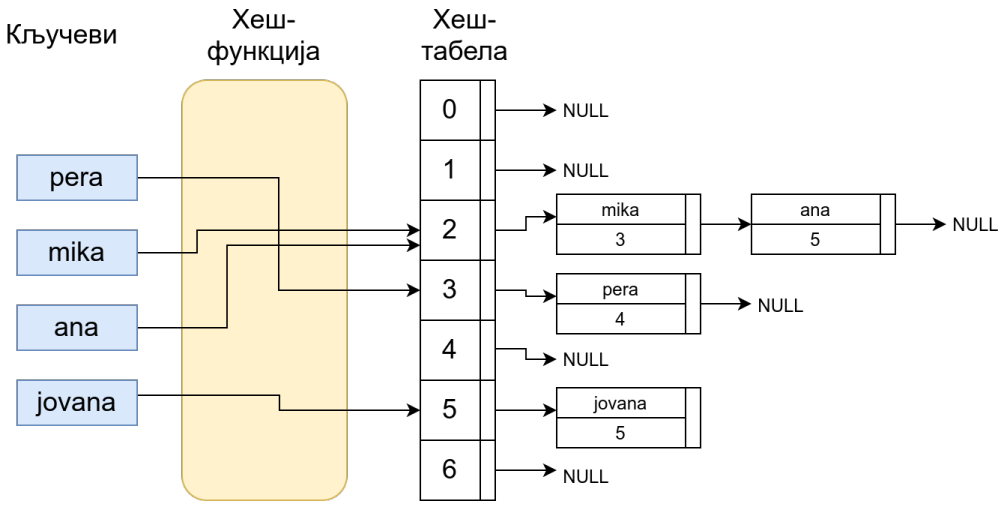
4.6.8.2 Разрешавање колизија

Постоји неколико начина да се разреше колизије. Два основна начина су:

- одвојено уланчавање
- отворено адресирање

Уланчавање

Најједноставнији начин је **одвојено уланчавање** (енгл. separate chaining), које се ређе назива и затворено адресирање или отворено хеширање. Оно подразумева да се на свакој позицији у низу формира посебна повезана листа у коју ће се смештати сви слогови чији се кључ хеш-функцијом пресликава на ту позицију. Приликом претраге слога са датим кључем врши се линеарна претрага листе свих елемената на позицији одређеној применом хеш-функције на тај кључ. Уметање новог слога се врши на почетак те листе (ако дупликати нису допуштени, претходно се линеарном претрагом проверава да ли елемент већ постоји у листи). Брисање елемента се врши из те листе (тако што се прво линеарном претрагом пронађе у листи).



Слика 4.4: Хеширање са уланчавањем

Пошто се у великом броју случајева на некој позицији налази само један елемент листе (или се не налази ни један), да би се смањио један ниво индирекције приликом приступа елементима, често се у низу уместо показивача чувају слогови (уз претпоставку да можемо да разликујемо попуњен и непопуњен слог).

Што листе постају дуже, то њихова линеарна претрага заузима све више и више времена. У најгорем случају (што често подразумева и да је избор хеш-функције направљен јако лоше) сви елементи могу да завршe у једној листи и сложеност свих операција за рад са хеш-табелом постаје линеарна. Ипак, у реалним имплементацијама се с времена на време врши повећање димензије низа (уз поновно распоређивање свих елемената, што подразумева поновно израчунавање свих хеш-вредности). Ова реалокација се врши у складу са геометријском стратегијом (на пример, димензија се повећава дупло приликом сваке реалокације), што гарантује да неће утицати на амортизовану сложеност операција. Реалокације се обично врше када фактор оптерећења (енгл. load factor) који се дефинише као укупан број елемената уписаних у табелу подељен димензијом табеле (бројем слотова тј. кофа) пређе одговарајућу границу (на пример, око 0,75). Други критеријум који може да се користи је дужина листа – када почну да се јављају дугачке листе, добро је повећати табелу. Наравно, уз лош одабир функције хеширања повећање димензије табеле не решава проблем и расипа се велика количина меморије (зато је јако важно да хеш-функција равномерно распоређује кључеве по табели).

Уместо повезаних листа, ланци се могу смештати и у низове који се динамички реалоцирају, па чак и у уређена балансирана бинарна дрвета. Ипак, показује се да ови компликованији приступи не значе обавезно и боље перформансе (јер се код кратких ланаца доста времена на одржавање дрвета – пре свега балансирање).

