# STRUKTURE PODATAKA I ALGORITMI

ORGANIZACIJA PROJEKTA

- C++ projekt dijeli na više datoteka:
  .h datoteke (zaglavlja) sadržavaju sučelja
  .cpp datoteke sadržavaju implementacije

- Kompajliraju se samo .cpp datoteke i to na sljedeći način:
  Na mjesto #include se iskopira sadržaj iz .h datoteke
  - primjer: #include "pravokutnik.h"
  Svaki .cpp se kompajlira neovisno o ostalima
  Na kraju se svi .cpp linkaju

- Zbog kopiranja .h datoteka u .cpp datoteke mogu nastati problemi ako se ista .h datoteka više puta uključi u isti .cpp
  Problem se rješava korištenjem include guard-ova u .h datotekama
  #ifndef_MOJAKLASA_H_ // Ako ovaj .h nije uključen.
  #define _MOJAKLASA_H_ //Označi da je sad uključen.
  class MojaKlasa{…};
  #endif

KONSTRUKTORI I DESTRUKTORI

- Konstruktori
  Svaka struktura i klasa može imati konstruktor
  - Metoda koja se poziva prilikom izrade objekta
  - Koristi se za postavljanje početnog stanja objekta
  - Naziv jednak nazivu strukture/klase, ali bez povratne vrijednosti (nema čak niti void)
  - Konstruktor bez parametara se naziva *defaultni* konstruktor (ne smijemo pisati zagrade)

  Primjeri pozivanja konstruktora

```
pravokutnik p1;
pravokutnik p2(4);
pravokutnik p3(6, 8);
pravokutnik p4[5];
pravokutnik* p5 = new pravokutnik(4, 6);
```

- type def služi za definiranje aliasa za postojeće tipove podataka
  - Sintaksa: **typedef postojeći_naziv novi_naziv;**
  - Primjer: **typedef unsigned int uint;**
  • Sad nam je sve jedno hoćemo li pisati uint ili unsigned int
- Destruktor

  Metoda koja se poziva prilikom uništenja objekta
  - Završetkom funkcije
  - Pozivom delete

Naziv jednak nazivu strukture/klase s tildom ispred, nema povratne vrijednosti, nema parametara

```
~pravokutnik() {
cout<< "Destruktor je pozvan" << endl;
}
```

KLASA STRINGSTREAM

- Klasa stringstream predstavlja ulazno/izlazni tijek prema međuspremniku znakova (u memoriji):

| | |
|---|---|
| - Uključimo zaglavlje: | #include<sstream> |
| - Napravimo varijablu: | stringstream sstr; |
| - U nju upisujemo kao u cout: | sstr << "XY" << 22 << endl; |
| - Iz nje čitamo kao iz cin: | sstr >> broj; |
| - Iz nje uzmemo string: | strings = sstr.str(); |
| - Čistimo njegov sadržaj: | sstr.str(""); |
| | sstr.clear(); |

BINARNE DATOTEKE

- Konstruktori klasa ifstream i ofstream kao drugi parametar mogu primiti konstantu **ios_base::binary**
  Na taj način otvaramo datoteku na binarni način
  :: je operator dosega (engl. *scope operator*)

  ifstream dat1("datoteka1.bin", **ios_base::binary**);
  ofstream dat2("datoteka2.bin", **ios_base::binary**);

- Pisanje u binarne datoteke
  Za pisanje koristimo metodu write() na varijabli tipa ofstream

  **dat.write(*pokazivač_na_početak*, *veličina*);**

  Pri tome su parametri sljedeći:
  - **pokazivač_na_početak** je tipa char*
    Predstavlja adresu komada memorije iz kojeg želimo zapisati podatke u datoteku
  - ***veličina*** je tipa int
    Predstavlja broj bajtova (tj. veličinu komada memorije) koji želimo zapisati u datoteku

  Kako ćemo dobiti char*za pisanje u datoteku:
  Svodi se na pitanje kako ćemo dobiti početak nekog dijela memorije

  Za sve primitivne tipove (npr. za int):
  Uzmemo adresu operatorom & (time dobijemo pokazivač na int, a ne na char):
  **&varijabla**
  Dobivenu adresu pretvorimo (engl. *cast*) u char* posebnim operatorom:
  **(char*)(&*varijabla*)**

Primjeri pisanja u datoteku:

```
Int broj1= 85;
dat1.write((char*)(&broj1), sizeof(broj1));

double broj2= 15.5;
dat1.write((char*)(&broj2), sizeof(broj2));
```

- Čitanje iz binarne datoteke

Čitanje podrazumijeva kopiranje iz datoteke na disku u **unaprijed pripremljeni** prostor u radnoj memoriji

Za čitanje nam treba pokazivač na **prvi bajt** te **ukupan broj bajtova** koje želimo pročitati

Za čitanje koristimo metodu read() na varijabli tipa ifstream

**dat.read(pokazivač_na_početak, veličina);**

Pri tome su parametri sljedeći:
- **pokazivač_na_početak** je tipa char*
  Predstavlja adresu komada memorije u koji želimo smjestiti podatke iz datoteke
- **veličina** je tipa int
  Predstavlja broj bajtova koji želimo pročitati iz datoteke
  Mora biti jednak veličini pripremljene memorije

Kako ćemo dobiti char*za čitanje iz datoteke:
**(char*)(&varijabla)**

Primjer čitanja iz datoteke:

```
// Učitavanje int-a.
int n1;
dat.read((char*)(&n1), sizeof(n1));

// Učitavanje double-a.
double n2;
dat.read((char*)(&n2), sizeof(n2));
```

- Pisanje stringa u binarne datoteke

Kod zapisivanja u datoteku ćemo uvijek prvo zapisati broj znakova u stringu, a tek onda sâm string

String ima metodu c_str() koja vraća char*, tako da je njegovo zapisivanje jednostavno:

```
// String kojeg želimo zapisati
string s = "Miro";
// Duljina u znakovima (dakle i u bajtovima)
int duljina = s.length();
```

```
// Upišemo prvo broj znakova u stringu
dat1.write((char*)(&duljina), sizeof(duljina));
// Upišemo string.
dat1.write(s.c_str(), duljina);

string imena[5];
for (int i = 0; i < 5; i++) {
int broj_znakova = imena[i].size();
// Broj znakova u stringu
dat.write((char*)(&broj_znakova), sizeof(broj_znakova));
// String
dat.write(imena[i].c_str(), imena[i].size());
```

