Algoritmi i strukture podataka
7. Čas — Implementacija struktura podataka

Sana Stojanović Đurđević

November 29, 2019

Pred vama je prateći materijal koji se može koristiti uz skriptu. Sastoji se od rešenja nekoliko zadataka koji su pređeni na vežbama i predavanjima.

1 Tipovi struktura podataka

sekvencijalni Svaki kontejner ima svoju specifičnu implementaciju:
`array` (statički niz),
`vector` (dinamički niz),
`list` (dvostruko povezana lista),
`forward_list` (jednostruko povezana lista),
`deque` (dinamički niz pokazivača na nizove fiksne veličine)

adaptori kontejnera Pružaju apstraktni interfejs iznad implementacije sekvencijalnog kontejnera, implementirajući funkcije tog interfejsa korišćenjem sekvencijalnog kontejnera za skladištenje podataka:
`stack` (`stack<int,vector<int>>` ili `stack<int,forward_list<int>>`),
`queue` (`queue<int, deque>>`),
`priority_queue` (`prioriti_queue<int,vector<int>,less<int>>`, struktura hip)

asocijativni kontejneri Implementirani pomoću samobalansirajućih uređenih binarnih drveta:
`set` (uređeno drvo čiji su svi elementi različiti),
`multiset` (uređeno drvo čiji elementi mogu da se ponavljaju, ili se čuva broj pojavljivanja svakog elementa),
`map` (uređeno binarno drvo formirano na osnovu različitih ključeva),
`multimap` (uređeno binarno drvo formirano na osnovu ključeva koji mogu da se ponavljaju)

neuređeni asocijativni kontejneri Implementirani pomoću heš tabela:
`unordered_set`,
`unordered_multiset`,
`unordered_map`,
`unordered_multimap`

2 Hip i hip-sort

2.1 Hip

Maks-hip (engl. Max-heap) je binarno drvo koje zadovoljava uslov da je svaki nivo osim eventualno poslednjeg potpuno popunjen, kao i da je vrednost svakog čvora veća ili jednaka od vrednosti u njegovoj deci.

Razmotrimo narednu implementaciju maks-hipa. Drvo smeštamo u niz. Koren (elemente na nivou 0) na poziciju 0, njegove sinove (elemente na nivou 1) na pozicije 1 i 2 , zatim njihove sinove (elemente na nivou 2) na pozicije 3, 4, 5 i 6 itd. Na primer, hip:

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

predstavljamo nizom: `11 9 7 4 3 5 6 2 3 1`

Ako se element nalazi na poziciji i, onda se njegovo levo dete nalazi na poziciji $2i + 1$, a desno dete na poziciji $2i + 2$, dok mu se roditelj nalazi na poziciji $\lfloor \frac{i-1}{2} \rfloor$.

```
#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;
}
```

Najveći element se uvek nalazi u korenu, tj. na poziciji 0. Složenost je $O(1)$.

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

Operacija uklanjanja najvećeg elementa iz maks-hipa. Poto se on nalazi u korenu, a drvo mora biti potpuno popunjeno (osim eventualno poslednjeg nivoa) na mesto izbačenog elementa se upisuje poslednji element iz hipa (najdešnji element poslednjeg nivoa).

Potrebno je proveriti da nije narušeno svojstvo hipa i da uporedimo vrednost u korenu sa vrednošću njegove dece (ako postoje). Ako je vrednost u korenu veća ili jednaka od tih vrednosti, onda koren zadovoljava uslov maks-hipa i procedura može da se završi (jer za sve ostale čvorove znamo da zadovoljavaju taj uslov, jer je izbacivanje krenulo od ispravnog hipa).

U suprotnom, menjamo vrednost u korenu sa većom od vrednosti njegove dece (tj. sa vrednošću njegovog deteta, ako ima samo jedno dete). Nakon toga koren zadovoljava uslov maks-hipa, i preostaje jedino da proverimo (i eventualno popravimo) ono poddrvo u čijem je korenu završila vrednost korena. Ovo je problem istog oblika, samo manje dimenzije u odnosu na polazni i lako se, dakle, rešava induktivno-rekurzivnom konstrukcijom.

Složenost je $O(\log n)$.

U narednom kodu se koristi funkcija `swap` za zamenu vrednosti elemenata u nizu. Napomenimo da je ranije ova funkcija bila u zaglavlju `<algorithm>`, a da je od standarda 2011 u zaglavlju `<utility>`.

