Strukture podataka i algoritmi

(v.2005-02-21)

Ova skripta namijenjena je studentima Odjela za stručne studije Sveučilišta u Splitu (bivše Veleučilište u Splitu) kao literatura za predmet "Strukture podataka i algoritmi". U prvom dijelu je izneseno gradivo sa predavanja, dok je drugi dio posvećen vježbama i konkretnoj implementaciji struktura i algoritama.

Dio sadržaja preuzet je iz predavanja M. Bulaje 2003. godine.

VAŽNO! U skripti se koristi C kod umjesto pseudo-koda i to isključivo kao ilustracija principa kako bi se nešto moglo iskodirati, a ne kao kod koji je spreman za uključivanje u vlastite programe. Dakle, dijelovi koda uopće *nisu testirani* i vjerojatno sadrže bugove ili čak greške zbog kojih se ne mogu niti kompajlirati!

1 Apstraktni tipovi podataka

Apstraktni tip podataka (ATP[1]) definiran je modelom podataka (što su podaci s kojima radimo) te dozvoljenim operacijama nad ATP-om. ATP *ne definira* konkretnu reprezentaciju podataka u memoriji niti kako se operacije izvršavaju. Međutim, ATP mora definirati *složenost* svake pojedine operacije.

Sučelje (interface) Sučelje ATP-a je specifikacija modela podataka u nekom programskom jeziku. Isto sučelje može imati više različitih implementacija.

Implementacija Implementacija je stvarna programska izvedba sučelja. Isto sučelje može imati više različitih implementacija.

Dobra programerska praksa nalaže da programi pristupaju ATP-u isključivo preko definiranog sučelja. Tada promjena implementacije ne utječe na rad takvih programa. S druge strane, ako programi pristupaju implementacijskim detaljima ATP-a, tada svaka promjena implementacije može unijeti bugove u programe koji su prije ispravno radili.

Kod dizajniranja ATP biblioteke postavlja se pitanje tko je "vlasnik" objekata koji se stavljaju u ATP. To je posebno važno ako su objekti pointeri. Postavlja se pitanje tko će osloboditi memoriju na koju pointeri pokazuju kada se element briše iz ATP-a, ili kada se cijeli ATP briše. U našoj implementaciji (a i većina postojećih implementacija tako radi) za to je zadužen korisnik ATP-a.

1.1 ATP u C-u

C jezik ima nekoliko svojstava koja će se koristiti pri implementaciji ATP-a. Ta svojstva su detaljnije opisana u sljedećim odjeljcima.

1.1.1 Header datoteke

C ima mehanizam header datoteka (tradicionalno sa `.h` ekstenzijom) koje se u program uključuju `#include` pretprocesorskom direktivom. Taj mehanizam će se iskoristiti za odvajanje sučelja od implementacije: sučelje ATP-a se definira u header datoteci, a implementira se nezavisno.

Dobra praksa pri pisanju header datoteka je sljedeći predložak:

```
#ifndef STACK_H__ /* ovo se obicno poklapa sa imenom .h datoteke uz */
#define STACK_H__ /* . zamijenjeno sa _ i dodano __ na kraju imena */

/* ostatak deklaracija */

#endif
```

`#ifndef/#define/#endif` omogućava višestruko uključivanje (`#include`) header datoteke u program bez izazivanja grešaka u procesu kompajliranja (npr. ako se dvaput definira ista struktura).

[1] engleski: ADT - Abstract Data Type

1.1.2 Anonimni pointeri

C dozvoljava deklariranje pointera na neku strukturu kojoj se ne zna definicija. Na primjer,

```
struct stack *p;
```

definira `p` kao pointer na `struct stack`, međutim ne može se pristupiti nijednom članu strukture `struct stack` dok se negdje ne definira njen sadržaj. U praksi će se struktura definirati tek u implementaciji ATP-a.

Svi ATP-ovi će biti dinamički alocirani te će se u sučelju uvijek napraviti `typedef` na *pointer* na strukturu, npr:[2]

```
typedef struct stack *Stack_t;
```

1.1.3 void pointeri

Za razliku od C++-a, C nema *template* mehanizam. To predstavlja problem ako se želi napraviti *univerzalna* struktura podataka koja može u sebi sadržavati *bilo koji* tip podatka – na prvi pogled je nužno za svaki tip podatka imati *posebnu* strukturu podataka i poseban skup funkcija koje rade s njom, npr. posebne tipove i funkcije za stack `int` i `float` brojeva.

Olakšavajuća okolnost[3] je ta da je `void*` *kompatibilan* sa svim vrstama pointera.[4]

Jedna moguća implementacija univerzalne strukture podataka je pohranjivanje isključivo void pointera. Nedostatak je što se tada i jednostavni tipovi (`int`, `short`, `float` i sl.) moraju dinamički alocirati.

Druga mogućnost je slična prvoj, jedino što se sam `void*` koristi za spremanje "malih" cjelobrojnih tipova podataka (onih za koje je `sizeof(TIP) <= sizeof(void*)`. Tada se podaci mogu dobiti "van" iz `void*` pomoću cast-a. "Veliki" tipovi i dalje moraju biti dinamički alocirani

Treća mogućnost je specificiranje veličine tipa u konstruktoru strukture podataka. Tada struktura podataka sama alocira "komade" memorije u koje fizički kopira podatke koje joj proslijedi glavni program. Sama struktura podataka *ne zna* interpretirati podatke – ona ih isključivo pohranjuje na određeni način. Ako je ipak potrebna interpretacija, tada korisnik prosljeđuje pointer na funkciju koja "zna" kako interpretirati podatke u komadu memorije. Takav pristup slijedi npr. C library funkcija `qsort`.

Budući da je ovim predavanjima cilj ilustracija implementacije, a ne izvedba univerzalne biblioteke, koristit će se fiksni tip pohranjen u strukturu. To znači da u programu možemo imati strukturu podataka samo jednog tipa, što nam ovdje neće smetati. Za stvarno univerzalne biblioteke podataka u C-u mogu se pogledati reference.

1.2 Pobrojani ATP-ovi

Sljedeći odjeljci taksativno nabrajaju najčešće ATP-ove, popisuju podržane operacije te (u jednostavnijim slučajevima) daju vremensku složenost (najgori slučaj) pojedinih operacija. U izrazima složenosti n označava broj elemenata trenutno u ATP-u.

---

[2] Iako su imena tipova koja završavaju sa `_t` formalno C standardom *rezervirana*, mi ćemo ih ipak koristiti jer C tradicionalno koristi imena koja se sastoje od svih malih slova. Tako imamo minimalnu mogućnost kolizije sa budućim proširenjima C jezika

[3] koja je *maknuta* iz C++-a

[4] Ali samo pointera na podatke. Prema ISO C-u, `void*` ne može držati funkcijski pointer. U većini implementacija to radi, ali nije garantirano.

1.2.1 stack

Apstrakcija stack-a (stoga): struktura podataka sa LIFO (Last In, First Out) pristupom. Element se može staviti na vrh stacka (*push*) ili skinuti sa vrha stacka (*pop*).

- Tipovi:
  - `Stack_t`: tip stack-a u sučelju
  - `Element_t`: tip elementa u stacku
- Implementacija pomoću:
  - niza
  - vezane liste
- Operacije:

```
1 Stack_t stack_new(void)
2 void stack_delete(Stack_t s)
3 void stack_push(Stack_t s, Element_t e)
4 void stack_pop(Stack_t s)
5 Element_t stack_top(Stack_t s)
6 bool stack_empty(Stack_t s)
```

1	$O(1)$	Stvara prazan stack.
2	Vidi napomenu.	Uništava stack.
3	$O(1)$	Stavlja element na vrh stacka.
4	$O(1)$	Uklanja element s vrha stacka.
5	$O(1)$	Vraća element na vrhu stacka.
6	$O(1)$	Ispituje da li je stack prazan.

Napomena: Ovisno o implementaciji, operacija ima sljedeću složenost:

- $O(1)$ za implementaciju nizom
- $O(n)$ za implementaciju vezanom listom.

1.2.2 queue

Apstrakcija reda (kao u trgovini): struktura podataka sa FIFO (First In, First Out) pristupom.

- Tipovi:
  - `Queue_t`: tip reda u sučelju
  - `Element_t`: tip elementa u redu
- Implementacija pomoću:
  - cirkularnog niza
  - vezane liste
- Operacije:

```
Queue_t queue_new(void)
void queue_delete(Queue_t q)
```

```
3   void queue_enqueue(Queue_t q, Element_t e)
4   void queue_dequeue(Queue_t q, Element_t e)
5   Element_t queue_front(Queue_t q)
6   bool queue_empty(Queue_t q)
```

1	$O(1)$	Stvara prazan red.
2	Vidi napomenu.	Uništava red.
3	$O(1)$	Ubacuje element na kraj reda.
4	$O(1)$	Uklanja element s početka reda.
5	$O(1)$	Vraća element na početku reda.
6	$O(1)$	Ispituje da li je red prazan.

Napomena: Ovisno o implementaciji, operacija ima sljedeću složenost:

- $O(1)$ za implementaciju cirkularnim nizom
- $O(n)$ za implementaciju vezanom listom.

1.2.3 set

Apstrakcija *neuređenog* skupa elemenata: nije moguće odrediti redoslijed kojim su elementi umetani. U skupu se svaki element pojavljuje najviše jednom.

- Tipovi:
  - `Set_t`: tip skupa u sučelju
  - `Element_t`: tip elementa u redu
- Implementacija pomoću:
  - bit-vektora (primjenjivo za podskup cijelih brojeva)
  - sortirane vezane liste
- Operacije:

```
1   Set_t set_new(void)
2   void set_delete(Set_t s)
3   void set_insert(Set_t s, Element_t e)
4   void set_remove(Set_t s, Element_t e)
5   bool set_is_member(Set_t s, Element_t e)
6   bool set_is_subset(Set_t s1, Set_t s2)
7   Set_t set_union(Set_t s1, Set_t s2)
8   Set_t set_intersection(Set_t s1, Set_t s2)
9   Set_t set_difference(Set_t s1, Set_t s2)
```

1	$O(1)$	Stvara novi skup.
2	Vidi napomenu.	Uništava skup.
3	Vidi napomenu.	Ubacuje element u skup.
4	Vidi napomenu.	Uklanja element iz skupa.
5	Vidi napomenu.	Ispituje da li je element u skupu.
6	$O(n)$	Ispituje da li je `s1` podskup od `s2`.
7	$O(n)$	Vraća uniju skupova `s1` i `s2`.

