Bibliotečke strukture podataka

Da bi algoritmi mogli efikasno funkcionisati potrebno je da podaci koji se obrađuju budu organizovani na način koji omogućava da im se efikasno pristupa i da se efikasno modifikuju i ažuriraju. Za to se koriste *strukture podataka* (ovaj pojam ne treba mešati sa strukturama u programskom jeziku C).

Strukture podataka su obično kolekcije podataka koje omogućavaju neke karakteristične operacije. U zavisnosti od implementacije samih struktura izvođenje tih operacija može biti manje ili više efikasno. Prilikom dizajna algoritma često se fokusiramo upravo na operacije koje nam određena struktura nudi, podrazumevajući da će sama struktura podataka biti implementirana na što efikasniji način. Sa stanovišta dizajna algoritma nam je potrebno samo da znamo koje operacije podrazumeva određena struktura podataka i da imamo neku procenu (obično asimptotsku ili amortizovanu) izvođenja te operacije. U tom slučaju kažemo da algoritme konstruišemo u odnosu na *apstraktne strukture podataka*, čime se prilikom dizajna i implementacije algoritama oslobađamo potrebe da brinemo o detaljima implementacije same strukture podataka. Najznačajnije strukture podataka su implementirane u bibliotekama savremenih programskih jezika (C++, Java, C#, Python i mnogi drugi imaju veoma bogate biblioteke struktura podataka). U ovom poglavlju ćemo se baviti upotrebom i primenom gotovih struktura podataka u implementaciji nekih interesantnih algoritama. Sa druge strane, poznavanje implementacije samih struktura podataka može biti značajno za njihovo duboko razumevanje, ali i za mogućnost njihovog prilagođavanja upotrebi u nekim specifičnim algoritmima, kao i za mogućnost definisanja novih struktura podataka, koje nisu podržane bibliotekom programskog jezika koji se koristi. Stoga ćemo se u narednom poglavlju baviti načinima implementacije struktura podataka.

Skalarni tipovi

Pojedinačni podaci se čuvaju u promenljivama osnovih, skalarnih tipova podataka (to su obično celobrojni i realni tipovi podataka, karakterski tip, logički tip, pa čak i niske, ako se gledaju kao atomički podaci, bez analize njihovih delova).

Parovi, torke, slogovi

Najjednostavnije kolekcije podataka sadrže samo mali broj (2, 3, 4, ...) podataka, koji mogu biti i različitih tipova. U savremenim programskim jezicima postoje dva načina da se podaci upakuju na taj način. Jedno su uređeni parovi (engl. pair) tj. uređene torke (engl. tuple) gde se svakom pojedinačnom podatku pristupa na osnovu pozicije. Drugo su slogovi tj. strukture (engl. record, struct) u gde se svakom pojedinačnom podatku pristupa na osnovu naziva. Prilikom korišćenja slogova potrebno je eksplicitno definisati novi tip podataka, pa njihovo korišćenje zahteva malo više programiranja, nego korišćenje parova i torki koje obično teče ad hoc.

Parovi

U jeziku C++ se tip para navodi kao `pair<T1, T2>` gde su `T1` i `T2` tipovi prve i druge komponente. Pristup prvom elementu para vrši se poljem `first`, a drugom poljem `second`. Par se od vrednosti dobija funkcijom `make_pair`. Funkcijom `tie`moguće je par razložiti na komponente (npr. `tie(x, y) = p` uzrokuje da promenljiva `x` sadrži prvu, a `y` drugu komponentu para). Prikažimo sada upotrebu parova u jeziku C++ za modelovanje tačaka u ravni.

