Algoritmi i strukture podataka
3. Čas, Liste, stek, red

Aleksandar Veljković

2017/2018

1 Liste

Liste su linearne dinamičke strukture koje se sastoje od povezanih čvorova u kojima su smešteni podaci. Čvorovi su povezani pokazivačima i u zavisnosti od broja pokazivača liste mogu biti jednostruko ili dvostruko povezane. Kod jednostruko povezanih listi, čvorovi liste sadrže pokazivač na sledeći element u listi dok čvorovi dvostruko povezane liste sadrže i dodatni pokazivač na prethodni čvor liste. Ukoliko se obilaskom svih čvorova liste, prateći pokazivače izmedju čvorova, može ponovo doći do polaznog čvora, lista je ciklična. Kod acikličnih listi, poslednji element ima pokazivač na null vrednost (**NULL**, **nullptr**), čime se oznaăva kraj liste.

Čvor dvostruko povezane liste koji sadrži celobrojnu vrednost se jezikom C++ može definisati kao struktura:

Primer:

```
struct Cvor {
    int vrednost;
    Cvor *sledeci;
    Cvor *prethodni;
}
```

Novi čvor ovakve liste se kreira korišćenjem operatora new, koji alocira memoriju za novi čvor.

Primer:

```
Cvor *novi_cvor = new Cvor;
```

Inicijalno se pokazivači čvora mogu postaviti na vrednost **nullptr**. Za korišćenje **nullptr** potrebno je navesti standard iz 2011. godine.

Primer:

```
struct Cvor {
    int vrednost;
    Cvor *sledeci = nullptr;
    Cvor *prethodni = nullptr;
}
```

Kao dodatnu pogodnost, moguće je definisati i funkciju konstruktor (iako to nije praksa kada se radi o strukturama), kojom se, pri kreiranju novog čvora, vrednost može inicijalizovati na zadatu vrednost:

Primer:

```
struct Cvor {
    Cvor(int n) : vrednost(n){};
    int vrednost;
    Cvor *sledeci = nullptr;
    Cvor *prethodni = nullptr;
};
```

Kreiranje novog čvora sa inicijalizovanom vrednošću (u primeru sa vrednošću 5) vrši se naredbom:

Primer:

```
Cvor *novi_cvor = new Cvor(5)
```

Ova struktura čvora se dalje može upotrebiti kao sastavni deo omotač strukture dvostruko povezane liste, čime se apstrakcija podiže na još viši nivo. Osnovna struktura dvostruko povezane liste može čuvati pokazivač na prvi i poslednji čvor, funkcije za dodavanje i uklanjanje čvorova liste sa početka i kraja liste, kao i funkciju za oslobadjanje liste. Ovakva apstrakcije se obično vrši pomoću klasa umesto struktura (iako su po prirodi vrlo srodni pojmovi), ali se primer implementacije ovakve strukture korišćenjem strukture može pronaći u pratećim materijalima za ovaj čas.

1.1 Struktura list

U okviru standardne C++ biblioteke dostupna je struktura dvostruko povezane liste (**list**). Da bi se ova struktura mogla koristiti u kodu, potrebno je uključiti zaglavlje biblioteke **<list>**.

Primer:

```
#include <list>
```

Struktura **list** može sadržati elemente bilo kog tipa pa se zato u okviru zagrada <i >navodi tip elemenata koji će se čuvati u listi.

Primer:

```
list<int> lista; // Lista celih brojeva
```

Dostupne funkcije za dodavanje elemenata na pocetak ili kraj liste i uklanjanje elemenata sa pocetka ili kraja liste. Vremenska složenost ovih funkcija je $O(1)$. Takodje, dostupne su i funkcije za uklanjanje svih elemenata u listi, uklanjanje elemenata sa odrenjenim vrednostima i mnoge druge.

Primer:

```
list<int> lista;

lista.push_back(1); // Dodavanje elementa na kraj liste
// [1]

lista.push_back(2);
// [1, 2]

lista.push_front(3); // Dodavanje elementa na pocetak liste
// [3, 1, 2]

lista.push_front(4);
// [4, 3, 1, 2]

lista.pop_back(); // Uklanjanje elementa sa kraja liste
// [4, 3, 1]

lista.pop_front(); // Uklanjanje elementa sa pocetka liste
// [3, 1]

cout << lista.front() << endl; // Funkcija front vraca vrednost prvog elementa liste
// -> 3

cout << liste.back() << endl; // Funkcija back vraca vrednost poslednjeg elementa liste
// -> 1
```

Elementi **list** strukture se obilaze iteratorima. Funkcije **next()** i **prev()** vraćaju iteratore na sledeći, odnosno prethodni, element liste.