`http://www.cplusplus.com/reference/algorithm/swap/`

```
// element na poziciji k se pomera nanize, razmenom sa svojom decom
// sve dok se ne zadovolji uslov hipa
void pomeriNanize(vector<int>& hip, int k) {
  // pozicija najveceg cvora (razmatrajuci roditelja i njegovu decu)
  int najveci = k;
  int levo = levoDete(k), desno = desnoDete(k);

  //ako se tako izracunate pozicije nalaze u hipu
  if (levo < hip.size() && hip[levo] > hip[najveci])
    najveci = levo;
  if (desno < hip.size() && hip[desno] > hip[najveci])
    najveci = desno;

  // ako roditelj nije najveci
  if (najveci != k) {
    // menjamo roditelja i vece dete
    swap(hip[najveci], hip[k]);
    // rekurzivno obradjujemo vece dete
    pomeriNanize(hip, najveci);
  }
}
```

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

Umetanje novog elementa. Element je najjednostavnije ubaciti na kraj niza. Ako je veći od svog roditelja, onda mu tu nije mesto i moguće je izvršiti njihovu razmenu. Nakon zamene, poddrvo T sa korenom u novom čvoru zadovoljava uslov hipa i celo drvo bez poddrveta T takođe zadovoljava uslov hipa. Svakim pomeranjem novog elementa naviše, ostatak drveta bez poddrveta T se smanjuje i problem se svodi na problem manje dimenzije i rešava se induktivno-rekurzivnom konstrukcijom. Jedna moguća implementacija je data u nastavku.

```
// element na poziciji k se pomera navise, razmenom sa svojim
// roditeljem, sve dok se ne zadovolji uslov hipa
void pomeriNavise(vector<int>& hip, int k) {
  // pozicija roditelja cvora k
  int r = roditelj(k);
  // ako cvor k nije koren i ako je veci 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 navise dok se ne zadovolji uslov hipa
  pomeriNavise(hip, hip.size() - 1);
}
```

```
void ispisi(vector<int> hip){
  for(int i=0; i<hip.size(); i++){
    cout << hip[i] << " ";
  }
  cout << endl;
}
```

```
int main(){

  vector<int> hip;

  cout << "Unesite elemente (0 kao oznaka za kraj):" << endl;
  int n;
  cin >> n;

  while(n){
    ubaci(hip, n);
    cout << "Sadrzaj hipa nakon ubacivanja: " << endl;
    ispisi(hip);
    cin >> n;
  }

  while(hip.size()!=0){
    izbaciNajveci(hip);
    cout << "Sadrzaj hipa nakon izbacivanja: " << endl;
    ispisi(hip);
  }

  return 0;
}
```

2.2 Hip sort

Sortiranje pomoću *heap*-a, predstavlja varijantu sortiranja selekcijom u kome se svi elementi niza postavljajuu maks-hip, a zatim se iz maks-hipa vadi jedan po jedan element i prebacuje na kraj niza.

Isti memorijski prostor (niz ili vektor) se može koristiti za smeštanje originalnog niza, za smeštanje hipa, i nakon toga za smeštanje sortiranog niza (čime se štedi memorijski prostor ali se moraju prilagoditi funkcije za rad sa hipom).

```
#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];
}
```

Funkcija za pomeranje naniže pored vektora u kome su smešteni elementi, mora kao parametar da primi i broj elemenata vektora koji predstavlja hip.

```
// element na poziciji k se pomera nanize, razmenom sa svojom decom
// sve dok se ne zadovolji uslov hipa
void pomeriNanize(vector<int>& hip, int hip_size, int k) {
  // pozicija najveceg cvora (razmatrajuci roditelja i njegovu decu)
  int najveci = k;
  int levo = levoDete(k), desno = desnoDete(k);

  //ako se tako izracunate pozicije nalaze u hipu
  if (levo < hip_size && hip[levo] > hip[najveci])
    najveci = levo;
  if (desno < hip_size && hip[desno] > hip[najveci])
    najveci = desno;

  // ako roditelj nije najveci
  if (najveci != k) {
    // menjamo roditelja i vece dete
    swap(hip[najveci], hip[k]);
    // rekurzivno obradjujemo vece dete
    pomeriNanize(hip, hip_size, najveci);
  }
}

void ispisi(vector<int> hip){
  for(int i=0; i<hip.size(); i++){
    cout << hip[i] << " ";
  }
  cout << endl;
}
```

Jedan način da se od niza formira hip (u istom memorijskom prostoru) je da sekrene od praznog hipa (sa desne strane) i da se jedan po jedan element ubacuje u hip. Formiranje hipa naviše ili Flojdov metod. Ideja je da se elementi originalnog niza obilaze od pozadi i da se svaki element umetne u hip čiji koren predstavlja, tako što se spusti naniže kroz hip. Induktivna hipoteza u ovom pristupu je to da su svi elementi iz intervala (i,n) koreni ispravnih maks-hipova.

```
// formira hip od elemenata niza a duzine n
// hip se smesta u istom memorijskom prostoru kao i niz
void formirajHip(vector<int>& a, int n){
  // sve elemente osim listova pomeramo nanize
  for (int i = n/2; i >= 0; i--)
    pomeriNanize(a, n, i);
}
```