8	$O(n)$	Vraća presjek skupova `s1` i `s2`.
9	$O(n)$	Vraća razliku skupova `s1` i `s2`.

Napomena: Ovisno o implementaciji, navedene operacije imaju sljedeću složenost:

- $O(1)$ za implementaciju bit-vektorom
- $O(n)$ za implementaciju vezanom listom

1.2.4 dictionary

Apstrakcija rječnika: *neuređeni* skup za koji nisu definirani pojmovi podskupa, unije, presjeka i razlike.

- Tipovi:
  - `Dictionary_t`: tip rječnika u sučelju
  - `Element_t`: tip elementa u rječniku
- Implementacija pomoću:
  - hash tablice
  - (balansiranog) binarnog stabla traženja
- Operacije:

```
1 Dictionary_t dictionary_new(void)
2 void dictionary_delete(Dictionary_t d)
3 void dictionary_insert(Dictionary_t d, Element_t e)
4 void dictionary_remove(Dictionary_t d, Element_t e)
5 bool dictionary_is_member(Dictionary_t d, Element_t e)
6 bool dictionary_is_empty(Dictionary_t d)
```

1	$O(1)$	Stvara novi rječnik.
2	Vidi nap. 1.	Uništava rječnik.
3	Vidi nap. 2.	Ubacuje element u rječnik.
4	Vidi nap. 2.	Uklanja element iz rječnika.
5	Vidi nap. 2.	Ispituje da li je element u rječniku.
6	$O(1)$	Ispituje da li je rječnik prazan.

Napomena 1: Ovisno o implementaciji, operacija ima sljedeću složenost:

- $O(1)$ za implementaciju hash tablicom
- $O(n)$ za implementaciju (balansiranim) binarnim stablom

Napomena 2: Ovisno o implementaciji, navedene operacije imaju sljedeću složenost:

- $O(n)$ za binarno stablo
- $O(\log n)$ za balansirano binarno stablo
- očekivano $O(1)$ za hash tablicu

1.2.5 list

Apstrakcija niza elemenata s definiranim poretkom.

- Tipovi:
  - `List_t`: tip liste u sučelju
  - `Element_t`: tip elementa u sučelju
- Implementacija pomoću:
  - niza
  - vezane liste
- Operacije:

```
1 List_t list_new(void)
2 void list_delete(List_t l)
3 void list_insert(List_t l, Element_t e)
4 void list_remove(List_t l)
5 Element_t list_element_get(List_t l)
6 void list_first(List_t l)
7 void list_last(List_t l)
8 bool list_next(List_t l)
9 bool list_prev(List_t l)
```

1	$O(1)$	Stvara novu listu.
2	Vidi nap.1	Uništava listu.
3	Vidi nap.2	Ubacuje element na trenutnu poziciju u listi.
4	Vidi nap.2	Uklanja element sa trenutne pozicije u listi.
5	$O(1)$	Vraća element na trenutnoj poziciji u listi.
6	$O(1)$	Postavlja listu na prvu poziciju.
7	$O(1)$	Postavlja listu na zadnju poziciju.
8	$O(1)$	Postavlja listu na sljedeću poziciju.
9	$O(1)$	Postavlja listu na prethodnu poziciju.

Napomena 1: Ovisno o implementaciji, operacija ima sljedeću složenost:

- $O(1)$ za implementaciju nizom
- $O(n)$ za implementaciju vezanom listom

Napomena 2: Ovisno o implementaciji, navedene operacije imaju sljedeću složenost:

- $O(n)$ za implementaciju nizom
- $O(1)$ za implementaciju vezanom listom

Međutim, ovakvim pristupom nije moguće *istovremeno* manipulirati sa dvije različite pozicije u listi. Tako nije moguće npr. napraviti bubble sort.

1.2.6 list + position

Apstrakcija liste s definiranom pozicijom u listi.

- Tipovi:

- `List_t`: tip liste u sučelju
- `Element_t`: tip elementa u sučelju
- `Position_t`: u sučelju, ovaj tip predstavlja poziciju u listi

- Implementacija pomoću: ovakvo sučelje predodređuje implementaciju dvostruko vezanom listom ili poljem.

- Operacije:

```
List_t list_new(void)
void list_delete(List_t l)
void list_insert(List_t l, Position_t p, Element_t e)
void list_remove(List_t l, Position_t p)
Element_t list_element_get(List_t l, Position_t p)
Position_t list_first(List_t l)
Position_t list_last(List_t l)
Position_t list_next(List_t l, Position_t p)
Position_t list_previous(List_t l, Position_t p)
```

Operacije imaju istu namjenu i složenost kao i u prethodnom slučaju. Razlika je u eksplicitnom specificiranju pozicije umjesto "trenutne pozicije". Time je omogućeno manipuliranje sa nekoliko različitih pozicija u listi.

1.2.7 list + iterator

Ideja modernih ATP biblioteka (pogotovo C++ STL-a) je *odvajanje implementacije* ATP-a i algoritama nad ATP-om. Ista implementacija algoritma treba biti iskoristiva za više različitih ATP-ova. Kako je u srži mnogih algoritama upravo prolaz po elementima ATP-a, uvodi se novi apstraktni tip.

Iterator Iterator je poseban tip podatka koji predstavlja poziciju u nekom ATP-u. Ovisno o operacijama koje iterator podržava razlikuju se "forward", "bidirectional" i "random access" iteratori.

Iteratorima se često zadaje *interval* elemenata u nekom ATP-u. Tada se vrlo praktičnom pokazala konvencija da je to *poluotvoreni* interval: prvi element je uključen, a posljednji *nije*.

- Tipovi:
  - `List_t`: tip liste u sučelju
  - `Element_t`: tip elementa u sučelju
  - `Iterator_t`: tip iteratora u listi

- Implementacija pomoću:
  - niza
  - vezane liste

- Operacije:

```
List_t list_new(void)
void list_delete(List_t l)
void list_insert(List_t l, Iterator_t p, Element_t e)
void list_remove(List_t l, Iterator_t p)
```

```
5    Iterator_t begin(List_t l)
6    Iterator_t end(List_t l)
```

1	$O(1)$	Stvara novu listu.
2	Vidi nap.1	Uništava listu.
3	Vidi nap.2	Ubacuje element na zadanu poziciju u listi.
4	Vidi nap.2	Uklanja element sa zadane pozicije u listi.
5	$O(1)$	Vraća prvu poziciju u listi.
6	$O(1)$	Vraća zadnju poziciju u listi.

Napomena 1: Ovisno o implementaciji, operacija ima sljedeću složenost:

- $O(1)$ za implementaciju nizom
- $O(n)$ za implementaciju vezanom listom

Napomena 2: Ovisno o implementaciji, navedene operacije imaju sljedeću složenost:

- $O(n)$ za implementaciju nizom
- $O(1)$ za implementaciju vezanom listom

Tip iteratora ima sljedeće sučelje:

```
1 void it_delete(Iterator_t i)
2 Element_t it_element_get(Iterator_t i)
3 void it_element_set(Iterator_t i)
4 bool it_next(Iterator_t i)
5 bool it_valid(Iterator_t i)
6 bool it_equal(Iterator_t i1, Iterator_t i2)
7 bool it_prev(Iterator_t i)
8 bool it_seek(Iterator_t i, Index_t n)
```

1	$O(1)$	Uništava iterator.
2	$O(1)$	Vraća element na poziciji na koju iterator pokazuje.
3	$O(1)$	Zamjenjuje element na poziciji sa novim. Ova operacija nije moguća na svim ATP-ovima (npr. na mapi).
4	Vidi nap.	Postavlja iterator na sljedeći element.
5	$O(1)$	Ispituje da li iterator pokazuje na ispravnu poziciju.
6	$O(1)$	Pokazuju li dva iteratora na istu poziciju.
7	Vidi nap.	Postavlja iterator na prethodni element.
8	Vidi nap.	Postavlja iterator na n-tu poziciju. Ova operacija uvodi *novi tip* indeksa: pozitivni cijeli broj.

Operacije 1-6 zajedničke su svim iteratorima, 7 je karakteristična za bidirectional iterator, a 8 je karakteristična za random access iterator.

Napomena: Složenost operacija stvarno ovisi o implementaciji ATP-a uz koji je iterator vezan.

1.2.8 map

Apstrakcija preslikavanja iz jednog skupa vrijednosti (*ključ*) u drugi skup. Najčešće se koristi za preslikavanje stringova u neke druge vrijednosti.

- Tipovi:

- `Map_t`: tip mape u sučelju
- `Iterator_t`: tip iteratora u mapi
- `Key_t`: tip ključa
- `Value_t`: tip vrijednosti
- `Element_t`: element mape je složen od dvije komponente: ključa i vrijednosti.

- Implementacija pomoću:
  - hash tablice
  - (balansiranog) binarnog stabla

- Operacije:

```
1 Map_t map_new(void)
2 void map_delete(Map_t m)
3 Iterator_t map_insert(Map_t m, Element_t e)
4 void map_remove(Map_t m, Key_t k)
5 Iterator_t map_find(Map_t m, Key_t k)
6 Iterator_t map_begin(Map_t m)
7 Iterator_t map_end(Map_t m)
```

1	$O(1)$	Stvara novu mapu.
2	Vidi nap.1	Uništava mapu.
3	Vidi nap.2	Ubacuje element sa ključem `e.key` i vrijednošću `e.value`.
4	Vidi nap.2	Briše element sa zadanim ključem.
5	Vidi nap.2	Pronalazi vrijednost sa zadanim ključem.
6	$O(1)$	Vraća iterator na "prvi" element.
7	$O(1)$	Vraća iterator na "zadnji" element.

Operacije nad iteratorom su iste kao i kod liste.

Napomena 1: Ovisno o implementaciji, operacija ima sljedeću složenost:

- $O(1)$ za implementaciju hash tablicom
- $O(n)$ za implementaciju (balansiranim) binarnim stablom

Napomena 2: Ovisno o implementaciji, navedene operacije imaju sljedeću složenost:

- očekivano $O(1)$ za hash tablicu
- $O(\log n)$ za balansirano binarno stablo

Implementacije traže parametriziranje operacija nad ključem (usporedba, hashiranje i sl.).