Отворено адресирање

Другчи често коришћен начин разрешавања колизија је **отворено адресирање** (енгл. open addressing), које се понекад назива и затворено хеширање. Када се користи отворено адресирање, за смештање података се користи само табела тј. само оригинални низ (за разлику од уланчавања где се подаци чувају у чворовима листе, ван оригиналног низа). У случају уписа података, када се деси колизија, место на које треба да се упише нови податак је већ заузето и податак ће тада бити уписан на неко друго место у низу. У зависности од тога како ћемо то друго место одабрати разликујемо:

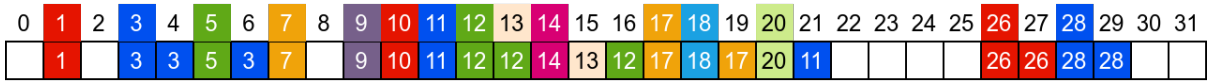
- линеарно попуњавање (енгл. linear probing)
- квадратно попуњавање (енгл. quadratic probing)
- двоструко хеширање (енгл. double hashing)

Линеарно пробање подразумева да се у случају да је позиција $h(x)$ већ заузета слог који одговара кључу x смешта на наредну позицију у низу (као наредна позиција последњој рачуна се прва), ако је она слободна. Ако није, гледа се њој наредна позиција и тако даље. Дакле, ако је позиција p заузета, проверава се позиција $(p+1) \bmod n$. Уместо вредности 1, може се користити и нека друга вредност k (мада се тиме не мења значајно сложеност). Дакле, позиција на коју ће се елемент x уписати је прва слободна позиција у низу који започиње на позицији $h(x)$ и где се сваки наредна позиција у низу добија од претходне применом функције $(p+k) \bmod n$. Низ позиција је, дакле, $h(x)$, $(h(x)+k) \bmod n$, $(h(x)+2k) \bmod n$ итд. Елемент тог низа који је i-ти по реду

(ако се броји од нуле) се налази на позицији $(h(x)+ik) \bmod n$. Када је вредност $k = 1$, тада се редом испитују узастопни елементи табеле, што има јако добре перформансе у односу на кеш-меморију савременог хардвера (много боље него било која друга техника разрешавања колизија) и зато се прилично често користи. Ако се користи вредност k различита од 1, пожељно је да је она узајамно проста са димензијом табеле n, да би низ $(h(x) + ik) \bmod n$ редом пролазио кроз све елементе табеле.

Поново се, дакле, формирају ланци елемената са истом хеш вредношћу, али су они овај пут не смештају у одвојеним листама, већ у самој табели. Додатно, ова схема узрокује и појаву секундарних колизија. Наиме, елемент x који се смести на позицију $(h(x) + k) \bmod n$ може заузети место неком елементу који има баш ту хеш-вредност, па се и он мора сместити на неку каснију позицију у низу (то зовемо *секундарним колизијама*).

Основна мана линеарног попуњавања је стварање дугачких ланаца и *груписање* тј. *нагомилавање* (енгл. clustering) слогова у одређеним деловима табеле, што може знатно деградирати перформансе. Наиме, када мало дужи ланци крену да се формирају, повећава се вероватноћа настанка секундарних колизија на позицијама тих ланаца, што даље продужује ланце и ствара се зачарани круг. На слици је приказан ефекат груписања. За сваки елемент у низу приказана је његова хеш-вредност. Види се постојање неколико дугачких ланаца и многи елементи су прилично удаљени од своје оригиналне позиције. Претрага за неким кључем који има хеш-вредност 13 захтева обилазак свих елемената, све до прве празне позиције (то је позиција 22). Претрага другог унетог кључа који има хеш-вредност 11 захтева обилазак свих елемената од позиције 11 до позиције 21.



Слика 4.5: Ефекат груписања елемената

Претрага се врши по истом принципу – претражује се тај низ позиција све док се или не наиђе на слог са траженим кључем или не наиђе на празну позицију (када закључујемо да кључ не постоји у табели).