Implementacija faze sortiranja vađenjem elemenata hipa (sledi nakon formiranja hipa). Invarijanta ove faze biće da se u nizu na pozicijama [0,i] nalaze elementi ispravno formiranog hipa, a da se u delu (i,n) nalaze sortirani elementi koji su svi veći ili jednaki od elemenata koji se trenutno nalaze u hipu. Na

početku je $i = n - 1$, pa je invarijanta trivijalno zadovoljena (elementi u intervalu $[0, n-1]$ čine ispravan hip, dok je interval $(n-1, n)$ prazan).U svakom koraku se najveći element hipa (element na poziciji 0) izbacuje iz hipa zamenom sa elementom na poziciji i (i dodaje na početak već sortiranog dela niza). Element sa pozicije i koji je ovom razmenom završio na vrhu hipa se pomera naniže kroz hip, sve dok se zadovolji uslov hipa.

```
// sortira se niz a duzine n
void hipSort(vector<int>& a, int n) {

  for(int i = n-1; i > 0; i--) {
    // najveci element vadimo iz hipa i ubacujemo ga na pocetak
    // sortiranog dela niza
    swap(a[0], a[i]);

    // pomeramo koren hipa nanize, popravljajuci hip
    // nakon izbacivanja najveceg, broj elemenata hipa jednak je i
    pomeriNanize(a, i, 0);
  }
}
```

```
int main(){

  vector<int> a;

  cout << "Unesite elemente (0 kao oznaka za kraj):" << endl;
  int n;
  cin >> n;

  while(n){
    a.push_back(n);
    cin >> n;
  }

  cout << "Uneli ste: " << endl;
  ispisi(a);

  // formiramo hip na osnovu elemenata niza
  formirajHip(a, a.size());
  cout << "Nakon formiranja hipa: " << endl;
  ispisi(a);

  // sortiramo niz na osnovu hipa
  hipSort(a, a.size());
  cout << "Nakon sortiranja: " << endl;
  ispisi(a);


  return 0;
}
```

3 Stek

3.1 Povezana lista

Stek možemo jednostavno implementirati pomoću jednostruko povezane liste. Smatramo da nam se vrh steka nalazi na početku liste. Tada su dodavanje na stek i uklanjanje sa steka zapravo dodavanje i uklanjanje sa početka liste. Potrebno je da čuvamo samo pokazivač na početak.

```
struct Cvor {
  int vr;
  Cvor *sled;
};

Cvor* noviCvor(int x) {
  Cvor *novi = new Cvor();
  novi->vr = x;
  novi->sled = nullptr;
  return novi;
}
```

```
struct Stek {
  Cvor *vrh = nullptr;
};

void dodaj(int x, Stek& s) {
  Cvor* novi = noviCvor(x);
  novi->sled = s.vrh;
  s.vrh = novi;
}

void skini(Stek& s) {
  Cvor* t = s.vrh;
  s.vrh = s.vrh->sled;
  delete t;
}

int vrh(Stek& s) {
  return s.vrh->vr;
}
```

Ispišimo pronalaženje prvog većeg prethodnika korišćenjem napisane strukture. Za to će nam biti potrebne još dve funkcije: određivanje da li je stek prazan i pošto radimo sa povezanom listom oslobađanje alocirane memorije.

```
bool prazan(Stek& s) {
  return s.vrh == nullptr;
}

void oslobodi(Stek& s) {
  while(!prazan(s))
    skini(s);
}
```

```
// za svaki broj ispisati njegovog prvog
// veceg prethodnika
int main() {
  int n, x;
  cin >> n;

  Stek s;
  for(int i = 0; i < n; i++) {
    cin >> x;

    while(!prazan(s) && vrh(s) <= x)
      skini(s);

    if(prazan(s))
      cout << "- ";
    else
      cout << vrh(s) << " ";

    dodaj(x, s);
  }

  oslobodi(s);
  return 0;
}
```

3.2 Niz

Drugi način da implementiramo stek je pomoću niza. Ako pretpostavimo da će se u svakom trenutku na steku naći najvise MAX_N elemenata, u nizu te dužine možemo čuvati elemente steka. Pored toga, pamtimo koliko trenutno elemenata imamo na steku. Vrh steka se nalazi na poziciji n. Dodavanje elemenata se onda vrši iza poslednjeg elementa steka.

```
struct Stek {
  int stek[MAX_N];
  int n = 0;
};
```

```
void dodaj(int x, Stek& s) {
  s.stek[s.n++] = x;
}

void skini(Stek& s) {
  s.n--;
}

int vrh(Stek& s) {
  return s.stek[s.n - 1];
}
```