1.2.9 priority queue

Apstrakcija ATP-a u kojem elementi imaju "prioritete". Efikasno je stavljanje elementa te dohvat elementa s najvećim prioritetom.

- Tipovi:
  - `Pq_t`: tip prioritetnog reda (PQ)
  - `Key_t`: tip ključa

  - `Value_t`: tip vrijednosti
  - `Element_t`: element PQ-a je složen od dvije komponente: ključa i

- Implementacija pomoću:

  - sortirane liste
  - hrpe (heap)

- Operacije:

```
1 Pq_t pq_new(void)
2 void pq_delete(Pq_t p)
3 void pq_push(Pq_t p, Element_t e)
4 Element_t pq_top(Pq_t p)
5 void pq_pop(Pq_t p)
6 bool pq_empty(Pq_t p)
```

1	$O(1)$	Stvara novi PQ.
2	Vidi nap.1	Uništava red.
3	Vidi nap.2	Ubacuje element s prioritetom `e.key` i vrijednošću `e.value`.
4	$O(1)$	Vraća element s najvišim prioritetom.
5	Vidi nap.3	Briše element s najvišim prioritetom.
6	Ispituje da li je PQ prazan.	

Napomena 1: Ovisno o implementaciji, operacija ima sljedeću složenost:

- $O(n)$ za implementaciju sortiranom listom
- $O(1)$ za implementaciju heapom

Napomena 2: Ovisno o implementaciji, operacija ima sljedeću složenost:

- $O(n)$ za implementaciju sortiranom listom
- $O(\log n)$ za implementaciju heapom

Napomena 3: Ovisno o implementaciji, operacija ima sljedeću složenost:

- $O(1)$ za implementaciju sortiranom listom
- $O(\log n)$ za implementaciju heapom

2 Jednostavne strukture podataka i algoritmi

2.1 Vektor

Vektor je najjednostavnija struktura podataka slična C-ovom array tipu podatka. Kao i array, dopušta izravan pristup bilo kojem elementu pomoću cjelobrojnog indeksa i to u *konstantnom vremenu* ($O(1)$ složenost). Za razliku od arraya, nije fiskne veličine (i može provjeravati indekse tako da se ne pristupa izvan granica vektora).

Ključno svojstvo vektora (koje vrijedi i za array), uz izravan pristup elementima, jest da se "susjedni" elementi vektora nalaze na *uzastopnim memorijskim lokacijama*, kao da se nalaze u običnom arrayu. Iako ima i drugih struktura podataka koje imaju $O(1)$ pristup elementim, ovo svojstvo je jedinstveno za vektor.

Prvo se definira struktura vektora koja sadrži sve potrebne podatke. `elts` je pointer na niz elemenata, `size` uvijek sadrži indeks zadnjeg inicijaliziranog elementa u vektoru, a `alloc` je ukupno alocirana veličina za elemente vektora (koja je radi efikasnosti redovito veća od `size`).
Kako će se dozvoliti "rupe" u vektoru, programeru se mora dozvoliti specificiranje defaultne vrijednosti elementa koji se nikako drukčije ne inicijalizira. Ta vrijednost se specificira u `vector_new` i dodjeljuje `dflt` polju.[5]
`vector_resize` je pomoćna funkcija koja povećava alocirani prostor za elemente vektora te nove elemente inicijalizira na defaultnu vrijednost.
Korištenje defaultne vrijednosti se može izbjeći alociranjem *posebnog bit vektora* koji za svaki element vektora sadrži 0 (nije inicijaliziran) ili 1 (inicijaliziran je). Tada bi svako čitanje elementa iz vektora provjeravalo je li element prethodno inicijaliziran i dojavilo grešku ako nije.

```
typedef struct vector {
  Element_t *elts;    /* elementi vektora */
  Element_t dflt;     /* default vrijednost el. */
  unsigned int size;  /* broj elemenata */
  unsigned int alloc; /* alocirana velicina */
} *Vector_t;
static void vector_resize(Vector_t v)
{
  unsigned int i;
  v->elts = realloc(v->elts, v->alloc * sizeof(Element_t));
  for(i = v->size; i < v->alloc; i++)
    v->elts[i] = v->dflt;
}
Vector_t vector_new(Element_t dflt)
{
  Vector_t v = malloc(sizeof(struct vector));
  v->dflt = dflt;
  v->size = 0;
  v->alloc = 64;
  v->elts = NULL;
  vector_resize(v);
  return v;
}
```

Ova funkcija vraća trenutan broj elemenata u vektoru. Ne moraju svi biti inicijalizirani; ovo je zapravo najveći trenutno dozvoljeni indeks prilikom čitanja elementa.
Ova funkcija oslobađa sav prostor alociran za elemente vektora i samu strukturu vektora.

```
unsigned int vector_size(Vector_t v)
{
  return v->size;
}
void vector_delete(Vector_t v)
{
  free(v->elts);
  free(v);
```

[5] Za ime se namjerno koristi kratica jer je `default` ključna riječ u C-u.

```
}
```

Funkcija `vector_get` prvo provjerava standardnim `assert` makroom da li je indeks manji od broja elemenata u vektoru te ako je, vraća element. U suprotnom prekida izvršavanje programa.

Funkcija `vector_put` provjerava da li je indeks veći ili jednak od alocirane veličine, te ako je povećava veličinu arraya te elemente popunjava defaultnom vrijednošću. Zatim povećava veličinu arraya ako se element sprema na veći indeks nego je trenutni broj elemenata te na zadani indeks zapisuje element.

Ovo je izuzetno jednostavna implementacija gdje se programer ne treba brinuti o "ručnom" održavanju veličine vektora (svaki indeks automatski postaje valjan nakon umetanja elementa). Mana ovog pristupa je da se u slučaju buga može (neotkriveno) potrošiti velika količina memorije.

```
Element_t vector_get(Vector_t v, unsigned int i)
{
  assert(i < v->size);
  return v->elts[i];
}
#define MAX(a,b) ((a) > (b) ? (a) : (b))
void vector_set(
  Vector_t v, unsigned int i, Element_t e)
{
  if(i >= v->alloc) {
    v->alloc = MAX(3*v->alloc/2, i);
    vector_resize(v);
  }
  if(i >= v->size)
    v->size = i;
  v->elts[i] = e;
}
```

—— Primjer ——

`assert` makro deklariran je u standardnom `<assert.h>` headeru. Uzima jedan argument koji mora biti *logički uvjet*, te ako tijekom izvršavanja programa taj uvjet nije istinit, prekida izvršavanje programa i ispisuje poruku o grešci poput:

```
zvrba@zax:/tmp$ ./a.out
a.out: z.c:5: f: Assertion 'a < b' failed.
```

u poruku je uključeno ime source datoteke i linija gdje je `assert` pozvan (`z.c` na liniji 5), ime funkcije u kojoj se desila greška (`f`) te sam uvjet koji nije bio zadovoljen.

Ako se prilikom kompajliranja programa definira makro `NDEBUG`[6] (njegovo ime je također propisano ANSI C standardom), tada su sve `assert` provjere *isključene*.

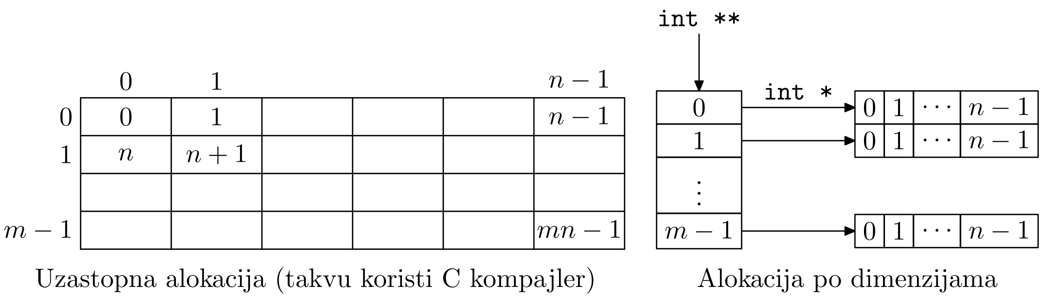
2.2 Višedimenzionalni nizovi

C jezik dozvoljava deklariranje 2-, 3- i višedimenzionalnih arraya, poput: `int a[2][3]`. Međutim deklariranje višedimenzionalnh arraya sa *varijabilnim* dimenzijama je problematično i u potpunosti je podržano tek u C99 standardu.[7]

Slika 2.1 prikazuje organizaciju dvodimenzionalnog niza veličine $m \times n$ u memoriji (m je broj redaka, a n broj stupaca). Element na lokaciji (i, j) se nalazi na *jednodimenzionalnom* indeksu $in + j$. Ova formula se može generalizirati i na višedimenzionalne nizove.

[6] Na bilo kakvu vrijednost. Čak mu i ne treba dodijeliti vrijednost; dovoljno je da je definiran.

[7] A čak ni tamo nije specificirano gdje se alocira prostor za array varijabilne dimenzije koji je lokalna varijabla funkcije. U praksi se lokalne varijable spremaju na stacku, no stack prostor je obično ograničen. Također se ne specificira što se desi ako nema dovoljno memorije da se alocira array zadane veličine.



Slika 2.1 Organizacija 2D niza u memoriji

Drugi način za alokaciju višedimenzionalnih nizova prikazan je na **slici 2.1**. Funkcije za alokaciju i dealokaciju su kompliciranije nego u prethodnom slučaju, ali je pristup tako alociranom dvodimenzionalnom arrayu jednostavniji: može se koristiti sintaksa poput `a[1][2]`. Kod ovakvog pristupa elementu ima više pristupa memoriji (točno onoliko koliko ima dimenzija; u prethodnom slučaju je uvijek jedan pristup), ali manje računanja.
Druga prednost ovakve organizacije arraya je ta što ne moraju svi retci biti jednake duljine.

```
int **alloc_array(int m, int n)
{
  int **m = malloc(m * sizeof(int*)), i;
  for(i = 0; i < m; i++)
    m[i] = malloc(n*sizeof(int));
}
void free_array(int **a, int m, int n)
{
  int i;
  for(i = 0; i < m; i++) free(a[i]);
  free(a);
}
```

—— Primjer

Uzmimo sljedeći kod:

```
int a1[2][3];
int *a2 = malloc(6*sizeof(int));
```

U oba slučaja se rezervira mjesto u memoriji za 6 uzastopnih `int`-ova. U prvom slučaju se element na indeksu $(1, 2)$ dohvaća izrazom `a[1][2]`, a u drugom izrazom poput `a2[1*3+2]` što je element `a2[5]`. Kod sličan ovome generira kompajler kada prevodi izraz poput `a[1][2]`.