- Kod čitanja iz datoteke ćemo prvo pročitati broj znakova u stringu, a tek onda sâm string i to u dva koraka:

1. Učitamo bajtove stringa iz datoteke u dinamičku memoriju
2. Iz te dinamičke memorije konstruiramo string na stogu

```
// Pročitamo duljinu stringa
int n;
dat.read((char*)(&n), sizeof(n));
// Alociramo memoriju na hrpi
char* pchar = new char[n];
// Pročitamo znakove iz datoteke u tu memoriju.
dat.read(pchar, n);
// Konstruiramo string na stogu
string s = string(pchar, n);
// Oslobodimo memoriju
delete[] pchar;
```

## MJERENJE TRAJANJA IZVRŠAVANJA KODA

- Da bismo precizno mjerili koliko traje izvršavanje nekoga dijela kôda, možemo koristiti priložene datoteke:

```
high_res_timer.h
high_res_timer.cpp
```

- Mjerenje radimo na sljedeći način:

```
hr_timer timer;
start_hr_timer(timer);
// Kôd čije trajanje želimo mjeriti.
stop_hr_timer(timer);
get_elapsed_time_microsec(timer);
```

GENERIRANJE SLUČAJNIH BROJEVA

- Da bismo generirali slučajne brojeve, prvo moramo uključiti zaglavlje ctime:
  #include <ctime>

- Nakon toga, moramo postaviti inicijalnu vrijednost(seed):
  srand(time(nullptr));

- Sad možemo generirati slučajni broj između min i max:
  int slucajni_broj = rand() % (max–min + 1)+ min;

LISTA

- **Lista** je konačni niz (od nula ili više) podataka **istog** tipa
- Podaci koji čine listu nazivaju se njeni **elementi**
- n je **duljina** liste
  - Uvijek smatramo da je n >= 0
  - Ako je n = 0, kažemo da je lista prazna

- Pazimo na redne brojeve da bismo izbjegli zabunu:
  - **a1** je **prvi** element liste
  - **ai** je **i-ti** element liste

Definicija strukture podataka:

- **ELTYPE** Tip elemenata liste (može biti bilo koji tip podataka)
- **LIST** Lista (konačan niz elemenata tipa ELTYPE)
- **element** Element liste (podatak tipa ELTYPE)
- **pos** Pozicija elementa u listi

Moguće operacije:

- **END()** Vraća poziciju prvog elementa iza kraja liste
- **FIRST()** Vraća poziciju prvog elementa u listi, a ako je lista prazna vraća END()
- **INSERT(element, pos)** Ubacuje element na pos pomičući preostale elemente iza pos za jedno mjesto dalje. Ako pos nije ispravan, javlja grešku.
- **READ(pos)** Vraća element liste na poziciji pos. Javlja grešku ako pos ne postoji ili ako je pos = END().
- **REMOVE(pos)** Briše element s pozicije pos pomičući preostale elemente iza pos za jedno mjesto unatrag. Javlja grešku ako posne postoji ili ako je pos = END().
- **FIND(element)** Vraća poziciju u listi koja ima vrijednost element. Ako ga nema, vraća poziciju END(). Ako ih postoji više, vraća poziciju prvoga pojavljivanja.
- **EMPTY()** Uklanja sve elemente iz liste i vraća END()
- **NEXT(pos)** Vraća prvu poziciju iza pos, uz pretpostavku da je pos ispravna pozicija i da nije jednaka END()
- **PREV(pos)** Vraća prvu poziciju ispred pos, uz pretpostavku da je pos ispravna pozicija i danije jednaka FIRST().

**Implementacija liste pomoću polja**

- Jedna klasa će predstavljati tip podataka za listu
- Elemente liste ćemo čuvati u polju koje će biti član klase
- Pozicija elementa u listi će biti jednaka indeksu elementa u polju + 1
  - Polje počinje s indeksom 0, a lista s elementom 1
- Brojač će sadržavati poziciju zadnjeg elementa u listi
  - Kako bismo znali koji dio polja je popunjen
  - Kapacitet liste je jednak veličini polja
- U polju ne smije biti "rupa" – svi elementi elementi polja od početka do kraja liste moraju biti popunjeni

- **Prednosti:**
  Pristup i-tom elementu liste je brz
  - Pristup elementima polja je brz jer su poslagani jedan iza drugoga u memoriji

- **Nedostaci:**
  Prilikom kreiranja liste moramo reći koliko najviše elemenata lista može sadržavati (kapacitet liste)
  - Ako lista sadržava manje elemenata, svejedno troši memoriju
  - Ako treba staviti više elemenata u listu, ne možemo bez promjene veličine polja (skupo)

  Operacije umetanja i brisanja su složene – elemente iza umetnog/obrisanog treba pomicati naprijed/nazad

- **Uvijek biti svjestan radi li se o poziciji u listi ili o indeksu u polju**

**Dinamička implementacija liste**

- Realizacija se ostvaruje dinamički pomoću pokazivača
  - Memorijski se prostor dinamički uzima i vraća sa hrpe za vrijeme izvršavanja programa
- Može imati proizvoljan broja čvorova
- Svaki čvor osim elementa liste sadrži i pokazivač na sljedeći čvor

**Jednostruko povezana lista** ili samo **povezana lista** (engl. linked list)

- **Prednost:**
  - Operacije umetanja i brisanja ne zahtijevaju nikakva kopiranja/premještanja
- **Nedostaci:**
  - Ne možemo direktno pristupiti *i*-tom elementu već uvijek moramo krenuti ispočetka
    - Složenost takvih operacija je O(n)
  - Drugi problem je "Pronađi čvor PRIJE ovoga čvora“
    - Zbog toga će patiti i umetanje i brisanje

**Prvi i zadnji element liste:**

- Definiramo **zaglavni čvor** (engl. header) tako da ne sadrži podatak već samo pokazivač na čvor s prvim elementom liste

- Definiramo **repni čvor** (engl. tail) tako da ne sadrži podatak već samo zadnji element liste pokazuje na njega

**Dvostruko povezana lista** (engl. *doubly linked list*)

- Prednost:
  - Sve operacije umetanja i brisanja su složenosti O(1)
  - Prolaz kroz listu bilo kojim smjerom je složenosti O(n)
- Nedostatak:
  - Troši se više memorije