Primer:

```
for(auto it = lista.begin(); it != lista.end(); it++)
{
    if(it != lista.begin())
    {
        auto prethodni = prev(it);
        cout << *prethodni << ","; // Ispis prethodnog elementa
    }

    cout << *it; // Ispis tekuceg elementa

    auto sledeci = next(it);

    if(sledeci != lista.end())
        cout << "," << *sledeci;

    cout << " ";
}
```

Funkcijom **erase()** uklanja se čvor liste na koji pokazuje iterator prosledjen kao argument funkcije dok se funkcijom **remove()** uklanjaju svi čvorovi liste koji sadrže vrednost prosledjenu kao argument funkcije. Funkciji **erase()** se mogu proslediti i dva iteratora pa funkcija uklanja sve čvorove iz liste koji se nalaze na pozicijama od prvog do drugog iteratora. Za umetanje cvora u listu () koristi se funkcija **insert** kojoj se kao argumenti navode iterator na poziciju na kojoj će se naći novi cvor i vrednost koja će se čuvati u čvoru.

Primer:

```
list<int> lista = {1, 2, 3, 2, 4, 2, 5};

auto it = next(next(lista.begin()));

/* [1, 2, 3, 2, 4, 2, 5]
 *        ^
 *        it
 */

lista.erase(it);
// [1, 2, 2, 4, 2, 5]

lista.remove(2);
// [1, 4, 5]

list<int> lista2 = {1, 2, 3, 4, 5, 6, 7};

auto it_levi = next(lista2.begin());
auto it_desni = prev(prev(lista2.end()));

/* [1, 2, 3, 4, 5, 6, 7]
 *     ^           ^
 *   it_levi    it_desni
 */

lista2.erase(it_levi, it_desni);
// [1, 6, 7]
```

```
lista2.insert(it_desni, 10);
// [1, 10, 6, 7]
```

Funkcijom **merge** se dve sortirane liste (druga lista je navedena kao argument funkcije) objedinjavaju u novu, sortiranu listu koja sadrži elemente obe liste. Rezultujuća lista je smeštena u promenljivu sa prvom listom dok druga lista postaje prazna.

Primer:

```
list<int> lista1 = {1, 4, 6};
list<int> lista2 = {2, 3, 5};

lista1.merge(lista2);
// lista1: [1, 2, 3, 4, 5, 6]
// lista2: []
```

2 Stek

Stek je LIFO (eng *Last-In-First-Out*) struktura, kod koje se elementi dodaju i uklanjaju sa istog kraja. Osnove operacije sa stekom su dodavanje (**PUSH**) i uklanjanje (**POP**) elemenata sa vrha steka. Još jedna često podržana operacija je provera koji element se trenutno nalazi na vrhu steka (**TOP**). Potrebno je da se sve operacije na steku izvršavaju u vremenu **O(1)**.

Stek se može implementirati pomoću niza ili liste. Obe implementacije imaju svoje prednosti i mane. Implementacija korišćenjem niza omogućava dobru iskorišćenost memorije, s obzirom da se čuvaju samo elementi steka bez dodatnih pokazivača. Sa druge strane, mana ovakve implementacije je potreba za realokacijom niza u slučaju popunjenosti steka, koja je u najgorem slučaju linearne vremenske složenosti po broju elemenata na steku.

Implementacija pomoću liste garantuje složenost svih operacija u vremenu O(1), ali zahteva dodatnu memoriju za smeštanje pokazivača izmedju čvorova liste. U pratećim materijalima za ovaj čas može se pronaći implementacija steka korišćenjem liste.

2.1 Struktura stack

Standardna C++ biblioteka sadrži strukturu **stack** koja predstavlja implementaciju strukture stek. Struktura se nalazi u okviru biblioteke <stack>.

Primer:

```
#include <stack>
```

Pri deklaraciji nove sturkture **stack** navodi se tip elemenata koji će se smeštati na stek. Funkcija **push()** dodaje novi element na vrh steka dok funkcija **pop()** skida element sa vrha steka. Funkcija **pop()** ne vraća vrednost elementa skinutog sa vrha steka već je za proveru vrednosti sa vrha steka potrebno iskoristiti funkciju **top()**. Funkcija **empty()** proverava da li je stek prazan.

Primer:

```
stack<int> s; // Stek koji cuva elemente tipa int

s.push(1);
s.push(2);
s.push(3);

while(s.empty() == false)
{
  cout << s.top() << " "; // Ispis elementa sa vrha steka
  s.pop(); // Uklanjanje elementa sa vrha steka
```

```
}

// ispis: 3 2 1
```

3 Red

Struktura red je FIFO (eng First-In-First-Out) struktura kod koje se elementi dodaju sa jednog kraja a uklanjaju sa drugog. Red na šalteru je primer strukture red iz svakodnevnog života, osoba koja prva stigne prva će završiti posao na šalteru, dok će osobe koje su kasnije došle dolaziti na kraj reda i čekati da svi ispred njih završe posao (pod uslovom da niko ne prolazi sa strane samo da pita nešto). Osnovne operacije nad redom su dodavanje (**ENQUEUE**) elementa na kraj reda i uklanjanje (**DEQUEUE**) elementa sa početka reda. Obe operacije su vremenske složenosti **O(1)**.