2.3 Stack

Kao što je već rečeno u **odjeljku 1.2.1**, najjednostavnije implementacije su pomoću niza i liste.

2.3.1 Array

U ovoj implementaciji se početno za stack alocira niz od `N` (npr. 64) elemenata nekog tipa, te varijabla tipa `int` koja pokazuje na element koji je trenutno na vrhu stacka. Sve to zajedno se stavlja u jednu strukturu.

```
typedef struct stack {
  Element_t *elts;    /* elementi stacka */
  unsigned int size; /* alocirana velicina */
  int top;           /* vrh stacka */
} *Stack_t;
```

Uzet će se da je stack *prazan* ako je `top == -1`. Element `size` pamti trenutnu količinu memorije alociranu nizu `elts`. To je potrebno ako se želi da stack dinamički raste kada ponestane mjesta –

obično se alocirana veličina niza povećava za faktor 1.5 (`size *= 1.5`) i tada se sa `realloc` alocira novi (veći) prostor.

Alokaciju strukture stacka radi funkcija `stack_new`. Budući da je `Stack_t` tip pointera, moramo funkciji `malloc` dati veličinu stvarne strukture, a to je `sizeof(struct stack)`. `typedef` tipovi i tagovi struktura su različiti i nema nikakvih problema ako imaju isto ime.

```
Stack_t stack_new(void)
{
  Stack_t s = malloc(sizeof(struct stack));
  s->size = 64; /* pocetna velicina */
  s->top  = -1; /* na pocetku je prazan */
  s->elts = malloc(s->size * sizeof(Element_t));
}
```

Dealokaciju stacka radi funkcija `stack_delete`. Prvo se oslobađa memorija koju zauzima niz, a zatim i sama struktura. Obrnutim redoslijedom se *ne može* dealocirati memorija jer se ne smije pristupiti memoriji na koju pokazuje pointer nakon što se napravi `free`.

```
void stack_delete(Stack_t s)
{
  free(s->elts);
  free(s);
}
```

Funkcija `stack_push` stavlja element na stack. Ukoliko nema mjesta u nizu da `push` stavi novi element na stack, došlo je do *stack overflowa*. Tada se može prekinuti program ili realocirati veći niz.

```
void stack_push(Stack_t s, Element_t e)
{
  if(++s->top >= s->size) {
    /* STACK OVERFLOW */
  }
  s->elts[s->top] = e;
}
```

Funkcija `stack_pop` uzima element s vrha stacka. Također, ako se pokuša napraviti `pop` sa praznog stacka, došlo je do *stack underflowa* – to je ponekad greška, a ponekad legitimna operacija.
`pop` vraća true (1)[8] ukoliko je bilo elemenata na stacku; vraća false (0) ukoliko je došlo do stack underflowa.

```
int stack_pop(Stack_t s)
{
  if(s->top < 0) {
    /* STACK UNDERFLOW */
    return 0;
  }
  --s->top;
  return 1;
}
```

Funkcija `stack_empty` vraća true ako je stack prazan.

```
int stack_empty(Stack_t s)
{
  return s->top >= 0;
}
```

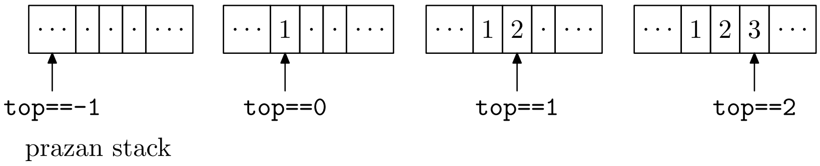
Funkcija `stack_top` vraća element na vrhu stacka. Ako je stack prazan, vratit će nedefiniranu vrijednost.

```
Element_t stack_top(Stack_t s)
{
  if(s->top >= 0) {
    return s->elts[s->top];
  }
}
```

[8] U C-u se istinom smatra sve što je različito od 0.

—— Primjer

Slika prikazuje sadržaj varijable `top` (pored strelice) i sadržaj niza nakom umetanja elemenata 1, 2, 3 (tim redom) na stack veličine 3 elementa. Točka označava prazno mjesto u nizu.



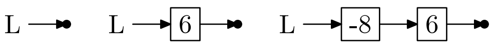
Slika 2.2 Stavljanje elemenata na stack

2.3.2 Jednostruko vezana lista s jednim krajem

Ovakva lista podržava umetanje i brisanje elemenata samo s jednog kraja i upravo je pogodna za implementaciju stacka koji ima upravo takav način pristupa elementima.

—— Primjer

Na slici je prikazano dodavanje dva elementa (prvo 6, zatim -8) u inicijalno praznu listu (L je pointer na početak liste; krug označava NULL pointer).



Slika 2.3 Dodavanje elemenata u listu

Kod za ovakav stack je iznimno jednostavan. Prvo definiramo strukturu povezane liste koja će držati jedan element stacka i funkciju za stavljanje na stack.

```
typedef struct stack {
  Element_t elt;
  struct stack *next;
} *Stack_t;
```

Budući da je u ovoj implementaciji cijeli stack predstavljen pointerom na vrh stacka (varijabla `s` u donjem primjeru), `stack_push` funkcija mora uzeti *pointer* na vrh stacka kako bi mogla promijeniti varijablu (u primjeru `s`) da pokazuje na novi element na vrhu stacka.

```
void stack_push(Stack_t *stk, Element_t elt)
{
  struct stack *n =
    malloc(sizeof(struct stack));
  if(!n) {
    /* STACK OVERFLOW */
  }
  n->elt = elt; n->next = *stk; *stk = n;
}
```

Funkcija `stack_pop` je malo složenija: prvo u pomoćnu varijablu treba spremiti prethodni pointer na vrh stacka, vrh stacka pomaknuti jedan element dalje i tek onda osloboditi stari vrh stacka. Sve to zato što nije dozvoljeno pristupati memoriji na koju pointer pokazuje nakon što se napravi `free`.

```
int stack_pop(Stack_t *stk)
{
  if(*stk) {
    struct stack *top = *stk;
    *stk = (*stk)->next;
    free(top);
    return 1;
  }
  /* STACK UNDERFLOW */
```

```
  return 0;
}
```

Funkcija `stack_top` vraća element na vrhu stacka ukoliko nije prazan.

```
Element_t stack_top(Stack_t stk)
{
  if(stk) return stk->elt;
}
```

Funkcija `stack_empty` jednostavno provjerava da li je vrh stacka NULL pointer (ako je, tada je stack prazan).

```
int stack_empty(Stack_t stk)
{
  return stk == NULL;
}
```

— PRIMJER —

Ovo je jednostavan primjer upotrebe stacka sa cijelim brojevima. Primijetite kako se funkcijama `stack_push` i `stack_pop` prosljeđuje *adresa* od `s`.

```
stack s = NULL;    /* prazan stack */
stack_push(&s, 1);
stack_push(&s, 2);
/* .. nesto se radi .. */
printf("vrh stacka je %f\n", stack_top(s));
stack_pop(&s);
```

2.4 Liste

Lista je linearno organizirana struktura podataka. Najjednostavnija implementacija je pomoću array-a. Međutim, tada imamo problem pri umetanju elemenata u sredinu liste: svi elementi iza mjesta na koje umećemo novi element se moraju pomaknuti jedno mjesto dalje, što je $O(n)$ operacija.

U nekim jezicima, npr. LISP-u, liste su glavna struktura podataka. Ali LISP liste su mnogo općenitije: te liste su *rekurzivne* strukture definirane na sljedeći način:

- *prazna lista* je lista (označava se sa `nil`)
- lista je *par* `(car,cdr)` gdje je prvi element (`car`) bilo što (atom (npr. broj), ali može biti i lista), a drugi element (`cdr`) je *lista*.

Može se pokazati da su ovako definirane liste dovoljno općenite da prikažu bilo kakvu složenu strukturu podataka, uključujući kružne liste, stabla, grafove, itd...

Liste se mogu implementirati na mnogo načina; ovdje će se pokazati neke od implementacija prema rastućoj složenosti implementacije. Najjednostavniji primjer vezane liste dan je već u **odjeljku 2.3.2** gdje se jednostruko povezana lista koristila za implementaciju stacka.

2.4.1 Jednostruko povezana lista s dva kraja i red (queue)

Za ovu vrstu liste potrebna je struktura čvora liste, koja je ista (osim tipa podatka) kao i struk-

```
struct node {
  Element_t data;    /* neki podaci */
```

tura za element stacka iz prethodnog odjeljka.

```
  struct node *next;
};
```

Struktura liste sadrži pointere na prvi i zadnji čvor liste. Lista je prazna kada je `begin == end == NULL`.

```
typedef struct list {
  struct node *begin;
  struct node *end;
} *List_t;
```

Funkcija `list_new` alocira praznu listu.

```
List_t list_new(void)
{
  List_t l = malloc(sizeof(struct list));
  l->begin = l->end = NULL;
  return l;
}
```

Funckija `list_delete` briše sve elemente iz liste, a zatim i samu strukturu liste. Redoslijed naredbi unutar `while` je *bitan*: kada bi se zamijenio (ili kada bi se pokušalo maknuti pomoćnu varijablu `tmp`), tada bi se u `n = n->next` pristupalo oslobođenoj memoriji.
Uvjet petlje je ispravan jer zadnji element liste pokazuje na NULL pointer.

```
void list_delete(List_t l)
{
  struct node *n = l->begin, *tmp;
  while((tmp = n) != NULL) {
    n = n->next;
    free(tmp);
  }
  free(l);
}
```

Budući da lista ima dva kraja, sada možemo stavljati elemente i na početak i na kraj. Tako imamo funkciju `list_insert_begin` koja stavlja element na početak liste.
Primijetite da se ubacivanje u praznu listu mora obrađivati kao poseban slučaj. To je potrebno i pri ubacivanju na kraj liste.

```
void list_insert_begin(List_t l, Element_t elt)
{
  struct node *n = malloc(sizeof(struct node));
  n->data = elt;
  if(l->begin == NULL) {
    /* umetanje u praznu listu */
    n->next = NULL;
    l->begin = l->end = n;
  } else {
    n->next = l->begin;
    l->begin = n;
  }
}
```

Funkcija `list_insert_end` stavlja element na kraj liste. Budući da je novi element koji se ubacuje (`n`) uvijek zadnji, njegov `next` pointer se uvijek postavlja na NULL, neovisno o tome je li lista prazna ili nije.

```
void list_insert_end(List_t l, Element_t elt)
{
  struct node *n = malloc(sizeof(struct node));
  n->data = elt;
  /* uvijek je zadnji */
  n->next = NULL;
  if(l->begin == NULL) {
    /* umetanje u praznu listu */
    l->begin = l->end = n;
  } else {
    l->end->next = n;
    l->end = n;
  }
```

```
}
```

Neposredni prethodnik i sljedbenik Neka su `n1` i `n2` dva čvora liste (tipa `struct node`) za koje vrijedi relacija `n1->next == n2`. Tada je `n1` **neposredni prethodnik** od `n2`, a `n2` je **neposredni sljedbenik** od `n1`.