Problem: Date su koordinate tri tačke trougla. Odredi koordinate njegovog središta.

```
#include <iostream>

#include <utility>

using namespace std;
```

```
int main() {

   pair<double, double> A, B, C;

   cin >> A.first >> A.second;

   cin >> B.first >> B.second;

   cin >> C.first >> C.second;

   pair<double, double> T;

   T.first = (A.first + B.first + C.first) / 3.0;

   T.second = (A.first + B.first + C.first) / 3.0;

   cout << T.first << " " << t.second << endl;

   return 0;

}
```

U jeziku Python, upotreba parova je još jednostavnija.

```
(Ax, Ay) = (int(input()), int(input()))

(Bx, By) = (int(input()), int(input()))

(Cx, Cy) = (int(input()), int(input()))

(Tx, Ty) = ((Ax + Bx + Cx) / 3.0, (Ay + By + Cy) / 3.0)

print(Tx, Ty)
```

Torke

U jeziku C++ se tip torke navodi kao `tuple<T0, T1, ...>` gde su `Ti` redom tipovi komponenata torke. Pristup i-tom elementu gorke vrši se funkcijom `get<i>`. Torka se od pojedinačnih vrednosti gradi funkcijom `make_tuple`. Funkcijom `tie` moguće je torku razložiti na komponente. Pod pretpostavkom da se pojedinačne komponente mogu porediti i torke se mogu porediti relacijskim operatorima (i tada se koristi leksikografski poredak). Prikažimo sada upotrebu torki u jeziku C++ za modelovanje poređenja datuma (koje je suštinski leksikografsko).

Problem: Sa ulaza se učitava niz datuma. Odredi najkasniji od njih.

```
#include <iostream>

#include <utility>

using namespace std;
```

```
tuple<int, int, int> ucitaj_datum() {
   int m, d, g;
   cin >> m >> d >> g;
   return make_tuple(g, m, d);
}

void ispisi_datum(const tuple<int, int, int>& datum) {
   cout << get<2>(datum) << " " <<
           get<1>(datum) << " " <<
           get<0>(datum) << endl;
}

int main() {
   int n;
   cin >> n;

   tuple<int, int, int> max_datum = ucitaj_datum();
   for (int i = 1; i < n; i++) {
      tuple<int, int, int> datum = ucitaj_datum();
      if (datum > max_datum)
         max_datum = datum;
   }
   ispisi_datum(max_datum);
   return 0;
}
```

Korišćenje torki u jeziku Python je još jednostavnije.

```python
def ucitaj_datum():
    (d, m, g) = (int(input()), int(input()), int(input()))
    return (g, d, m)


def ispisi_datum(datum):
    (g, d, m) = datum
    print(d, m, g)


n = int(input())
max_datum = ucitaj_datum()
for i in range(1, n):
    datum = ucitaj_datum()
    if datum > max_datum:
       max_datum = datum
ispisi_datum(max_datum)
```

Slogovi (strukture)

Nema velike razlike između definisanja i korišćenja struktura u jeziku C i jeziku C++. Ilustrujmo to na primer razlomaka.

Problem: Definisati strukturu za predstavljanje razlomaka, funkciju za sabiranje razlomaka i glavni program koji ih testira.

```cpp
#include <iostream>
using namespace std;


struct razlomak {
   int brojilac, imenilac;
};


int nzd(int a, int b) {
```

```
    while (b != 0) {
        int tmp = a % b;
        a = b;
        b = tmp;
    }
    return a;
}

void skrati(razlomak& r) {
   int n = nzd(r.brojilac, r.imenilac);
   r.brojilac /= n;
   r.imenilac /= n;
}

razlomak saberi(const razlomak& r1, const razlomak& r2) {
   razlomak zbir;
   zbir.brojilac = r1.brojilac * r2.imenilac + r2.brojilac * r1.imenilac;
   zbir.imenilac = r1.imenilac * r2.imenilac;
   skrati(zbir);
   return zbir;
}

int main() {
   razlomak r1{3, 5}, r2{2, 3};
   razlomak zbir = saberi(r1, r2);
   cout << zbir.brojilac << "/" << zbir.imenilac << endl;
   return 0;
```

```
}
```

Nizovi (statički, dinamički)

Nizovi (u raznim oblicima) predstavljaju praktično osnovne kolekcije podataka. Osnovna karakteristika nizova je to da omogućavaju efikasan pristup (po pravilu u složenosti $O(1)$ ili bar u amortizovanoj složenosti $O(1)$) elementu niza na osnovu njegove pozicije (kažemo i indeksa). Pristup van granica niza obično prouzrokuje grešku u programu (u nekim jezicima poput Java, C# ili Python se podiže izuzetak, a u nekim poput C ili C++ ponašanje nije definisano). U zavisnosti od toga da li je broj elemenata niza poznat (i ograničen) u trenutku pisanja i prevođenja programa ili se određuje i menja tokom izvršavanja programa nizove delimo na statičke i dinamičke. Osnovna operacija u radu sa nizovima je pristup i-tom elementu. U velikom broju savremenih programskih jezika (npr. C++, Java, C#, Python) brojanje elemenata počinje od nule.