- Svaki čvor sadrži:
  - Podatak (može biti bilo koji tip)
  - Pokazivač na sljedeći čvor u listi
  - Pokazivač na prethodni čvor u listi na još jedan pokazivač po čvoru

STOG

- **Stog** (engl. stack) je lista organizirana prema LIFO principu
  - LIFO (engl. last in first out) znači da zadnji element koji uđe na stog će biti prvi element koji će izaći sa stoga
    - Ekvivalentno, prvi koji uđe će biti zadnji koji će izaći (FILO)
  - Podaci koji su na stogu nazivaju se **elementi stoga**
  - n je broj elemenata na **stogu**
    - Ako je n = 0, kažemo da je stog prazan
  - Svaki stog ima **vrh**(engl. top) – to je posljednji element dodan na stog i prvi koji će biti skinut sa stoga

Struktura podataka:

- **ELTYPE** Tip elemenata stoga (može biti bilo koji tip podataka)
- **STACK** Stog (konačan niz elemenata tipa ELTYPE)
- **element** Element stoga (podatak tipa ELTYPE)
- **vrh** Pozicija vrha

Operacije:

- **EMPTY()** Uklanja sve elemente sa stoga
- **ISEMTPY()** Vraća istinu ako je stog prazan, inače laž
- **PUSH(element)** Ubacuje element na vrh stoga
- **POP()** Skida (uklanja) element sa vrha stoga i vraća ga
- **TOP()** Vraća element s vrha stoga, ali ga ne uklanja sa stoga

**Implementacija stoga pomoću polja**

- **Prednosti**:
  - Pristup vrhu je uvijek brz
    - Jer su elementi poslagani jedan iza drugoga u memoriji
- **Nedostaci:**
  - Prilikom kreiranja stoga moramo reći koliko najviše elemenata stog može sadržavati

- Ako stog sadržava manje elemenata, svejedno troši memoriju
- Ako treba staviti više elemenata na stog, ne možemo bez promjene veličine polja (skupo)
- Pojava kad je stog popunjen pa ne možemo na njega staviti više elemenata se naziva stack overflow

**Dinamička implementacija stoga**

- Može imati proizvoljan broja elemenata
- Svaki element pokazuje **na element ispod**
- Potreban nam je **pokazivač na vrh**
- Potreban nam je jedan element koji predstavlja "dno stoga"
  - Alternativno možemo koristiti **nullptr**

RED

- **Red** (engl. queue) je struktura podataka organizirana prema FIFO principu
  - FIFO (engl. first in first out) znači da će prvi element koji uđe u red biti i prvi element koji će izaći iz reda
    - Ekvivalentno, zadnji koji uđe će biti zadnji koji će izaći (LILO)
- Osnovne karakteristike reda su sljedeće:
  - Elemente u red **ubacujemo** (engl. enqueue) samo na jednom kraju kojeg nazivamo **ulaz** (engl. tail, rear)
    - Element ne možemo ubaciti nigdje drugdje osim na ulazu
  - Elemente iz reda **vadimo** (engl. dequeue) samo na jednom kraju kojeg nazivamo **izlaz** (engl. head, front)
    - Jedini element kojeg možemo izvaditi u nekom trenutku je onaj koji je na izlazu, tj. onaj koji je prvi ušao
  - Ulaz mora biti različit od izlaza

Struktura podataka:

- **ELTYPE** Tip elemenata reda (može biti bilo koji tip podataka)
- **QUEUE** Red (konačan niz elemenata tipa ELTYPE)
- **element** Element reda (podatak tipa ELTYPE)

Operacije:

- **ISEMTPY**() Vraća istinu ako je red prazan, inače laž
- **ENQUEUE(element)** Ubacuje element na kraj reda
- **FRONT()** Vraća element s početka reda, ali ga ne uklanja iz reda
- **DEQUEUE()** Skida (uklanja) element s početka reda i vraća ga

**Implementacija reda pomoću polja**

- Praćenje popunjenosti reda:
  - **_head** pokazuje na indeks **prvog** elementa u redu
    - Onog kojeg treba sljedećeg izvaditi
  - **_tail** pokazuje na indeks **odmah iza zadnjeg** elementa u redu

- Tamo gdje treba sljedećeg ubaciti
- Ako je **_head ==_tail**, onda je red prazan
    - Pritome vrijednosti ne moraju biti jednake nuli
- Ovakva implementacija ima **nedostatak neiskorištenog prostora na početku polja**
- Kada se **dodaje** u red,_head je fiksiran, a _tail se pomiče naprijed za 1
- Kada se **uklanja** iz reda, _tail je fiksiran, a _head se pomiče naprijed za 1

- Rješenje navedenog problema je u **cirkularnom korištenju polja**
- U cirkularnom polju će i _head i _tail prolaskom zadnjeg elementa polja doći na poziciju 0
    - Zbog toga za sljedeću poziciju iza i uzimamo:
    **(i + 1) % veličina_polja**

- Zbog _head == _tail u cirkularnom polju **jedno mjesto u polju** uvijek mora biti prazno

**Dinamička implementacija reda**

- Koriste se dva pokazivača, **_head** i **_tail**
  - **_head** pokazuje na prvi element u redu
  - **_tail** pokazuje na zadnji element u redu
  - Svaki element pokazuje samo na sljedeći element
    - Zadnji pokazuje na nullptr
    - Od _heada ne možemo doći do _taila, ali nam to niti ne treba