Prethodnik i sljedbenik Neka za dva čvora n_0 i n_k postoji niz čvorova $n_1, n_2, \ldots, n_{k-1}$ tako da je za svaki $0 < i \leq k$, čvor n_{i-1} neposredni prethodnik čvora n_i. Tada je čvor n_0 **prethodnik** čvora n_k, odn. čvok n_k je **sljedbenik** čvora n_0.

Iako se elementi sada mogu ubacivati i na početak i na kraj liste, *brisanje* elemenata je ograničeno. Ukoliko se ima pointer na neki element, tada se može izbrisati jedino njegov neposredni sljedbenik. Pri tome također treba paziti da se popravi pointer na zadnji element liste (`end`) ukoliko je to potrebno.

2.4.1.1 Broj elemenata i podliste

Osim što je neefikasno traženje prethodnika elementa, jednako je neefikasno *prebrojavanje* elemenata u listi (također se mora prijeći cijela lista). Moguće je u `struct list` dodati još jedan član koji će brojati koliko ima elemenata u listi i popravljati ga prilikom svakog umetanja ili brisanja iz liste. Takav pristup pak ima nedostatak jer je teško održati konzistentno stanje brojača, što će se vidjeti iz sljedećeg razmatranja.

Podlista Neka imamo listu `L` i dva pointera na čvorove unutar te liste (npr. `l1` i `l2`; neka je `l1` prethodnik od `l2`). Ta dva čvora definiraju **podlistu** liste `L`.

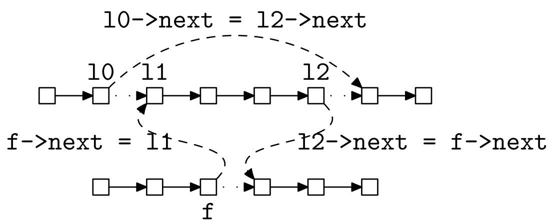
U kontekstu prebrojavanja elemenata liste, čak i kada bi znali broj elemenata cijele liste, *nikako* ne možemo znati koliko elemenata ima dana podlista, osim da ih prebrojimo.

Dakle, traženje prethodnika (i općenito, k-tog člana liste) i prebrojavanje elemenata liste su $O(n)$ operacije (kod nizova su to $O(1)$ operacije, n je broj elemenata). Međutim, *brisanje i umetanje podliste (bilo gdje u listi)* su $O(1)$ operacije (kod nizova su to $O(n)$ operacije).

Neka je `l0` neposredni prethodnik elementa `l1`. Također neka je `F` neka druga lista te `f` neki čvor unutar liste `F`. Želimo *izbrisati* elemente između `l1` i `l2` iz liste `L` i *umetnuti* ih u listu `F` *iza* elementa `f`.

```
/* izbacivanje iz L */
l0->next = l2->next;
/* umetanje u F */
l2->next = f->next; f->next = l1;
```

Ova operacija se ponekad naziva (engl.) **splice** i očito ima složenost $O(1)$ jer ne ovisi o broju elemenata ni u kojoj (pod)listi. Grafički je prikazana na **slici 2.4** gdje su točkastim linijama prikazani *izbrisani*, a iscrtkanim linijama *novi* linkovi.



Slika 2.4 splice dvije liste

2.4.1.2 Red (queue)

Pomoću ovakve liste može se lako napraviti struktura podataka **red**[9]. Tako se zove jer odgovara redovima čekanja u npr. banci i radi po FIFO[10] principu. Podržava operacije stvljanja u red (enqueue) i uzimanja iz reda (dequeue).

Element se stavlja na *kraj* liste.

```
void enqueue(List_t l, Element_t elt)
{
  list_insert_end(l, elt);
}
```

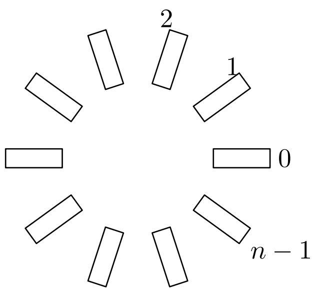
Element se uzima sa *početka* liste i usput se dealocira čvor (u kodu se ne provjerava mogućnost praznog queuea, što odgovara praznoj listi).
Ukoliko je u redu bio samo jedan element (koji je upravo uzet), `begin` pointer postaje NULL. U tom se slučaju, zbog prije navedene konvencije za praznu listu, i `end` pointer postavlja na NULL.

```
Element_t dequeue(List_t l)
{
  struct node *n = l->begin;
  Element_t retval = n->data;
  l->begin = l->begin->next;
  if(l->begin == NULL) l->end = NULL;
  free(n);
  return retval;
}
```

Osim vezanom listom, red se lako može implementirati nizom; imamo dva indeksa: `beg` koji pokazuje na početak reda i `end` koji pokazuje na kraj reda. Neka je `que` npr. cjelobrojni niz koji sadržava elemente reda. Uzima se da je red *prazan* ukoliko vrijedi `beg == end`; i tako u početku obje varijable moraju biti inicijalizirane.

```
void enqueue(Element_t x)
{
  que[end++] = x;
}
Element_t dequeue(void)
{
  return que[beg++];
}
```

Međutim, ovakva implementacija ima ozbiljan nedostatak: indeksi `beg` i `end` mogu postati *proizvoljno veliki*, mada možda red niti u jednom trenutku nema više od npr. jednog elementa – koliko god mjesta alocirali za niz `que`, nikada neće biti dovoljno. Takav slučaj se može desiti uzastopnim stavljanjem u red i uzimanjem jednog elementa.
Zbog toga se niz `que` promatra kao kružni niz (**slika 2.5**), a operacije zbrajanja se rade mod n.



Slika 2.5 Kružni red kao niz

U ovom slučaju se može desiti da nema mjesta za dodavanje novog elementa u red, o čemu treba voditi računa (kao što je pokazano gore u kodu). Budući da `beg == end` znači prazan red, u niz veličine n (promatran kao red) možemo spremiti $n-1$ elemenata.

```
void enqueue(Element_t x)
{
  end = (end+1)%n;
  if(end == beg) {
    /* nema mjesta */
  } else {
```

[9] engl. queue
[10] First In, First Out

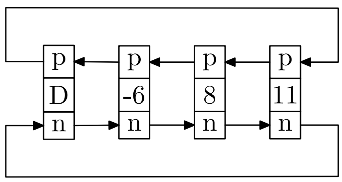
Naime, kada bi u nizu bilo n elemenata, n-ti bi se mogao dohvatiti samo kada je `beg == end` i ne bi se znalo kada je red prazan. U praksi ovo ograničenje ne stvara poteškoće.

```
    que[end] = x;
  }
}
Element_t dequeue(void)
{
  if(beg == end) {
    /* prazan red */
  } else {
    beg = (beg+1)%n;
    return que[beg];
  }
}
```

Broj elemenata u redu može se izračunati kao razlika indeksa kraja i početka. No, treba paziti jer `beg` (indeks početka) može biti *veći* od `end` (indeks kraja). `n` je veličina niza `que`.

```
int que_size(void)
{
  int s = end - beg;
  return s + (s<0 ? n : 0);
}
```

2.4.2 Dvostruko povezana lista s "dummy" čvorom; kružna lista



Slika 2.6 Kružna lista (D je dummy čvor)

Kao što je pokazano u **odjeljku 2.4.1**, prazna lista se mora uvijek promatrati kao poseban slučaj. Stoga je zgodno uvesti konvenciju da lista uvijek ima *bar jedan, tzv. dummy čvor*. Taj čvor ne sadrži nikakav koristan podatak.

Također je zgodno staviti da posljednji element liste pokazuje na dummy čvor, a dummy čvor pokazuje na prvi stvarni element liste. U praznoj listi dummy čvor pokazuje *na sebe*. Tako se dobiva *kružna (cirkularna) lista*.

Konačno, zgodno je uvesti i pointere na *prethodni* element, a ne samo na sljedeći kako bi se olakšao prolazak po listi. Tako se dobiva kružna dvostruko povezana lista s dummy čvorom čija je struktura prikazana na **slici 2.6**.

U dvostruko povezanoj listi svaki čvor, uz podatak, mora imati i dva pointera: na prethodni i sljedeći član liste (oznake p i n na **slici 2.6**). Budući da u dvostruko povezanoj kružnoj listi iz dummy čvora možemo dobiti i prvi i zadnji element liste, lista se može poistovijetiti s pointerom na upravo taj dummy čvor.

```
typedef struct node {
  Element_t data;
  struct node *next, *prev;
} *List_t;
```

Prilikom alokacije liste alocira se samo dummy čvor i pri tome se uredi tako da on pokazuje sam na sebe. Rutina za dealokaciju je vrlo slična rutini `list_delete` iz **odjeljka 2.4.1** uz jednu vrlo bitnu izmjenu: kako je lista kružna, *niti jedan pointer nije NULL*. Zato uvjet u `while` petlji koji glasi `!= NULL` treba promijeniti u `!= l` (gdje je `l` pointer na dummy čvor koji vrati funkcija

```
List_t list_new(void)
{
  List_t l = malloc(sizeof(struct list));
  l->prev = l->next = l;
  return l;
}
```

list_new).
Također, treba izbaciti zadnji free(l) jer u ovom slučaju lista nije zasebna struktura već je poistovijećena sa pointerom na svoj dummy čvor koji je oslobođen u prvoj iteraciji petlje.