Statički nizovi

Rad sa statičkim nizovima u jeziku C++ je veoma sličan radu sa statičkim nizovima u C-u. Ilustrujmo to jednim jednostavnim primerom.

Problem: Učitava se najviše 100 brojeva. Izračunaj element koji najmanje odstupa od proseka.

```
#include <iostream>

#include <cmath>

using namespace std;



int main() {

   // učitavamo broj elemenata i zatim sve elemente u niz

   int a[100];

   int n;

   cin >> n;

   for (int i = 0; i < n; i++)

      cin >> a[i];



   // izračunavamo prosek

   int zbir = 0;

   for (int i = 0; i < n; i++)

      zbir += a[i];

   double prosek = (double) zbir / (double) n;
```

```
    // određujemo i ispisujemo element koji najmanje
    // odstupa pod proseka
    int min = 0;
    for (int i = 1; i < n; i++)
       if (abs(a[i] - prosek) < abs(a[min] - prosek))
          min = i;
    cout << a[i] << endl;

    return 0;
}
```

Naglasimo da je u ovom zadatku bilo neophodno elemente prilikom učitavanja memorisati, jer su nam za rešenje zadatka potrebna dva prolaska kroz podatke (prvi u kom se određuje prosek i drugi u kom se određuje minimalno odstupanje od proseka). Da se tražilo samo određivanje proseka ili prosečno odstupanje od minimuma, niz nije bilo neophodo koristiti (razmisli kako bi se to uradilo u memorijskoj složenosti $O(1)$).

Dinamički nizovi

Kada nije unapred poznata dimenzija niza ili kada se očekuje da će se ona prilično menjati pri raznim pokretanjima programa poželjno je korišćenje dinamičkih nizova. Dinamički nizovi obično podržavaju rezervaciju prostora za određeni broj elemenata (a taj broj kasnije može biti menjan). Takođe, dopuštena operacija je dodavanje na kraj niza. U slučajevima kada u nizu nema dovoljno prostora za smeštanje elemenata vrši se automatska realokacija.

U jeziku C++ dinamički nizovi su podržani klasom `vector<T>` gde je `T` tip podataka koji se smeštaju u vektor. Prilikom konstrukcije moguće je navesti inicijalni broj elemenata. Metod `size` se može koristiti za određivanje veličine niza (tj. broja njegovih elemenata). Metodom `resize` vrši se efektivna promena veličine niza. Rezervaciju prostora (ali bez promene veličine niza) moguće je postići pozivom metode `reserve`. Metodom `push_back` vrši se dodavanje elementa na kraj niza.

Prikažimo upotrebu vektora na sledećoj varijaciji prethodnog zadatka.

Problem: Učitavaju se brojevi do kraja standardnog ulaza. Koji element najmanje odstupa od proseka?

```
#include <iostream>

#include <vector>

#include <string>
```

```
int main() {

   // učitavamo sve elemente u vektor

   vector<int> a;

   string linija;

   while (cin >> linija)

      a.push_back(stoi(linija));



   // izračunavamo prosek

   int zbir = 0;

   for (int i = 0; i < n; i++)

      zbir += a[i];

   double prosek = (double) zbir / (double) n;



   // određujemo i ispisujemo element koji najmanje

   // odstupa pod proseka

   int min = 0;

   for (int i = 1; i < a.size(); i++)

      if (abs(a[i] - prosek) < abs(a[min] - prosek))

         min = i;

   cout << a[min] << endl;



   return 0;

}
```

Iako se osnovne kolekcije u jeziku Python nazivaju liste, one zapravo predstavljaju dinamičke nizove (pre svega zbog efikasnog indeksnog pristupa i zbog mogućnosti dodavanja elemenata metodom `append`).

```
import sys
```

```
# učitavamo elemente u listu

a = []

for linija in sys.stdin:

    a.append(int(linija))



prosek = sum(a) / len(a)



min = 0;

for i in range(1, n):

    if abs(a[i] - prosek) < abs(a[min] - prosek):

        min = i



print(a[min])
```

U jeziku Java dinamički nizovi su podržani putem klase `ArrayList`, a u jeziku C# putem klase `List` (pri čemu u ovim jezicima ne postoje klasični statički nizovi, već se obični nizovi takođe alociraju na hipu, tokom izvršavanja programa, jedino što im nije moguće dinamički menjati dimenziju).