STABLA

- Stablo (engl. tree) je skupina povezanih čvorova sa svojstvima:
  - Svaki čvor (engl. node) sadrži jednu ili više vrijednosti
  - Čvorovi su hijerarhijski organizirani (roditelj –djeca)
  - Postoji točno jedan čvor koji nema roditelja i koji se naziva **korijen** ili **ishodište stabla** (engl. tree root)
- Svaki čvor je ujedno i **korijen podstabla** (engl. subtreeroot), a to podstablo može biti složeno (sastavljeno od više čvorova) ili trivijalno (sastavljeno samo od 1 čvora)
- Čvorove koji se nalaze direktno ispod nekog čvora nazivamo njegovom **djecom** (engl. children)
- Osim korijena stabla, svaki čvor ima točno jednog **roditelja** (engl. *parent*), a to je čvor direktno iznad njega
- Čvorove s istim roditeljem nazivamo **braćom** (engl. siblings)
- **Put** (engl. path) od čvora x do čvora y čini niz čvorova kojima se može direktno doći od x do y (pri čemu je svaki čvor na putu roditelj sljedećem čvoru na tom putu)
- Ako se neki put sastoji od n čvorova, onda je **duljina** tog **puta** jednaka n–1
- Ako gledamo neki čvor x:
- **Potomci** (engl. descendants) čvora x su svi čvorovi u stablu do kojih postoji put od x
- **Preci** (engl. ancestors) čvora x su svi čvorovi u stablu od kojih postoji put do x
- **List** (engl. leaf) je čvor koji nema djece
- **Unutrašnji čvor** (engl. internal) je čvor koji ima djece
- **Razina** ili **nivo** ili **dubina čvora** (engl. node level, node depth) predstavlja njegovu udaljenost (duljinu puta) od korijena

- Korijen ima razinu 0, njegova djeca imaju razinu 1, njihova djeca razinu 2, itd.
- **Dubina stabla** (engl. tree depth) je jednaka maksimalnoj razini nekog čvora u stablu
- **Stupanj čvora** je jednak broju njegove djece
- **Stupanj stabla** je jednak stupnju čvora s najviše djece

## Binarna stabla

- **Binarno stablo** (engl. binary tree) je stablo čiji stupanj može biti najviše 2
  - To znači da svaki čvor može imati najviše dva djeteta
- Razlikujemo **lijevo** i **desno** dijete svakog čvora

Tipovi binarnih stabala:

- **Puno** (engl. full) binarno stablo je ono u kojemu svaki čvor koji nije list ima točno 2 djeteta
- **Savršeno** (engl. perfect) binarno stablo je ono koje je puno i u kojem su svi listovi u istoj razini
- **Potpuno** (engl. complete) binarno stablo je ono u kojemu su sve razine (osim možda zadnje) popunjene, a zadnja razina ima sve čvorove popunjene s lijeva

Struktura podataka:

- **ELTYPE** Tip koji se čuva u čvorovima (bilo koji tip podataka)
- **BTREE** Stablo sastavljeno od čvorova tipa ELTYPE
- **node** Čvor stabla (podatak tipa ELTYPE)
- **POSITION** Pozicija čvora unutar stabla

Operacije:

- **CREATE(e)** Kreira korijen stabla u kojeg je smješten element *e*
- **INSERT_LEFT(p, e)** Ubacuje element *e* kao lijevo dijete čvora *p*
- **INSERT_RIGHT(p, e)** Ubacuje element *e* kao desno dijete čvora *p*
- **ROOT()** Vraća korijen binarnog stabla
- **LEFT_CHILD(node)** Vraća lijevo dijete čvora node
- **RIGHT_CHILD(node)** Vraća desno dijete čvora node
- **GET_NODE(pos, e)** Vraća sadržaj zadanog čvora
- **INORDER()** Obilazi stablo INORDER algoritmom
- **PREORDER()** Obilazi stablo PREORDER algoritmom
- **POSTORDER()** Obilazi stablo POSTORDER algoritmom

### Obilazak binarnog stabla

- INORDER Princip obilaska: Left, Parent, Right
  Često se koristi na binarnim stablima traženja (BST) budući da vraća vrijednosti u sortiranom obliku (definiranom samim BST-om)

- PREORDER Princip obilaska: Parent, Left, Right
  Često se koristi za dupliciranje stabla jer prvo obradi roditelja, a tek onda djecu

- POSTORDER Princip obilaska: Left, Right, Parent
  Često se koristi pri brisanju čvorova i uništenju stabla jer prvo obradi djecu, a tek onda roditelja

## Implementacija binarnog stabla pomoću polja

- Način numeriranja:
  - Nadopunit ćemo naše stablo do savršenog stabla
  - U savršenom stablu ćemo napraviti numeriranje čvorova:
    - Korijen će imati broj 0
    - Čvorovi razine 1 će imati brojeve od 1 do n1
    - Čvorovi razine 2 će imati brojeve od n1+ 1 do n2
    - I tako dalje sve dok ne obradimo sve razine

- Na taj način smo dobili sljedeće:
  - i-ti čvor stabla se nalazi na indeksu i
  - Lijevo dijete i-tog čvora se nalazi indeksu 2i + 1
  - Desno dijete i-tog čvora se nalazi na indeksu 2i + 2

## Implementacija binarnog stabla pomoću polja

- Svaki čvor će uz element sadržavati i dva pokazivača: jedan za lijevo i jedan za desno dijete
  - Ako neki čvor nema lijevo, odnosno desno dijete onda odgovarajući pokazivač pokazuje na nullptr

## Binarno stablo traženja BST

Struktura podataka:

- **ELTYPE** Tip koji se čuva u čvorovima (bilo koji tip podataka)
- **BST** Binarno stablo traženja sastavljeno od čvorova tipa ELTYPE
- **Node** Čvor stabla (podatak tipa ELTYPE)
- **POSITION** Pozicija čvora unutar stabla

Operacije:

- **INSERT(e)** Umeće zadani element na odgovarajuće mjesto u stablu
- **EXISTS(e)** Vraća postoji li tražena vrijednost u stablu ili ne

## Hrpa

Hrpa ili gomila (engl. heap) je struktura podataka koja zadovoljava uvjete:

- Potpuno je binarno stablo
  - Može biti bilo kakvo potpuno stablo, ali ćemo mi promatrati samo binarna stabla
- Vrijednost u čvoru roditelju je:
  - Uvijek veća ili jednaka svim vrijednostima djece (engl. max-heap)
  - Uvijek manja ili jednaka svim vrijednostima djece (engl. min-heap)
  - Napomena: primijetimo da se ništa ne kaže o vrijednostima braće

Struktura podataka:

- **ELTYPE** Tip koji se čuva u čvorovima (bilo koji tip podataka)
- **HEAP** Hrpa sastavljena od čvorova tipa ELTYPE
- **node** Čvor stabla (podatak tipa ELTYPE)

- **POSITION** Pozicija čvora unutar stabla

Operacije:
- **IS_EMPTY()** Vraća true ako je hrpa prazna, inače false
- **INSERT(e)** Umeće zadani element na hrpu
- **REMOVE()** Uklanja element s vrha hrpe i vraća ga