Budući da je lista dvostruko povezana, lako je implementirati umetanje elementa u listu iza ili ispred *proizvoljnog* čvora. Zbog dvostrukih veza, potrebno je implementirati samo jednu od te dvije operacije, druga trivijalno proizlazi zbog cirkularnosti i dvostrukih veza. Ovdje će se implementirati list_insert_after. Funkcija uzima pointer na čvor n iza kojeg treba umetnuti element elt.
Za umetanje na *početak (odn. kraj)* liste treba funkcijama proslijediti pointer na dummy čvor (tj. pointer na samu listu).
Brisanje elementa je također lako napraviti. Ova funkcija briše i dealocira zadani čvor te vraća podatak spremljen u njemu. Ovoj funkciji se *ne smije* proslijediti pointer na dummy čvor.

```
void list_insert_after(struct node *n, Element_t elt)
{
  struct node *nn = malloc(sizeof(struct node));
  nn->data = elt;
  nn->next = n->next; n->next = nn;
  nn->prev = n; nn->next->prev = nn;
}
void list_insert_before(struct node *n, Element_t elt)
{
  list_insert_after(n->prev, elt);
}

Element_t list_remove(struct node *n)
{
  Element_t retval = n->data;
  n->prev->next = n->next;
  n->next->prev = n->prev;
  free(n);
  return retval;
}
```

2.5 Bit vektor

Bit vektor implementacija je pogodna za predstavljanje skupa cijelih brojeva. Za označavanje prisutnosti elementa u skupu dovoljan je jedan *bit:* taj bit je 1 ako je element prisutan u skupu, 0 ako nije.[11]

Uz pretpostavku da unsigned char ima 8 bitova[12], potreban je niz od $\lceil n/8 \rceil$ unsigned char elemenata za skup od n elemenata. Najvažnije operacije su *umetanje* jednog elementa n u skup i *brisanje* tog elementa iz skupa. n/8 je redni broj elementa u nizu unisgned char elemenata u kojem se nalazi n-ti po redu bit. n&7 je redni broj *bita* u bajtu koji odgovara n-tom bitu ukupno po redu.

Operacije postavljanja, brisanja i provjere n-tog bita u nizu unsigned char elemenata rade *makroi* BIT_SET, BIT_CLR i BIT_ISSET. Njihova je namjena da služe isključivo kao *ispomoć* pri pisanju funkcija ATP-a set.

```
#define BIT_SET(v, n)   (v[(n)>>3] |= 1U << ((n) & 7))
#define BIT_CLR(v, n)   (v[(n)>>3] &= ~(1U << ((n) & 7)))
#define BIT_ISSET(v, n) (v[(n)>>3] & (1U << ((n) & 7)))
```

[11] Iz ovoga odmah proizalzi da *partitivni skup* skupa od n elemenata ima 2^n elemenata.
[12] Iako malo vjerojatno, može se desiti da bude i druge veličine. Točna veličina se nalazi u konstanti CHAR_BIT iz standardnog headera limits.h.

Ostale skupovne operacije se direktno mapiraju na C bitovne operatore. Uniji i presjeku odgovaraju bitovni operatori “or” (`|`) i “and” (`&`). Operaciji komplementa skupa odgovara bitovni “not” operator (`~`).

Razlika se temelji na sljedećem identitetu: $A \backslash B = A \cap \overline{B}$. Za relaciju podskupa vrijedi sljedeći identitet: $A \cap B = A \Leftrightarrow A \subseteq B$.

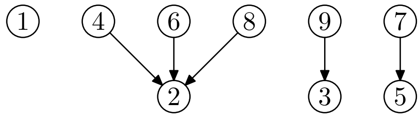
2.6 Klase ekvivalencije

Za neke primjene (npr. minimizacija konačnog automata) potrebno je brzo ispitati da li su dva elementa iz nekog skupa u istoj klasi ekvivalencije. Sve klase ekvivalencije čine *particiju* skupa u podskupove. Opisat će se jednostavan način za grupiranje elemenata u klase ekvivalencije ukoliko se oni na jednostavan način mogu obrojčiti (označiti prirodnim brojevima).

Particija Neka je A skup te neka su $A_1, A_2, \ldots, A_n$ podskupovi skupa A. Ako su svi skupovi u parovima disjunktni (za sve $1 \leq i < j \leq n$ vrijedi $A_i \cap A_j = \emptyset$) i ako je $A_1 \cup A_2 \cup \ldots \cup A_n = A$ kažemo da skupovi A_i, $1 \leq i \leq n$ čine particiju skupa A.

—— PRIMJER

Za skup $A = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ mogu se napraviti sljedeće proizvoljne klase ekvivalencije: $A_1 = \{1\}$, $A_2 = \{2, 4, 6, 8\}$, $A_3 = \{3, 9\}$, $A_4 = \{5, 7\}$. Iste te klase ekvivalencije mogu se prikazati sa skupom stabala na **slici 2.7**. Iz svake klase ekvivalencije izabran je jedan *predstavnik*; svi ostali elementi iste klase ekvivalencije su “djeca” predstavnika.



Slika 2.7 Klase ekvivalencije

Ovakav skup stabala lako se može preslikati u jednodimenzionalan cjelobrojni niz (npr. `equiv`) tako da elementu s brojem `i` odgovara element niza `equiv[i]`. Elementi niza se popunjavaju prema sljedećim pravilima:

- Ako je element `p` *predstavnik* klase ekvivalencije, stavlja se `equiv[p] = p`.
- Inače, neki element `i` *nije* predstavnik klase ekvivalencije; on se nalazi u istom skupu sa svojim predstavnikom `p`. Za takav element se stavlja `equiv[i] = p`.

—— PRIMJER

Klasama ekvivalencije sa **slike 2.7** odgovara niz `equiv`. Element 1 se sam nalazi u klasi ekvivalencije pa je sam svoj predstavnik.

i	1	2	3	4	5	6	7	8	9
equiv[i]	1	2	3	2	5	2	5	2	3

Funkcija `equiv_init` alocira niz od `n` cijelih brojeva – takav niz može predstavljati particiju skupa od ukupno `n` elemenata. Niz se inicijalizira tako da je svaki element u posebnoj klasi ekvivalencije.

```
int *equiv_init(int n)
{
  int i, *e = malloc(n * sizeof(int));
  for(i = 0; i < n; i++) e[i] = i;
  return e;
}
```

Osnovna operacija kod klasa ekvivalencije je traženje predstavnika. Funkcija je napisana kao petlja jer neki element ne mora biti direktno

```
int equiv_parent(int *equiv, int i)
{
  int p;
```

povezan sa svojim predstavnikom; takva veza može biti i indirektna.

```
  for(p = equiv[i]; i != p; i = p, p = equiv[p])
    ;
  return p;
}
```

Element `e` se može staviti u istu klasu ekvivalencije sa nekim već postojećim elementom `e0`. Taj element ne mora biti predstavnik klase; on se lako nađe. Ako je element bio u nekoj drugoj klasi ekvivalencije, tada se briše iz nje.
Ako je taj element bio *predstavnik* neke klase ekvivalencije, i svi ostali elementi iz te klase prelaze u novu klasu ekvivalencije.

```
void equiv_add(int *equiv, int e0, int e)
{
  equiv[e] = equiv_parent(equiv, e0);
}
```

Lako je provjeriti da li se dva elementa `a` i `b` nalaze u istoj klasi ekvivalencije: oni imaju istog predstavnika.

```
int equiv_eq(int *equiv, int a, int b)
{
  return
    equiv_parent(equiv, a) ==
    equiv_parent(equiv, b);
}
```

Kada se elementi premještaju iz jedne u drugu klasu ekvivalencije, pogotovo ako se premještaju predstavnici, može se desiti da neki element nije više direktno povezan sa svojim predstavnikom. Tada se usporava traženje predstavnika (funkcija `equiv_parent`). Zbog toga se ponekad klasa ekvivalencije "komprimira" tako da su svi elementi izravno povezani sa svojim predstavnicima. `n` je ukupan broj elemenata skupa.

```
void equiv_compress(int *equiv, int n)
{
  while(n--) equiv[n] = equiv_parent(n);
}
```

Struktura podataka i algoritmi opisani u ovom odjeljku mogu se naći u drugoj literaturi pod nazivom "union find".

2.7 Algoritmi nad sortiranim nizovima

2.7.1 Skupovne operacije

U ovom odjeljku će se opisati unija i presjek nad dva sortirana niza u kojima se *elementi ne ponavljaju (dakle, skupovi)*; rezultat je opet sortirani niz. Oba algoritma su složenosti $O(n)$ i rade samo jedan prolaz kroz svaki niz.

Funkcije imaju sljedeće argumente i povratnu vrijednost:

- `a`, `an`: prvi niz i broj elemenata
- `b`, `bn`: drugi niz i broj elemenata
- `c`: rezultat; mora biti prealociran. Za uniju treba biti barem veličine $an + bn$. Za presjek treba biti barem veličine $\min(an, bn)$.
- Povratna vrijednost: broj elemenata u rezultatu.