Брисање је проблематичнија операција, јер се брисањем елемента може избрисати неки елемент из средине ланца, што може узроковати да неки каснији елементи у ланцу не буду пронађени (на, пример, ако се три кључа која имају хеш-вредност $h(x)$ упишу редом на позиције $h(x)$, $h(x) + 1$ и $h(x) + 2$, и ако се затим обрше слог са позиције $h(x)+1$, кључ који је на позицији $h(x)+2$ неће моћи да се пронађе у табели). Зато је приликом брисања потребно пребацити неки каснији елемент из ланца на позицију обрисаног слога (а затим на исти начин попунити и те позиције са којих су елементи пребачени). Једноставнија алтернатива је да се позиције са којих је обрисан слог означе као обрисане (а не као празне тј. слободне) и да се уметање и претрага модификују тако да узму у обзир и постојање таквих ћелија. Мана овог приступа је то што могу да троше велику количину меморије, као и то што ланци могу постати јако дугачки услед великог броја обрисаних ћелија.

Смањивање нагомилавања елемената може се постићи ако се уместо линеарног попуњавања $(h(x)+ik) \bmod n$ користи квадратно пробање $(h(x) + ik_1 + i^2k_2) \bmod m$ или двоструко хеширање, када се користе две независне хеш-функције h_1 и h_2 и када се наредна позиција одређује применом друге хеш функције $(h_1(x) + ih_2(x)) \bmod m$. Двоструко хеширање у потпуности спречава

Код отвореног адресирања је веома важно да се фактор оптерећења табеле (количник броја унетих слогова и димензије табеле) стално држи испод вредности 1. Када фактор оптерећења нарасте, врши се реалокација табеле уз поновно хеширање и уметање свих слогова. Да би се перформансе одржале, пожељно је да повећање димензије врши у складу са неком геометријском стратегијом (на пример, да се сваки пут табела повећа два пута).

Задатак: Мапа

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

Задатак се лако може решити коришћењем библиотечке имплементације асоцијативног низа (било уређене, било неуређене).

```
#include <iostream>
#include <unordered_map>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);

  unordered_map<string, int> map;
  char c;
  while (cin >> c) {
    if (c == 'w') {
      string k; int v;
      cin >> k >> v >> ws;
      map[k] = v;
    } else if (c == 'r') {
      string k;
      cin >> k >> ws;
      auto it = map.find(k);
      if (it != map.end())
        cout << it->second << '\n';
      else
        cout << "-" << '\n';
    }
  }
  return 0;
}
```

Задатак можемо решити и ручном имплементацијом хеш-табеле уз разрешавање колизија коришћењем одвојеног уланчавања. У језику C++ можемо употребити библиотечку имплементацију функције хеширања (то је функција `hash` доступна кроз заглавље `functional`, која је дефинисана за разне типове кључева, укључујући и тип `string`). Једноставности ради, табелу можемо имплементирати коришћењем колекције типа `vector`.

```
// cvor liste
struct cvor {
  string kljuc;
  int vrednost;
  cvor* sledeci;
};

// kreira se novi cvor sa datim atributima
cvor* napravi_cvor(const string& kljuc, int vrednost, cvor* sledeci) {
  cvor* novi = new cvor();
  // potrebno je proveriti da li je alokacija uspela
  novi->kljuc = kljuc;
  novi->vrednost = vrednost;
  novi->sledeci = sledeci;
  return novi;
}

// hes-tabela sa odvojenim ulancavanjem
vector<cvor*> tabela(1, nullptr);
// hes-funkcija
hash<string> h;
// broj kljuceva u mapi
int broj_kljuceva;

// inicijalizacija tabele na datu velicinu
void inicijalizuj_tabelu(int velicina) {
```

```
  tabela.resize(velicina, nullptr);
}

// uvecavanje tabele, da bi se smanjio broj kolizija i skratile liste
void povecaj_tabelu() {
  // nova tabela
  vector<cvor*> nova_tabela(2*tabela.size(), nullptr);
  // prepisujemo sve kljuceve i vrednosti iz originalne tabele, ponovo
  // hesirajuci kljuceve
  for (cvor* c : tabela) {
    for (cvor* cc = c; cc != nullptr; cc = cc->sledeci) {
      int p = h(cc->kljuc) % nova_tabela.size();
      nova_tabela[p] = napravi_cvor(cc->kljuc, cc->vrednost, nova_tabela[p]);
    }
  }
  // razmenjujemo novu i staru tabelu (swap radi brze nego dodela)
  tabela.swap(nova_tabela);
}

void upisi(const string& kljuc, int vrednost) {
  // pozicija u tabeli na koju treba da bude smesten kljuc
  int p = h(kljuc) % tabela.size();

  // proveravamo da li u listi kljuc vec postoji
  for (cvor* c = tabela[p]; c != nullptr; c = c->sledeci)
    if (kljuc == c->kljuc) {
      c->vrednost = vrednost;
      return;
    }

  // dodajemo novi cvor na pocetak liste
  tabela[p] = napravi_cvor(kljuc, vrednost, tabela[p]);
  broj_kljuceva++;

  // faktor opterecenja
  double opterecenje = (double)broj_kljuceva / (double)tabela.size();
  if (opterecenje >= 0.75)
    povecaj_tabelu();
}

// pretraga vrednosti pridruzene datom kljucu
// funkcija vraca informaciju o tome da li je vrednost pronadjena,
// a sama vrednost, ako postoji, se vraca preko povratnog parametra
bool procitaj(const string& kljuc, int& vrednost) {
  // pozicija u tabeli na kojoj treba da se nalazi kljuc
  int p = h(kljuc) % tabela.size();
  // trazimo kljuc u listi
  for (cvor* c = tabela[p]; c != nullptr; c = c->sledeci)
    if (kljuc == c->kljuc) {
      vrednost = c->vrednost;
      return true;
    }
  return false;
}

// brisanje svih cvorova u tabeli
void obrisi_tabelu() {
  for (cvor* c : tabela) {
```

```
    cvor* cc = c;
    while (cc != nullptr) {
      cvor* sledeci = cc->sledeci;
      delete cc;
      cc = sledeci;
    }
  }
}
```