Određivanje prvog većeg prethodnika se implementira identično kao u slučaju implementacije preko povezane liste (naravno, bez oslobađanja memorije).

```
bool prazan(Stek& s) {
  return s.n == 0;
}

// za svaki broj ispisati njegovog prvog
// veceg prethodnika
int main() {
  int n, x;
  cin >> n;

  Stek s;
  for(int i = 0; i < n; i++) {
    cin >> x;

    while(!prazan(s) && vrh(s) <= x)
      skini(s);

    if(prazan(s))
      cout << "- ";
    else
      cout << vrh(s) << " ";

    dodaj(x, s);
  }

  return 0;
}
```

4 Red

4.1 Povezana lista

Red, kao i stek, možemo implementirati pomoću jednostruko povezane liste. Potrebno je čuvati pokazivače na početak i kraj liste. Dodavanje novih elemenata vršimo na kraj liste, dok uklanjanje elemenata vršimo sa početka (zašto ne obrnuto?).

```
struct Red {
  Cvor *pocetak = nullptr, *kraj = nullptr;
};

void dodaj(int x, Red& q) {
  if(q.pocetak == nullptr)
    q.pocetak = q.kraj = noviCvor(x);
  else {
    q.kraj->sled = noviCvor(x);
    q.kraj = q.kraj->sled;
  }
}

void skini(Red& q) {
  Cvor* t = q.pocetak;
  q.pocetak = q.pocetak->sled;
  delete t;
  if(q.pocetak == nullptr)
    q.kraj = nullptr;
}

int prvi(Red& q) {
  return q.pocetak->vr;
}
```

Napišimo program koji pomoću napisane strukture ispisuje koliko postoji segmenata dužine k takvih da su im prvi i poslednji elementi jednaki.

```
void oslobodi(Red& q) {
  while(q.pocetak != nullptr)
    skini(q);
}
```

```
// ispisati koliko ima segmenata duzine k
// ciji su prvi i poslednji element jednaki
int main() {
  Red q;
  int n, k, x;
  cin >> n >> k;

  int brSegmenata = 0;
  for(int i = 0; i < n; i++) {
    cin >> x;
    dodaj(x, q);
    if(i >= k - 1) {
      if(x == prvi(q))
        brSegmenata++;
      skini(q);
    }
  }

  cout << brSegmenata << endl;
  oslobodi(q);
  return 0;
}
```

4.2 Niz

Ukoliko pretpostavimo da imamo informaciju da se u svakom trenutku u redu nalazi manje od MAX_N elemenata, možemo kreirati kružnu implementaciju reda i čuvati elemente reda u nizu te dužine. Pored samog niza čuvaćemo i dva brojača u kojima ćemo pamtiti poziciju početka, odnosno poziciju kraja reda.

Invarijanta koju održavamo je da se u delu niza ograničenog pozicijama $[pocetak, kraj)$ nalaze elementi reda (u slučaju da je $pocetak \leq kraj$), odnosno na pozicijama $[pocetak, MAX\_N) \cup [0, kraj)$ (u slučaju da je $kraj < pocetak$).

Dodavanje elemenata se vrši na kraj reda (na poziciju $kraj$), nakon čega se $kraj$ uvećava za 1. Kada $kraj$ dođe do vrednosti MAX_N, potrebno ga je vratiti na 0 kako bi se invarijanta održala u sledećem koraku.

Pristup prvom elementu reda se izvršava uz pomoć pozicije *pocetak* tako što se samo čita element koji se nalazi na toj poziciji.

Izbacivanje elementa sa početka reda se izvršava jednostavnim uvećavanjem vrednosti promenljive *pocetak*.

```
struct Red {
  int red[MAX_N];
  int pocetak = 0, kraj = 0;
};
```

```
void dodaj(int x, Red& q) {
  q.red[q.kraj] = x;
  q.kraj = (q.kraj + 1) % MAX_N;
}

void skini(Red& q) {
  q.pocetak = (q.pocetak + 1) % MAX_N;
}

int prvi(Red& q) {
  return q.red[q.pocetak];
}
```

Rešenje zadatka se ne menja u slučaju implementacije pomoću niza, osim što sada ne moramo da oslobođamo zauzetu memoriju.

```
// ispisati koliko ima segmenata duzine k
// ciji su prvi i poslednji element jednaki
int main() {
  Red q;
  int n, k, x;
  cin >> n >> k;

  int brSegmenata = 0;
  for(int i = 0; i < n; i++) {
    cin >> x;
    dodaj(x, q);
    if(i >= k - 1) {
      if(x == prvi(q))
        brSegmenata++;
      skini(q);
    }
  }

  cout << brSegmenata << endl;
  return 0;
}
```