2.7.1.1 Unija

U svakom prolazu petlje uspoređuju se elementi na početku dva niza. Manji element iz dva niza se kopira u rezultantni niz. Za jedan se pomiče početak niza u kojem je nađen manji element. Ako se u nekom trenutku na početku oba niza nađu jednaki elementi, u rezultantni niz se taj element kopira samo *jednom*, a pomiču se počeci *oba* niza. Ako su nizovi različite duljine, iza petlje se kopiraju preostali elementi iz jednog niza (svi su veći od najvećeg elementa u "potrošenom" nizu) na kraj rezultantnog niza. Na kraju se izračuna broj elemenata u rezultatu. Za sortiranje je potrebna modificirana operacija unije koja ispravno radi nad nizovima u kojima se elementi *ponavljaju*. Ako se element x u skupu A pojavljuje m puta, a u skupu B n puta, u rezultatu se mora pojaviti $m + n$ puta. Takav algoritam se zove *merge* (spajanje) i bit će opisan u **poglavlju 5**.

```
static unsigned int union_merge(
  Element_t *a, unsigned int an,
  Element_t *b, unsigned int bn,
  Element_t *c)
{
  unsigned int n;
  Element_t *rest, *c0 = c;
  while(an && bn) {
    if(*a < *b) {
      *c++ = *a++;
      --an;
    } else if(*a > *b) {
      *c++ = *b++;
      --bn;
    } else {
      *c++ = *a++; ++b;
      --an; --bn;
    }
  }
  n = an ? an : bn;
  rest = an ? a : b;
  memcpy(c, rest, n*sizeof(Element_t));
  return c - c0 + n;
}
```

2.7.1.2 Presjek

Operacija presjeka radi na sličan način kao operacija unije. Jedina razlika je što se u rezultantni niz stavljaju elementi koji su jednaki u oba niza. Kad se dođe do kraja kraćeg niza, odmah se izlazi van iz petlje jer nije moguće da se neki preostali elementi nalaze u rezultatu.

```
static unsigned int intersect(
  Element_t *a, unsigned int an,
  Element_t *b, unsigned int bn,
  Element_t *c)
{
  Element_t *c0 = c;
  while(an && bn) {
    if(*a < *b) {
      ++a; --an;
    } else if(*a > *b) {
      ++b; --bn;
    } else {
      *c++ = *a++; ++b;
      --an; --bn;
    }
  }
  return c - c0;
}
```

2.7.2 Binarno traženje

Binarno traženje je brzi algoritam traženja primjenjiv isključivo na *već sortirane* nizove: u nizu od n elemenata, treba najviše $\lg n$ koraka da pronađe element (ili ustanovi da ga nema). To je veliko ubrzanje u usporedbi sa običnim (linearnim) traženjem koje u najgorem slučaju treba n koraka da pronađe element.
Prikazani kod implementira binarno traženje u nizu cijelih brojeva od `N` elemenata. Vraća poziciju traženog elementa u nizu ili -1 ako element ne postoji.

```
int binsearch(int *arr, int N, int v)
{
  int l = 0, r = N-1, x;
  while(r >= l) {
    x = (l+r) / 2;
    if(v < arr[x]) r = x-1;
    else l = x+1;
    if(v == arr[x]) return x;
  }
  return -1;
}
```

—— Primjer ——

Rad algoritma će se prikazati na sljedećem nizu `arr`:

x	0	1	2	3	4	5	6	7	8	9
arr[x]	-7	-5	1	6	9	15	16	22	27	31

Tablica prikazuje sadržaj varijabli `l`, `r`, `x` i `arr[x]` pri traženju vrijednosti 22:

l	r	x	arr[x]	
0	9	4	9	$22 > 9$, stavlja se `l=x+1`
5	9	7	22	pronađen je traženi element; kraj

Ukoliko se traži nepostojeća vrijednost, npr. 2:

l	r	x	arr[x]	
0	9	4	9	$2 < 9$, stavlja se `r=x-1`
0	3	1	-5	$2 > -5$, stavlja se `l=x+1`
2	3	2	1	$2 > 1$, stavlja se `l=x+1`
3	3	3	6	$2 \neq 6$, `r==l`: element ne postoji; kraj

2.7.2.1 Standardna C implementacija

Iako jednostavan za napisati, binarno traženje je jako važan algoritam pa je stoga uključen u standardni C library kao funkcija `bsearch` u headeru `<stdlib.h>`. Njen prototip je sljedeći:

```
typedef int (*cmp_fn)(const void *a, const void *b);
void *bsearch(const void *key, const void *arr, size_t n,
  size_t sz, cmp_fn cmp);
```

- `key`: Pointer na varijablu koja sadrži vrijednost koju tražimo.
- `arr`: Pointer na prvi element niza u kojem se traži element.
- `n`: Broj elemenata u nizu.
- `sz`: Veličina *pojedinačnog* elementa.

- `cmp`: Pointer na funkciju usporedbe.
- Povratna vrijednost: Pointer na element unutar niza `arr` koji je jednak traženoj vrijednosti. Ukoliko ima više jednakih elemenata (tada moraju biti uzastopni jer je niz sortiran), nije definirano koji od njih vraća. Ako element ne postoji vraća NULL.

`cmp_fn` je typedef (ovdje stavljen isključivo radi jasnoće, nije definiran u C headerima) pointera na **funkciju usporedbe**. Funkcija usporedbe uzima pointere na dva elementa koja treba usporediti[13] i vraća:

- broj < 0 ako je `a<b`
- 0 ako je `a==b`
- broj > 0 ako je `a>b`

Točna vrijednost koju funkcija vraća *nije bitna*: bitan je samo predznak. Funkcija `bsearch` stane kada funkcija usporedbe vrati 0.

—— Primjer ——

Za usporedbu cijelih brojeva može se uzeti funkcija `cmp_int`. `cmp_int0` je čest način implementacije funkcije usporedbe za brojeve, no u općem slučaju je **neispravan**.
Za usporedbu stringova može se uzeti funkcija `cmp_str`. Funkcija `strcmp` iz `<string.h>` bi bila dobra funkcija usporedbe kada bi imala ispravan prototip (nema jer uzima dva `const char` pointer argumenta). Zbog toga ju moramo staviti unutar funkcije ispravnog prototipa. Cast nije potreban jer se `void` pointeri automatski implicitno castaju u bilo koji drugi pointer tip.

```
int cmp_int0(const void *a, const void *b)
{
  return *(int*)a - *(int*)b;
}
int cmp_int(const void *a, const void *b)
{
  int ai = *(int*)a, bi = *(int*)b;
  return ai > bi ? 1 : (ai < bi ? -1 : 0);
}
int cmp_str(const void *a, const void *b)
{
  return strcmp(a, b);
}
```

Funkcija `cmp_int0`, iako zgodna jer razlika cijelih brojeva daje upravo onakav predznak kakav treba vraćati funkcija usporedbe, je *neispravna*. Naime, u općem slučaju bilo kakvih ulaznih podataka može doći do aritmetičkog overflowa. Npr. `int` veličine 32 bita može držati brojeve u intervalu $[-2^{31}, 2^{31}-1]$. Za $a = -2^{31}+2$ i $b = 3$ imamo (prema pravilima računanja u dvojnom komplementu) $a - b = 2^{31} - 1 > 0$. Tako ispada da je negativan broj veći od pozitivnog – očigledno neispravan rezultat usporedbe.

Ovdje je prikazano traženje elementa u nizu cijelih brojeva `arr` veličine `N`. Funkcija bsearch vraća pointer (`elem`) na element unutar niza `arr`; ukoliko nas zanima indeks tog elementa unutar niza, može se dobiti pointer aritmetikom kao `elem - arr`.

```
int x = 3; /* vrijednost koja se trazi */
int *elem = bsearch(&x, arr, N, sizeof(int), cmp_int);
```

---

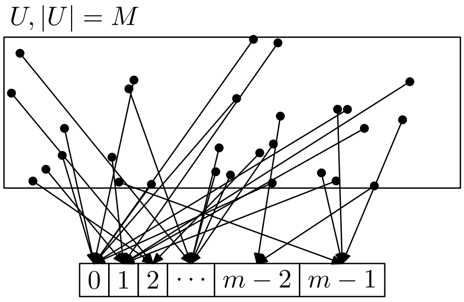
[13] Argumenti su `const void` pointeri pa ih treba unutar funkcije cast-ati na pointere željenog tipa.

3 Složenije strukture podataka

3.1 Hash

Hash je generalizacija običnog niza (koji se indeksira cjelobrojnim indeksima) na polje indeksirano proizvoljnim *ključevima* (koji mogu biti brojevi, nizovi znakova, i sl.). Također se implementira nizom, ali pozicija elementa u nizu ovisi o aritmetičkoj transformaciji ključa.

Ideja hasha je da se ulazni skup veličine M (koja može biti jako velika; veća od raspoložive memorije) svede na manji broj, m, pretinaca. Tu "kompresiju" radi hash funkcija (**slika 3.1**).



Slika 3.1 Preslikavanje hash funkcije

Hash funkcija Neka je U skup svih mogućih vrijednosti ključa, $|U| = M$, te neka je m veličina hash tablice (to se još zove i broj **pretinaca**). Hash funkcija h je preslikavanje iz skupa vrijednosti ključa u skup indeksa hash tablice: $h : U \to \{0, 1, \ldots, m-1\}$.

Sudar Ako dva ključa imaju istu vrijednost hash funkcije, kažemo da je nastao sudar (kolizija).

3.1.1 Hash funkcija

O hash funkciji jako ovisi brzina umetanja i traženja elemenata u hashu. Dobra hash funkcija treba raspršivati ključeve što slučajnije i uniformnije jer u tom slučaju ima najmanje kolizija. U idealnom slučaju su ubacivanje i traženje elementa složenosti $O(1)$.[14]

Dobra svojstva hash funkcija imaju *kriptografske hash funkcije* (poput SHA-1, MD5, RIPEMD160) i *checksum funkcije* (poput CRC16, CRC32, ADLER32). Međutim, njihova loša strana je da su *spore*, *daju preveliki rezultat* ili oboje. Npr. kriptografske hash funkcije daju rezultate 64-160 bitova, a redovito je potreban 32-bitni rezultat nakon kojega se još radi mod na veličinu tablice.

Sljedeće su se pokazale kao dobre i brze hash funkcije. x je već ključ pretvoren u cijeli broj.

- $h(x) = x \bmod m$, m prost.
- $h(x) = (x \bmod p) \bmod m$ za p prost i $m < p \ll M$.
- $h(x) = ((ax + b) \bmod p) \bmod m$; a i b se slučajno biraju i $a > 0$, $b < p$.

U praksi se hash tablice često koriste za spremanje podataka čiji su ključevi stringovi. Funkcija `hashstr` pretvara string u cijeli broj. `p` bi trebao biti prost broj, uz $p \geq m$.