Višedimenzionalni nizovi, matrice (statički, dinamički)

U nekim slučajevima su nam potrebni višedimenzionalni nizovi. U nekim jezicima oni su podržani direktno, a nekada se realizuju kao nizovi nizova. U dvodimenzionom slučaju jedno od osnovnih pitanja je da li su u pitanju matrice kod kojih sve vrste imaju isti broj elemenata ili su u pitanju nizovi vrsta kod kojih svaka vrsta može imati različit broj elemenata. I višedimenzioni nizovi mogu biti statički i dinamički.

U jeziku C++ statički višedimenzionalni nizovi se koriste na veoma sličan način kao jednodimenzionalni (i veoma slično kao u C-u). Ilustrujmo to jednim jednostavnim primerom.

Problem: Napisati program koji učitava i pamti matricu dimenzije 5×5 i zatim izračunava njen trag (zbir dijagonalnih elemenata).

```
#include <iostream>

using namespace std;
```

```
int main() {

  // učitavamo matricu

  int A[5][5];

  for (int v = 0; v < 5; v++)

    for (int k = 0; k < 5; k++)

      cin >> A[v][k];



  // izračunavamo i ispisujemo trag

  int trag = 0;

  for (int i = 0; i < 5; i++)

    trag += A[i][i];

  cout << trag << endl;



  return 0;

}
```

Ukoliko dimenzija nije unapred poznata, najlakše nam je da za smeštanje matrice upotrebimo vektor vektora.

Problem: Napisati program koji učitava i pamti matricu dimenzije $n\times n$ i zatim izračunava njen trag.

```
#include <iostream>

#include <vector>

using namespace std;



int main() {

  // učitavamo elemente matrice

  int n;

  cin >> n;

  vector<vector<int>> A(n);
```

```
  for (int v = 0; v < n; v++) {
     A[v].resize(n);
     for (int k = 0; k < n; k++)
       cin >> A[v][k];
   }

   // izračunavamo i ispisujemo trag
   int trag = 0;
   for (int i = 0; i < n; i++)
     trag += A[i][i];
   cout << trag << endl;

   return 0;
}
```

U slučaju vektora vektora, vrste mogu imati i različit broj kolona.

Zadatak: Napisati program koji učitava i pamti ocene nekoliko studenata i za svakog izračunava prosečnu ocenu.

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  // učitavamo tabelu ocena svih učenika
  int brojUcenika;
  cin >> brojUcenika;
  vector<vector<int>> ocene(brojUcenika);
  for (int u = 0; u < brojUcenika; u++) {
```

```
    int brojOcena;
    cin >> brojOcena;
    ocene[u].resize(brojOcena);
    for (int o = 0; o < n; o++)
      cin >> ocene[u][o];
  }


  // izračunavamo i ispisujemo sve proseke ocena
  for (int u = 0; u < brojUcenika; u++) {
    int zbir = 0;
    int brojOcena = ocene[u].size();
    for (int o = 0; o < brojOcena; o++)
      zbir += ocene[u][o];
    cout << (double) zbir / (double) brojOcena << endl;
  }