**Prioritetni red**

Prioritetnim redom nazivamo FIFO strukturu u kojoj svaki element ima definiran prioritet
- Prvo se obrađuju elementi najvišeg prioriteta po FIFO principu
- Nakon toga se na jednak način obrađuju elementi nižeg prioriteta i tako sve do najnižeg prioriteta

Može biti izveden na razne načine:
- Pomoću reda
  - Kod dodavanja poštujemo prioritete, vadimo s početka
  - Dodavanje na kraj, kod vađenja poštujemo prioritete
- Pomoću hrpe
  - Element s najvišim prioritetom je uvijek u korijenu
  - Svaku razinu popunjavamo s lijeva na desno prema silaznom (opadajućem) prioritetu

SORTIRANJE

Sortiranje (engl. sorting) je postupak slaganja niza elemenata u određeni redoslijed
Kod sortiranja je bitno:
- Kriterij sortiranja
  - Primjerice, osobe možemo sortirati po imenu i prezimenu
- Složenost algoritma
- Koliko algoritmu treba memorije kako bi izvršio sortiranje
  - Dobra mjera je i broj zamjena (engl. swap)
- Je li algoritam rekurzivan ili iterativan
- Je li algoritam stabilan (stabilan je ako ne mijenja mjesta dva jednaka elementa)

**Comparison sort algoritmi**

- uspoređuje elemente samo na osnovu jednog operatora za uspoređivanje
  - Primjerice, manje od, veće od, …

**Bogo sort**

- Izrazito neefikasan. U praksi se nikad ne koristi
- Taktika:
  1. Provjeri jesu li elementi sortirani (ako jesu, kraj)
  2. Ako nisu, promiješaj

**Bubble sort**

- Taktika:
  1. usporedi dva susjedna elementa i ako nisu u dobrom redoslijedu, zamijeni ih
- Jedina prednost je ugrađena sposobnost detekcije da je polje već sortirano

**Selection sort**

- Taktika:
  1. Neka polje ima sortirani i nesortirani dio
  2. Pronađi najmanji element u nesortiranom dijelu:
  - Zamijeni ga s prvim elementom u nesortiranom dijelu
  - Proglasi da sad taj element pripada sortiranom dijelu
  3. Ponavljaj dok cijelo polje ne bude sortirano

**Insertion sort**

- Taktika (polje ima sortirani i nesortirani dio):
  1. Uzmi prvi element u nesortiranom dijelu:
  - Proglasi da sad taj element pripada sortiranom dijelu
  - Mijenjaj mu mjesto s elementima u sortiranom dijelu dok ne dođe na ispravno mjesto
  2. Ponavljaj dok cijelo polje ne bude sortirano

**Shell sort**

- Taktika:
  1. polje ćemo prirediti tako da insertion sort na njemu radi brže.
  - Dijelimo polje na h1 potpolja, a svako potpolje čine sljedeći elementi:
    1. potpolje: data[0], data[h1], data[2h1], …
    2. potpolje: data[1], data[h1+1], data[2h1+1], …
    h1-to potpolje: data[h1-1], data[2h1-1], data[3h1-1], …

- Niz h1, h2, … , hk= 1 nazivamo sekvenca razmaka (engl. gap sequence)

  Tokudina sekvenca iz 1992:

```
void shell_sort(int data[], int n) {
// Prvo generiramo sekvencu.
vector<int> sequence;
int h;
int k = 1;
while (true) {
        h = ceil((pow(9, k) - pow(4, k)) / (5 * pow(4, k - 1)));
        if (h < n) {
                sequence.push_back(h);
                k++;
        }
        else {
                break;
        }
```

```
    }
    // Sad koristimo elemente sekvence.
    for (int j = sequence.size() - 1; j >= 0; j--) {
            int step = sequence[j];
            for (int i = step; i < n; i++) { // Krećemo od step
                    int temp = data[i];
                    int j;
                    for (j = i; j >= step && data[j - step] > temp; j = j - step) {
                            swap(data[j], data[j - step]);
                    }
            }
    }
}
```

**Merge sort**

- Taktika:
  1. dva već sortirana polja je jednostavno spojiti u jedno veće uz održavanje sortiranosti
  - Algoritam početno dijeli polje na dva (pod)jednaka dijela
  - Svaki od tih dijelova ponovo dijeli na dva dijela sve dok ne dođe do dijela veličine 1
    - Njegovi elementi su po definiciji već sortirani
  - U svakom sljedećem koraku, algoritam spaja dva po dva dijela
  - U zadnjem koraku spaja dva dijela u jedan i time dobijemo sortirano polje

**Quick sort**

- Taktika:
  1. Odredi pivotni element
  2. Elemente manje od njega prebaci lijevo, veće prebaci desno.
     - Sad je pivotni element na finalnom mjestu, te imamo dio s manjim elementima i dio s većim elementima
       - Jednake možemo staviti na jednu ili drugu stranu
  3. Rekurzivno napravi prethodne korake na dijelu s manjim i na dijelu s većim elementima
- Rekurzivan algoritam
  - Polje koja ima 0 ili 1 element ne treba sortirati (uvjet zaustavljanja)

- Postoji nekoliko varijanti izbora pivota:
  1. Uzeti slučajni element
  2. Uzeti srednji element
  3. Uzeti medijan (vrijednost u sredini) prvog, srednjeg i zadnjeg elementa

**Counting sort**

- Counting sort je algoritam sortiranja temeljen na ključevima koji su mali cijeli brojevi
  - Nije baziran na uspoređivanju elemenata kao algoritmi do sada
- Svi elementi koje želimo sortirati moraju imati definiran ključ
  - Ključ broja je upravo taj broj
  - Nama će vrijediti: ključ = element
- Ideja algoritma je prebrojati koliko puta se koji ključ pojavljuje
  - Na osnovu te informacije znamo na koje mjesto u sortiranom polju dolazi element