Задатак можемо решити и ручном имплементацијом хеш-табеле уз разрешавање колизија коришћењем отвореног адресирања уз линеарно попуњавање.

```
// cvor u tabeli
struct cvor {
  string kljuc;
  int vrednost;
  bool pun;
};

// hes-tabela sa otvorenim adresiranjem
vector<cvor> tabela;
// hes-funkcija
hash<string> h;
// broj kljuceva u mapi
int broj_kljuceva;

// inicijalizacija tabele na datu velicinu
void inicijalizuj(int velicina) {
  tabela.resize(velicina);
  for (cvor& c : tabela)
    c.pun = false;
}

// pronalazi poziciju datog kljuca u datoj tabeli
int pronadji_poziciju(const string& kljuc, const vector<cvor>& tabela) {
  int p0 = h(kljuc) % tabela.size();
  int p = p0;
  int i = 0;
  while (tabela[p].pun && tabela[p].kljuc != kljuc)
    p = (p0 + i++) % tabela.size();
  return p;
}

// uvecavanje tabele, da bi se smanjio broj kolizija i skratile liste
void povecaj_tabelu() {
  // nova tabela
  vector<cvor> nova_tabela(2*tabela.size());
  // prepisujemo sve kljuceve i vrednosti iz originalne tabele, ponovo
  // hesirajuci kljuceve
  for (cvor& c : tabela) {
    if (c.pun) {
      int p = pronadji_poziciju(c.kljuc, nova_tabela);
      nova_tabela[p] = c;
    }
  }
  // razmenjujemo novu i staru tabelu (swap radi brze nego dodela)
  tabela.swap(nova_tabela);
}
```

```
void upisi(const string& kljuc, int vrednost) {
  // pozicija u tabeli na koju treba da bude smesten kljuc
  int p = pronadji_poziciju(kljuc, tabela);
  if (!tabela[p].pun)
    broj_kljuceva++;
  tabela[p].kljuc = kljuc;
  tabela[p].vrednost = vrednost;
  tabela[p].pun = true;

  // faktor opterecenja
  double opterecenje = (double)broj_kljuceva / (double)tabela.size();
  if (opterecenje >= 0.75)
    povecaj_tabelu();
}

// pretraga vrednosti pridruzene datom kljucu
// funkcija vraca informaciju o tome da li je vrednost pronadjena,
// a sama vrednost, ako postoji, se vraca preko povratnog parametra
bool procitaj(const string& kljuc, int& vrednost) {
  // pozicija u tabeli na kojoj treba da se nalazi kljuc
  int p = pronadji_poziciju(kljuc, tabela);
  if (tabela[p].pun) {
    vrednost = tabela[p].vrednost;
    return true;
  } else
    return false;
}
```