  return 0;
}
```

Stekovi

Stek predstavlja kolekciju podataka u koju se podaci dodaju po LIFO principu - element se može dodati i skinuti samo na vrha steka.

U jeziku C++ stek se realizuje klasom `stack<T>` gde `T` predstavlja tip elemenata na steku. Podržane su sledeće metode:

- `push` - postavlja dati element na vrh steka
- `pop` - skida element sa vrha steka
- `top` - očitava element na vrhu steka (pod pretpostavkom da stek nije prazan)
- `empty` - proverava da li je stek prazan
- `size` - vraća broj elemenata na steku

Prikažimo upotrebu steka kroz nekoliko interestantnih primera.

Istorija veb-pregledača

Problem: Pregledač veba pamti istoriju posećenih sajtova i korisnik ima mogućnost da se vraća unatrag na sajtove koje je ranije posetio. Napisati program koji simulira istoriju pregledača tako što učitavaju adrese posećenih sajtova (svaka u posebnom redu), a kada se učita red u kome piše `back` pregledač se vraća na poslednju posećenu stranicu.

Rešenje je krajnje jednostavno. Spisak posećenih sajtova čuvamo na steku.
Naredbom `back` uklanjamo i ispisujemo sajt sa vrha steka (uz proveru da stek nije prazan).

```
#include <iostream>

#include <string>

#include <stack>



int main() {

   stack<string> istorija;

   string linija;

   while (getline(cin, linija)) {

      if (linija == "back")

         if (!istorija.empty()) {

            cout << istorija.top() << endl;

            istorija.pop();

         } else {

            cout << "-" << endl;

         }

      else

         istorija.push(linija);

   }

   return 0;

}
```

U jeziku Python stek možemo simulirati pomoću najobičnije liste. Metodom `append` dodajemo element na kraj liste, a metodom`pop` skidamo element sa kraja.

```
import sys



istorija = []

for linija in sys.stdin:

    linija = linija.strip()

    if linija == "back":

        if istorija:

            print(istorija.pop())

        else:

            print("-")

    else:

        istorija.append(linija)
```

Problem: Napisati program koji sve učitane linije sa standardnog ulaza ispisuje u obratnom redosledu.

Jedno od mogućih rešenja je da se sve učitane linije smeste na stek, a da se zatim ispišu uzimajući jednu po jednu sa steka. Pošto stek funkcioniše po LIFO principu, redosled će biti obrnut (najkasnije dodata linija biće prva skinuta i ispisana, dok će prva postavljena linija biti skinuta i ispisana poslednja).

```
#include <iostream>

#include <string>

#include <stack>



int main() {

   stack<string> s;

   string linija;

   while (cin >> linija)

      s.push(linija);

   while (!s.empty()) {

      cout << s.top() << endl;
```

```
        s.pop();

    }

    return 0;

}
```

Uparenost zagrada

Problem: Napisati program koji proverava da li su zagrade u infiksno zapisanom izrazu korektno uparene. Moguće je pojavljivanje malih, srednjih i velikih zagrada (tzv. običnih, uglastih i vitičastih).

Kada bismo radili samo sa jednom vrstom zagrada, dovoljno bi bilo samo održavati broj trenutno otvorenih zagrada. Pošto imamo više vrsta zagrada održavamo stek na kome pamtimo sve do sada otvorene, a ne zatvorene zagrade. Kada naiđemo na otvorenu zagradu, stavljamo je na stek. Kada naiđemo na zatvorenu zagradu, proveravamo da li se na vrhu steka nalazi njoj odgovarajuća otvorena zagrada i skidamo je sa steka (ako je stek prazan ili ako se na vrhu nalazi neodgovarajuća otvorena zagrada, konstatujemo grešku). Na kraju proveravamo da li se stek ispraznio.

```
#include <iostream>

#include <stack>

using namespace std;



bool uparena(char oz, char zz) {

    return oz == '(' && zz == ')' ||

           oz == '[' && zz == ']' ||

           oz == '{' && zz == '}';

}



bool otvorena(char c) {

    return c == '(' || c == '[' || c == '{';

}



bool zatvorena(char c) {

    return c == ')' || c == ']' || c == '}';

}
```

```
int main() {
   string izraz;
   cin >> izraz;
   stack<char> zagrade;
   bool OK = true;
   for (char c : izraz) {
      if (otvorena(c))
        zagrade.push(c);
      else if (zatvorena(c)) {
         if (zagrade.empty() || !uparena(zagrade.top(), c))
            OK = false;
         zagrade.pop();
      }
   }
   if (!zagrade.empty())
      OK = false;

   cout << (OK ? "tacno" : "netacno") << endl;
   return 0;
}
```