**Radix sort**

- Postoje dva različita pristupa implementaciji:
  1. Jedan koja pregledava znamenke s lijevo na desno
  • Prvo pregledava najznačajniju znamenku, pa se uobičajeno naziva MSD radix sort (engl. most significant digit)
  2. Drugi koji pregledava znamenke s desna na lijevo
  • Prvo pregledava najmanje značajnu znamenku, pa se naziva LSD radixsort (engl. least significant digit)
- Taktika:
  1. Pronađemo najveći broj i on nam daje najveći broj znamenki
  • Brojeve s manje znamenki nadopunjujemo nulama s lijeva
- Počevši s desne strane, za svaku znamenku ponavljamo:
  • Napravimo counting sortu pomoćno polje prema toj znamenci
  • Counting sort je vrlo efikasan jer je max= 9
  • Prekopiramo sadržaj iz pomoćnog polja u glavno
- Broj ponavljanja je određen brojem znamenki najvećeg elementa
- Za svaku znamenku dobivamo "sortiranije" polje
  • Nakon zadnje znamenke dobivamo potpuno sortirano polje

PRETRAŽIVANJE

- Algoritam pretraživanja ustanovljava postoji li neki element u kolekciji ili ne
- dva osnovna algoritma pretraživanja:
  1. Linearno pretraživanje
  2. Binarno pretraživanje

**Linearno pretraživanje**

- Pregledava kolekciju element po element
- Ukoliko bismo imali sortiranu kolekciju, traženje bi moglo biti brže

**Binarno pretraživanje**

Algoritam binarnog pretraživanja (engl. binary search) radi na sortiranoj kolekciji:

- Ako kolekcija sadrži samo jedan element
  - Usporedi element s traženom vrijednošću
- Inače, odredi u kojoj polovici kolekcije se tražena vrijednost može nalaziti
- Traži vrijednost u toj polovici koristeći binarno pretraživanje

```
//implementacija binarnog pretraživanja

int search(int *polje, int n, int what) {
    int low = 0;
    int high = n - 1;

    while (low <= high) {
            int mid = (low + high) / 2;
```

```
            if (polje[mid] == what) {
                return mid;
            }
            else if (polje[mid] > what) {
                high = mid - 1;
            }
            else {
                low = mid + 1;
            }
        }
        //moramo vratit nešto jer ovo gore vraća samo ako ima nešto

        return -1;
    }
```

**Binarno stablo traženja**

Binarno stablo traženja (engl. BST –binary search tree) je podvrsta binarnog stabla sa sljedećim svojstvima:

- Svi podaci u lijevom podstablu su manji od podatka u korijenu podstabla
- Svi podaci u desnom podstablu su veći ili jednaki podatku u korijenu podstabla
- Svako podstablo je i samo binarno stablo traženja

**Digitalno stablo traženja**

Digitalno stablo traženja (engl. DST – digital search tree) se temelji na binarnoj reprezentaciji ključa i procesiranju njegovih bitova s lijeva na desno

- Ishodište sadržava bilo koji ključ
- Svi ostali ključevi koji započinju s 0 se nalaze u lijevom podstablu
- Svi ostali ključevi koji započinju s 1 se nalaze u desnom podstablu
- Svako lijevo i desno podstablosu digitalna stabla traženja prema preostalim bitovima
- Broj razina stabla određuje broj bitova u ključu
  - Broj bitova mora biti jednak za sve ključeve (Prema potrebi nadopunimo s nulama s lijeve strane)

**Binary trie**

Binary trie (prema information retrieval) je binarno stablo koje ima dvije vrste čvorova:

- Čvorovi grananja (engl. branch nodes)
  - Svaki čvor grananja sadrži samo dva pokazivača:
  1. Pokazivač na lijevo dijete
  2. Pokazivač na desno dijete
  - Čvorovi grananja služe kao putokazi do čvorova s elementima

- Čvorovi s elementima (engl. element nodes)
  - Svaki čvor s elementima sadržava točno jedan element

## Komprimirani binary trie

Komprimirani binary trie (engl. compressed binary trie) je način kako se može smanjiti dubina stabla uklanjanjem čvorova grananja s jednim null pokazivačem

- 1. Svakom čvoru grananja se dodaje novi član GledajBit
  - Ima vrijednost jednaku dubini čvora + 1
  - Govori koji bit treba gledati prilikom grananja u tom čvoru
- 2. Svaki čvor grananja s jednim null pokazivačem se uklanja

## Tablica simbola – APSTRAKTNI TIP PODATAKA

Apstraktni tip podataka tablica simbola(engl. symbol table)je nesortirana kolekcija parova ključ/vrijednost (engl. key/value pairs)

- Ključ je string koji jedinstveno označava par (ne mogu postojati dva jednaka ključa)
- Vrijednost je podatak vezan uz ključ i može biti bilo kojeg tipa

Sljedeće operacije su nam zanimljive na tablici simbola:

- Umetanje para ključ/vrijednost
- Traženje prema ključu
- Uklanjanje prema ključu

Dvije moguće implementacije:

- Jednostavna implementacija (dvostruko) povezanom listom
- Efikasnija implementacija hash tablicama

**//tablica_simbola.h**

```
#include <string>
#include "osoba.h"
#include "lista_dinamicka.h"

using namespace std;

class tablica_simbola {
private:
	lista_dinamicka _lista;
public:
	bool put(string key, ELTYPE value);
	ELTYPE* get(string key);
	bool remove(string key);
};
```

**//tablica_simbola.cpp**

```
#include "tablica_simbola.h"
#include "lista_dinamicka.h"

using namespace std;

bool tablica_simbola::put(string key, ELTYPE value) {
	for (POSITION node = _lista.first(); node != _lista.end(); node = node->next) {
		if (node->element.oib == key) {
			return false; // Ključ već postoji.
		}
	}
```

```
	_lista.insert(value, _lista.end());
	return true;
}

ELTYPE* tablica_simbola::get(string key) {
	for (POSITION node = _lista.first(); node != _lista.end(); node = node->next) {
		if (node->element.oib == key) {
			return &node->element; // Vrati adresu.
		}
	}
	return nullptr; // neme elementa s traženim ključem.
}

bool tablica_simbola::remove(string key) {
	for (POSITION node = _lista.first(); node != _lista.end(); node = node->next) {
		if (node->element.oib == key) {
			_lista.remove(node); // Ukloni.
			return true;
		}
	}
	return false; // Ključ ne postoji.
}
```

**Rječnici**

U velikom broju aplikacija postoji potreba za posebnim tipom kolekcija koje se nazivaju rječnici (engl. dictionaries):

