Elektrotehnički fakultet Sarajevo
Odsjek za računarstvo i informatiku
Predmet: Algoritmi i strukture podataka

Vježba 2

Cilj vježbe 2 je upoznavanje sa strukturom podataka lista, pri čemu će studenti za pripremu uraditi implementaciju liste, a zatim na vježbi modifikovati implementaciju i primijeniti je za rješavanje nekoliko problema

Zadatak 1.

Diskusija o različitim strategijama implementacije Liste:

- Koliko memorije zauzima NizLista a koliko JednostrukaLista, u slučajevima:
  - prazan niz
  - 10 elemenata
  - 10000 elemenata
- Na koji način je osigurano da je veličina NizListe neograničena?
  - Rješenje 1: svaki poziv metode dodaj*() vrši realokaciju. Vrijeme izvršenja ove metode je konstantno, ali su performanse očito slabe.
  - Rješenje 2: metoda prosiri() poziva se kada se postojeća alokacija popuni. Koliko proširiti svaki put? Npr. ako se stavi da je default veličina alokacije 1000000 elemenata, ova metoda se možda neće nikada pozvati. Performanse su odlične, ali je zauzeće memorije loše.
- Da li je u klasi NizLista korišten pokazivač na tekući element?
  - Dovoljno je čuvati indeks tekućeg elementa. S druge strane, u JednostrukojListi je pokazivač puno praktičniji.
- Da li u klasi JednostrukaLista postoji atribut za broj elemenata?
  - Iako ovaj atribut tehnički nije nužan, prebrojavanje elemenata u listi je O(n) ako se svaki put računa.
- Koje metode mjenjaju stanje liste?
- Kakva je kompleksnost metoda koje su implementirane, može li se postići bolja?

Napraviti novi main() program u kojem će se demonstrirati prednosti i nedostatke obje razvijene klase.

Koristiti metode mjerenja vremena izvršenja koje su obrađene na prošlom tutorijalu. Napraviti program koji koristi *polimorfizam* tako što omogućuje korisniku da izabere da li će se koristiti NizLista ili JednostrukaLista, a zatim bez dupliciranja koda poziva pojedine metode u petlji i ispisuje vrijeme izvršenja. Obratiti pažnju da je pojedinačno izvršenje metoda npr. operatora [] prekratko za mjerenje pa je potrebno napraviti petlje koje izvršavaju tu operaciju veliki broj puta.

Zadatak 2.

Napraviti funkcije koja primaju konstantnu referencu na NizLista i JednostrukaLista a vraćaju njihov najveći član:

```
template<typename Tip>
Tip dajMaksimum(const NizLista<Tip>& n) {
template<typename Tip>
Tip dajMaksimum(const JednostrukaLista<Tip>& n) {
```

Pozovati ove funkcije iz programa. Pri realizaciji ovog zadatka nije dozvoljeno uvoditi nove metode odnosno mijenjati način rada postojećih metoda.

Diskusija.

Ako se koristi operator [], funkcija neće dati optimalne performanse za Listu.

S druge strane, ako se koriste metode **sljedeci()** (ili **prethodni()**) mijenja se vrijednost atributa, što nije dozvoljeno kod konstantnih referenci. Sličan problem je i sa NizListom.

Jedino ispravno rješenje ovog problema je uvesti novu klasu: *Iterator*. Iterator klasa je prijateljska klasa NizLista i JednostrukaLista i kao takva može čitati vrijednosti atributa, a u sebi sadrži odgovarajuće pokazivače i metode za prelazak na sljedeći/prethodni element.

Najlakši način da se napravi ova klasa je da u atributima postoje pokazivači i na niz i na listu, a zatim se naprave dva konstruktora koji primaju NizLista odnosno JednostrukaLista i na osnovu toga koji od ta dva konstruktora je pozvan, postave se vrijednosti ostalih atributa.

Npr. Iterator klasa može sadržavati sljedeće atribute:

- const NizLista<Tip> *niz;
- const JednostrukaLista<Tip> *lista;
- int trenutniNiz;
- typename JednostrukaLista<Tip>::Cvor *trenutniLista;

a zatim sljedeće metode:

- konstruktor, prima konstantnu referencu na NizLista, postavlja pokazivač niz na adresu parametra, sve ostale atribute na 0;
- konstruktor koji prima konstantnu referencu na JednostrukaLista, postavlja pokazivač lista na adresu parametra, trenutniLista na prvi član liste, sve ostale atribute na 0;
- metoda trenutni() najprije provjerava da li je u pitanju niz ili lista:
-

```
if (niz!=0) // niz je
pa u tom slučaju obavlja operacije za niz:
return niz->_niz[trenutniNiz];
U suprotnom slučaju obavlja operacije za listu:
return trenutniLista->element;
```

Ne zaboraviti provjeru da li je niz/lista prazan.

- na sličan način implementirati i metode prethodni(), sljedeci(), pocetak() i kraj().

Da bi se proglasila klasa Iterator za prijateljsku klasu klase NizLista mora se u klasu NizLista dodati:

```
friend class Iterator<Tip>;
```

no ako klasa Iterator nije definisana prije NizListe, ovdje će kompajler prijaviti grešku. Ako se Iterator definiše prije NizListe dobiti će se greška u liniji gdje se definiše atribut Niz. Način da se zaobiđe ovaj koka-jaje problem je da predefiniše klasa Iterator prije niza:

```
template <typename Tip>
class Iterator; // Predefinisana klasa iterator
template <typename Tip>
class Niz {
      … // Definicija klase Niz
      friend class Iterator<Tip>;
};
template <typename Tip>
class Lista {
      … // Definicija klase Lista
      friend class Iterator<Tip>;
};
template <typename Tip>
class Iterator {
      … // Sada normalno implementirati Iterator
};
```

Zadatak 3.

Implementaciju JednostrukaLista proširiti metodom *void izbaciSvakiNTi(int n)* koja će iz liste izbaciti svaki n-ti element po redu. Primjer: Neka je dat niz 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Za uneseno n=3 niz treba da bude 1, 2, 4, 5, 7, 8, 10. U slučaju brisanja trenutnog čvora, trenutni čvor postaje predhodni čvor, a ako nema predhodnog trenutni će postati sljedeći čvor. Ako je nemoguće obaviti izbacivanje baciti izuzetak.

Napomena: Paziti na specijalne slučajeve