Red se, kao i stek, može implementirati pomoću niza ili liste. Važe iste prednosti i mane za obe implementacije uz dodatni problem sa nizom. Potrebno je obratiti pažnju pri uklanjanju elemenata iz reda jer se elementi uklanjaju sa početka niza, pa je moguća situacija da je u redu samo jedan element (na kraju niza), ali da nema sledećeg praznog mesta u nizu za druge elemente koji pristižu. Ovakav problem se može rešiti kružnom implementacijom reda kod koje se nakon popunjavanja poslednje pozicije u nizu popunjavaju prazna polja sa početka niza uz evidenciju o poziciji novog početka reda. U pratećim materijalima za ovaj čas može se pronaći implementacija reda korišćenjem liste.

3.1 Struktura queue

U standardnoj biblioteci jezika C++ nalazi se stuktura **queue** koja predstavlja strukturu red. Struktura queue se nalazi u okviru biblioteke <queue>.

Primer:

```
#include <queue>
```

Pri deklaraciji nove sturkture **queue** navodi se tip elemenata koji će se smeštati u red. Funkcija **push()** dodaje novi element na kraj reda dok funkcija **pop()** skida element sa početka. Funkcija **pop()**, kao i kod strukture **stack** ne vraća vrednost elementa uklonjenog sa početka reda već je za proveru vrednosti sa početka reda potrebno iskoristiti funkciju **front()**. Funkcija **empty()** proverava da li je red prazan.

Primer:

```
queue<int> r; // Stek koji cuva elemente tipa int

r.push(1);
r.push(2);
r.push(3);

while(r.empty() == false)
{
  cout << s.front() << " "; // Ispis elementa sa pocetka reda
  s.pop(); // Uklanjanje elementa sa kraja reda
}

// ispis: 1 2 3
```

4 Zadaci

1. Napisati funkciju vremenske i prostorne složenosti $O(1)$ koja za zadati pokazivač na čvor jednostruko povezane liste uklanja čvor na koji pokazivač p pokazuje, osim u slučaju kada pokazivač pokazuje na poslednji čvor liste.

2. Napisati funkciju koja iz dvostruko povezane liste uklanja sve elemente sa parnim vrednostima.

3. Napisati funkciju koja proverava da li u listi postoji ciklus.

4. Implementirati strukturu stek pomocu liste.

5. Implementirati strukturu red pomocu liste.

6. Elementima dvostruko povezane liste predstavljene su cifre nenegativnog celog broja. Napisati funkciju kojom se modifikuje lista tako da sadrzi cifre obrnutog ulaznog broja.

7. Napisati funkciju koja proverava da li su zagrade u niski, prosledjenoj kao argument, ispravno uparene.

8. Implementirati strukturu stek na koji će se smeštati celi brojevi i implementirati funkcije koje u vremenu O(1) obavljaju operacije dodavanja elementa na stek, skidanja elementa sa vrha steka i vraćanja vrednosti minimalnog elementa na steku.

9. Implementirati strukturu trocifrenog brojača pomoću tri strukture red. Implementirati funkciju koja inicijalizuje brojač na 000, funkciju koja inkrementira brojač za 1 i funkciju koja ispisuje trenutnu vrednost brojača.

10. Implementirati strukturu stek i funkcije za rad sa stekom pomocu dva reda.

11. Implementirati strukturu red i funkcije za rad sa redom pomocu dva steka.

12. Napisati funkciju koja izračunava vrednost izraza napisanog u infiksnoj notaciji, pri čemu su svi operandi i operacije grupisani u zagrade. Smatrati da izraz obuhvata cele pozitivne brojeve i operacije +, -, * i /. Npr. za ulaz **((1+3)*3)** funkcija vraća vrednost **12**.

13. Napisati funkciju koja transformiše izraz zapisan u infiksnoj notaciji u isti izraz zapisan u postfiksnoj notaciji i ispisuje ga na izlazni tok. Smatrati da su svi operandi i operacije grupisani u zagrade kao i da su operandi celi pozitivni brojevi a operacije +, -, * i /. Npr. za ulaz **((1+2)*3)** funkcija ispisuje **1 2 + 3 *** dok za ulaz **(3*(1+2))** funkcije ispisuje **3 1 2 + ***.

14. Napisati funkciju koja izračunava vrednost izraza napisanog u postfiksnoj notaciji. Smatrati da su operandi celi pozitivni brojevi a operacije +, -, * i /. Npr. za ulaz **1 2 + 3 *** funkcija vraća vrednost **9**.