- Rječnik sadržava parove ključ/vrijednost(engl. key/value pairs)
- Vrijednosti dohvaćamo pomoću ključeva koji moraju biti jedinstveni
- Rječnik podržava samo operacije INSERT, SEARCH i DELETE čija bi složenost trebala biti O(1)
  • Zbog toga su rječnici izuzetno brzi
- Tablice simbola je podvrsta rječnika bez velike fleksibilnosti

**Hash tablica** (hrv. raspršena tablica, engl. hash table) je struktura podataka posebno pogodna za implementaciju rječnika

**Tablice s direktnim adresiranjem**

- Tablica s direktnim adresiranjem je u stvari obično polje
- Kod hash tablica mjesto u polju na nekom indeksu se često naziva **slot ili bucket**
- Svaki rječnik se sastoji od parova ključ/vrijednost
  **Ključ** u tablici s direktnim adresiranjem je indeks polja (uvijek cijeli broj > 0)
  **Vrijednost** je ono što piše u polju na tome mjestu
- **Na osnovu ključeva radimo polje** tako da se svaki ključ može spremiti u polje (ključ je indeks!)

**HASH TABLICE**

- Kod tablica s direktnim adresiranjem, element s ključem key je smješten na indeks key - Jer je indeks = ključ
- Kod hash tablica, element s ključem key je smješten na indeks **h(key),** odnosno, indeks = h(key)
  - Koristimo **hash funkciju** h kako bismo od ključa izračunali indeks
  - To nam omogućava da imamo polje koje je znatno manje od ukupnog broja ključeva
- Hash funkcija može više ključeva pretvarati u isti indeks, što se naziva **kolizija** (engl. collision) i o čemu treba voditi računa
- Postoje razni načini **rješavanja kolizija**:
  1. Kod tablica s direktnim adresiranjem kolizija je izbjegnuta funkcijom h(key) = key
  2, Tehnika otvorenog adresiranja
  3. Ulančavanje (engl. chaining)

**<u>Ulančavanje</u>**

Kod ulančavanja, sve ključeve koji se hashiraju u isti bucket stavljamo u **listu** koja pripada tom bucketu

- Performanse su najbolje kad su sve liste u svim bucketima podjednako duge (ali ne preduge)
- Ako su liste preduge, treba povećati broj bucketa
- Na duljinu listi utječe hash funkcija

Postoje dva jednostavna načina kako dobiti dobru hash funkciju:

- Metoda dijeljenja
  **h(key) = key mod m**

  - Pri čemu je m broj bucketa
  - Što je veći *m*, to su manje liste u *bucketima*

- Metoda množenja
  **h(key) = floor( m* ( (key * A) mod1) )**

  - gdje je 0 < A< 1
  - (key* A) mod1 označava decimalni dio od key* A
  - A= (√5–1) / 2= **0,6180339887**…
  - Vrijednost m nije kritična, obično je neka potencija broja 2

**<u>Ključevi drugog tipa</u>**

Koristit ćemo sljedeći algoritam koja prima ulazni **string** i izračunava broj iz $N_0$:
n <- length (ulazni_string)
zbroj <- INITIAL_VALUE * $M^n$
foreach znak in ulazni_string:
n <- n - 1
zbroj <- zbroj + ascii(znak) * $M^n$

- Algoritam u stvari radi sljedeće (primjer pretvaranja „Bart”):

zbroj = INITIAL_VALUE * $M^4$ + 'B' * $M^3$ + 'a' * $M^2$ + 'r' * M + 't'

- **INITIAL_VALUE** je neka unaprijed definirana konstantna vrijednost
- **M** je neka unaprijed definirana konstantna vrijednost
- **ASCII vrijednost** nekog znaka je njegova brojčana vrijednost u ASCII tablici, primjerice:

```
char znak= 'A';
cout << znak << endl; // Ispisuje: A
cout<< (unsigned int)znak<< endl; // Ispisuje: 65
```

- Bernsteinova funkcija ima INITIAL_VALUE= 5381 i M= 33
- Kernighan/Ritchie funkcija ima INITIAL_VALUE= 0 i M= 31

- potrebno koristiti **unsigned long long**

```
// rijecnik_ulancavanjem.cpp
class rjecnik {
private:
	static const int BROJ_ELEMENATA = 23;
	lista_dinamicka* _polje[BROJ_ELEMENATA];
	unsigned int h(KEY key);
	const int INITIAL_VALUE = 5381;
	const int M = 33;
public:
	rjecnik();
	void insert(KEY key, ELTYPE value);
	void remove(KEY key);
	ELTYPE* search(KEY key);
	void display_buckets();
	~rjecnik();
};

//trazenje po string kljucu (rijecnik_ulancavanjem.cpp)
unsigned int rjecnik::h(KEY key) {
	unsigned int n = key.size();
	unsigned long long index = INITIAL_VALUE * pow(M, n);
	for (unsigned int i = 0; i < key.size(); i++) {
		n--;
		index += (unsigned int)key[i] * pow(M, n);
	}
	cout << "Key:" << key << ", Value:" << index << endl;
	return index % BROJ_ELEMENATA;
}

//prikazivanje bucketa (rijecnik_ulancavanjem.cpp)
void rjecnik::display_buckets() {
	for (int i = 0; i < BROJ_ELEMENATA; i++) {
		cout << "Bucket " << i << " : " << _polje[i]->count() << endl;
	}
```

***//lista_dinamicka.h***

```
class lista_dinamicka {
private:
        POSITION _head; // "Prvi ispred" početka liste.
        POSITION _tail; // "Prvi iza" kraja liste.

public:
        lista_dinamicka();
        POSITION end();
        POSITION first();
        bool insert(ELTYPE element, POSITION pos);
        bool read(POSITION pos, ELTYPE& element);
        bool remove(POSITION pos);
        POSITION find(ELTYPE element);
        POSITION find(string kljuc);
        POSITION empty();
        POSITION next(POSITION pos);
        POSITION prev(POSITION pos);
        unsigned int count();
        ~lista_dinamicka();
};
```

***//lista_dinamicka.cpp***

```
unsigned int lista_dinamicka::count() {
        unsigned int c = 0;
        cvor* temp = _head;
        while (temp->next != end()) {
                c++;
                temp = temp->next;
        }
        return c;
}
```

## Preljevno područje

Preljevno područje (engl. over flow area) je još jedna metoda rješavanja kolizije kod hash tablica

Polje dijelimo na dva dijela:

- Primarno područje
  - Predstavlja dio u koji pokušavamo staviti element
  - Dimenzioniramo ga prema mogućim ključevima i hash funkciji

- Preljevno područje
  - Dodatno ćemo element povezati s elementom u primarnom području s kojim se desila kolizija
  - Princip sličan ulančavanju jer imamo „listu” povezanih elemenata
  - Prvi element je uvijek u primarnom, a svi ostali u preljevnom području
  - Preljevno područje možemo implementirati drugim poljem

- Svaki bucket će sadržavati jedan element sastavljen od:
  1. Ključa
  2. Vrijednosti
  3. Adrese sljedećeg elementa (putokaza next)

**// prikazivanje sdrzaja bucketa (rijecnik_preljevnim_podrucjem.cpp)**

```
void rjecnik::display_buckets() {
	cout << "PRIMARNO" << endl;
	for (int i = 0; i < N_PRIMARNO; i++) {
		cout << "Bucket:" << i << " kljuc:" << _primarno[i].key << "(" <<
_primarno[i].naziv << ")" << endl;
	}
	cout << "PRELJEVNO" << endl;
	for (int i = 0; i < N_PRELJEVNO; i++) {
		cout << "Bucket:" << i << " kljuc:" << _preljevno[i].key << "(" <<
_preljevno[i].naziv << ")" << endl;
	}
}
```

// **rijecnik_preljevnim_podrucjem.h**

```
class rjecnik {
private:
	static const int PRAZNO = -1;
	static const int N_PRIMARNO = 50;
	static const int N_PRELJEVNO = 200;
	ELTYPE _primarno[N_PRIMARNO];
	ELTYPE _preljevno[N_PRELJEVNO];
	unsigned int h(KEY key);

public:
	rjecnik();
	void insert(KEY key, ELTYPE value);
	void remove(KEY key);
	ELTYPE* search(KEY key);
	void display_buckets();
	~rjecnik();
};
```

**// pretrazivanje po kljucu (rijecnik_preljevnim_podrucjem.cpp)**

```
ELTYPE* rjecnik::search(KEY key) {
	unsigned int index = h(key);

	if (_primarno[index].key != PRAZNO) { // Postoji vrijednost u primarnom.
		if (_primarno[index].key == key) { // Tu je tražena vrijednost.
			return &_primarno[index];
		}
		else { // Nije u primarnom, možda je u preljevnom.
			ELTYPE* curr = _primarno[index].next;
			while (curr != nullptr) {
				if (curr->key == key) {
					return curr;
				}
				curr = curr->next;
			}
		}
	}

	return nullptr;
}
```

```
// brisanje po kljucu (rijecnik_preljevnim_podrucjem.cpp)
void rjecnik::remove(KEY key) {
	unsigned int index = h(key);

	if (_primarno[index].key != PRAZNO) { // Postoji vrijednost u primarnom.
		if (_primarno[index].key == key) { // Brišemo iz primarnog.
			ELTYPE* next = _primarno[index].next;
			_primarno[index].key = PRAZNO;

			if (next != nullptr) { // Prebaci iz preljevnog u primarno.
				_primarno[index] = *next;
				next->key = PRAZNO;
			}
		}
		else { // Nije u primarnom, možda je u preljevnom.
			ELTYPE* curr = &_primarno[index];
			while (curr->next != nullptr) {
				if (curr->next->key == key) { // Pronašao traženi ključ!
					curr->next->key = PRAZNO; // Obrišemo.
					curr->next = curr->next->next; // Prespojimo.
					return;
				}
				curr = curr->next;
			}
		}
	}
}
```

## Otvoreno adresiranje

U otvorenom adresiranju se parovi ključ/vrijednost čuvaju u samom polju, nema povezanih lista

- Svaki bucket može ili biti prazan ili sadržavati točno jedan par ključ/vrijednost
- Posljedica je ta da se hash tablica može napuniti do vrha i da je sljedeće umetanje nemoguće
- Hash funkcija za svaki ključ izračunava **niz indeksa** i svaka operacija (INSERT/SEARCH/DELETE) koristi taj niz za svoj rad

Hash funkcija na osnovu dva parametra izračunava indeks na traženom mjestu niza

- Ključ **key**
- Cijeli broj **i** koji označava željeno mjesto u nizu i koji je u intervalu [0, m–1] (gdje je m veličina polja)

- **Vrijednost ključa -1** označava prazni element polja
  *static const int PRAZNO = -1;*

- **Vrijednost ključa -2** označava obrisani element polja
  *static const int OBRISANO = -2;*

Kod otvorenog adresiranja hash funkcija mora moći izračunati niz indeksa - **sekvenca isprobavanja** (engl. probing sequence):

- Za svaki ključ sekvenca treba biti drukčija
- Svaka generirana vrijednost u sekvenci treba biti jedinstvena unutar te sekvence
- Sekvenca treba sadržavati sve (ili skoro sve) buckete

Dizajniranjem dobre hash funkcije dobivamo dobru sekvencu isprobavanja

- Linearno isprobavanje
  **h(key, i) = (h'(key) + i) mod m**

  Gdje je:
  - m, veličina polja
  - i = 0, 1, 2, …, m–1
  - h'(key) je neka pomoćna hash funkcija
    Može biti bilo kojeg oblika kojeg smo do sada koristili (npr. dobivena metodom dijeljenja, …)
  - problem **primarnog grupiranja** (engl. primary clustering)

- Kvadratno isprobavanje
  **h(key, i) = (h'(key) + $c_1 * i + c_2 * i^2$) mod m**

  Gdje je:
  - m veličina polja
  - i= 0, 1, 2, …, m–1
  - h'(key) pomoćna hash funkcija
  - c1 i c2 konstante (uz c2 različit od 0)
    - Ako je c1= 1 i c2= 0, dobijemo linearno isprobavanje
  - problem **sekundarnog grupiranja** (engl. secondary clustering)

- Dvostruko hashiranje
  **h(key, i) = ($h_1$(key) + i * $h_2$(key)) mod m**

  Gdje je:
  - m veličina polja – najbolje potencija broja 2
  - i= 0, 1, 2, …, m–1
  - $h_1$(key) i $h_2$(key) pomoćne hash funkcije, neovisne jedna o drugoj
    - dizajnirati tako da $h_2$ uvijek generira neparni broj