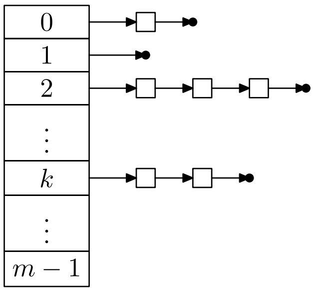
```
unsigned int hashstr(const char *s, unsigned int p)
{
  unsigned int h;
  for(h = 0; *s; v++) {
    h = ((h << 8) + (unsigned char)*s) % p;
  }
  return h;
}
```

[14] Ova složenost je *nakon* računanja hash funkcije. Samo računanje hash funkcije može imati složenost koja ovisi o duljini ključa.

3.1.2 Kolizije

Kolizije se mogu rješavati bilo ulančavanjem bilo otvorenim adresiranjem. Otvoreno adresiranje je jednostavnije za implementaciju i ekonomičnije u potrošnji memorije (ne treba čuvati dodatne pointere), ali se bitno usporava kad je puno pretinaca popunjeno.

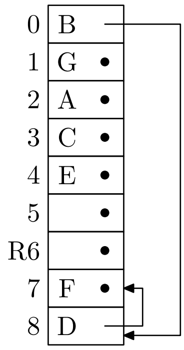
3.1.2.1 Ulančavanje



Slika 3.2 Lista za svaki ključ (separate chaining)

Jedna metoda rješavanja kolizija prikazana je na **slici 3.2**. Ovdje je svaki element hash tablice pointer na početak jednostruko vezane liste kao u **odjeljku 2.3.2**; točke su opet NULL pointeri koji označavaju kraj liste.
Kolizije se rješavaju tako da se novi element umetne u postojeću listu: tako na indeksu 0 postoji samo 1 element (i nema kolizije), na indeksu 1 još nema nijednog elementa pa pokazuje na NULL pointer, na nekom općem indeksu k su dva elementa u listi, itd...
Kada se neki ključ traži u listi (neka se hashira na poziciju k u tablici), tada je potrebno pretražiti cijelu listu da bi se ustanovilo da li postoji u tablici ili ne.

Druga metoda je ulančavanje *unutar same tablice*. U ovom algoritmu se koristi pomoćna varijabla R koja u početku pokazuje na posljednji element tablice (u ovom slučaju 8). **Slika 3.3** prikazuje stanje u tablici nakon umetanja ključeva A do G. Do prve kolizije dolazi prilikom umetanja ključa D koji je trebao biti na poziciji 0. Umjesto toga se umeće na poziciju R (u početku 8), a R se umanjuje za 1. Do sljedeće kolizije dolazi prilikom umetanja ključa F koji se trebao umetnuti na mjesto 8, međutim ta lokacija je već zauzeta pa se umeće na iduću slobodnu ($R = 7$) i R se umanjuje za 1.
Općeniti postupak kod umetanja ključa K je sljedeći: izračuna se hash ključa; neka je to h. Ako je pozicija h slobodna, umeće se na to mjesto. Ako je zauzeta, slijedi se niz linkova počevši od te lokacije. Ako se u nizu linkova nađe K, on već postoji u tablici i ne radi se ništa. Inače, jedan ili više puta se umanjuje R dok se ne nađe prazna pozicija na koju se umeće ključ i link na lokaciji h se postavlja na R. Ako R postane 0, tablica je puna i novi ključ se ne može umetnuti.



Slika 3.3 Lista unutar tablice (coalesced chaining)

3.1.2.2 Otvoreno adresiranje

Druga metoda razrješavanja kolizija je sljedeća: umjesto pointera na listu, tablica sama kao takva sadržava ključeve. Neka se novi ključ hashira na poziciju k. Ukoliko je pozicija k zauzeta, proba se na pozicije $k + 1$, $k + 2$,... Ako su zauzete sve pozicije do $N - 1$-ve (veličina tablice), probe se nastavljaju od pozicija 0, 1, 2... Ako se nakon nastavka pretraživanja od početka tablice ponovo dođe to k-te pozicije, a da se nije našlo prazno mjesto, tada je *tablica puna*. Ova metoda zove se **linearno ispitivanje)** (linear probing).

Ovakvo pretraživanje jako usporava traženje ključa pa se zato preporuča realocirati tablicu (i sve postojeće elemente rehashirati i umetnuti na nove pozicije u većoj tablici) kada postane oko 80% puna. Ovakvo ispitivanje ima lošu stranu da se stvaraju dugačke nakupine elemenata (clustering) pa se pretraga lako može pretvoriti u $O(n)$.

Zbog clusteringa postoje još dvije metode za traženje slobodne pozicije u slučaju kolizije. Jedna je **kvadratno ispitivanje**: za element x se računa $h_2(x) = h(x) + c \cdot i^2$; c je neka konstanta, a i je redni broj re-hasha.

Posljednja metoda je **dvostruko hashiranje**: računa se $h(x) + i \cdot h_2(x)$ gdje je h_2 druga hash funkcija. Rezultat te funkcije *mora biti relativno prost* sa veličinom hash tablice m. i veći od 0. To se lako postiže ako je m prost. Neke funkcije koje se koriste u praksi su $h_2(x) = m-2-k \bmod (m-2)$ i $h_2(x) = 8 - (k \bmod 8)$. Druga koristi samo zadnja tri bita od x, ali je brža.

3.1.3 Brisanje

Prilikom brisanja iz hash tablice treba paziti: ako više elemenata ima isti hash, lako se može desiti da se brisanjem jednog elementa izbrišu i *svi ostali koji imaju taj hash.* Npr. ako se u primjeru sa **slike 3.3** izbriše element B, nepovratno se gubi i element F jer F ima isti hash, a nalazi se na drugoj lokaciji. Slična situacija se dešava pri brisanju kod otvorenog adresiranja.

Jedna mogućnost je da se za svaku lokaciju čuva i *stanje* koje može biti prazna, zauzeta i izbrisana. Kada se traži neki ključ, izbrisane lokacije se preskaču kao da su zauzete. Prilikom umetanja, ključ se umeće na prvu lokaciju koja je označena kao izbrisana. Međutim, ovo rješenje je loše zato što izbrisane lokacije *nikad* više ne postanu označene kao prazne. Nakon mnogo umetanja i brisanja, *sve* lokacije će biti označene kao izbrisane (više nema praznih) i svako traženje nepostojećeg ključa će imati m koraka (tražit će se cijela tablica).

Kod otvorenog adresiranja postoji jednostavan algoritam koji "popravi" hash tablicu nakon brisanja elementa tako da nije potrebno uvoditi "izbrisano" stanje, a ne gube se ni elementi koji imaju isti hash kao i upravo izbrisani element.

3.1.4 Složenost

Statistički, složenost umetanja i traženja ključa u hash tablici je $O(1)$ Iako hash tablice imaju jako dobru *očekivanu* složenost, *najgori* slučaj je i dalje $O(n)$ koji može nastupiti ako se loše "poklope" izabrana hash funkcija i ulazni podaci.

3.2 Stabla

Matematički, stablo je poseban slučaj grafa; nejasni pojmovi iz ovog poglavlja mogu se pogledati u prva dva odjeljka **poglavlja 4**. U ovom odjeljku će se opisati strukture podataka bazirane na stablima.

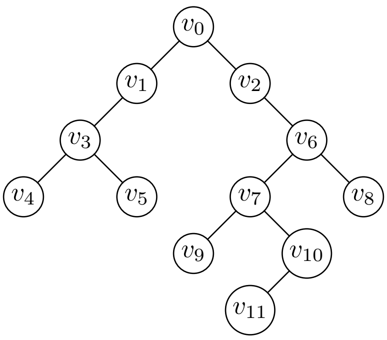
3.2.1 Definicije

Stablo Stablo je rekurzivno definirana struktura; ono je:

1. prazno, ili
2. ima korijen i 0 ili više podstabala.

Osnovni pojmovi uvedeni su primjerom i izravno se mogu proširiti i na stabla sa većim grananjem.

PRIMJER



Slika 3.4 Binarno stablo

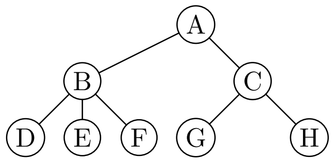
Svaki čvor ima 0 ili više **djece**. Tako čvor v_3 ima dvoje djece (v_4 i v_5), v_5 nema djece, a v_2 ima jedno dijete. Suprotno, svaki čvor (osim korijena) ima *točno jednog* **roditelja**. Tako je v_1 roditelj čvora v_3. **Korijen** (v_0) je čvor bez roditelja. Čvorovi sa istim roditeljem su **siblinzi** (v_4 i v_5). Čvorovi na "dnu" stabla (koji nemaju nijedno dijete, to su v_4, v_5, v_9, v_{11} i v_8) još se nazivaju i **listovi**.

Stupanj stabla Stupanj stabla je je najveći broj djece koje posjeduje neki čvor u stablu.

Visina stabla Duljina najduljeg puta od nekog lista prema korijenu naziva se **visina** stabla. Tako je stablo na **slici 3.4** visine 5 (to je put preko čvorova v_{10}, v_7, v_6, v_2 i v_0). Uzima se da stablo koje ima samo 1 čvor, tj. korijen ima visinu 0.

Obilazak Obilazak stabla je procedura koja svaki čvor "posjeti" točno jednom. Vrste obilaska su sljedeće (primjeri su za **sliku 3.5**):

- **postorder**: djeca pa korijen: DEFBGHCA.
- **preorder**: korijen pa djeca: ABDEFCGH.
- **inorder**: lijevo dijete, korijen, desno dijete. Ovaj obilazak ima smisla samo za binarno stablo. Kada u stablu iz primjera ne bi bilo čvora F, inorder obilazak bi bio: DBEAGCH.

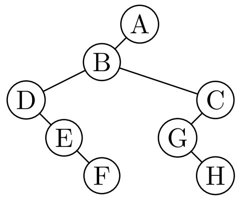


Slika 3.5 Obilasci stabla

Kako je stablo samo poseban slučaj grafa, metode za implementaciju grafova iz **poglavlja 4** su presložene za binarna stabla; stoga će se ovdje pokazati jednostavnija i efikasnija implementacija.

3.2.2 Reprezentacija stabala

U **eksplicitnoj** reprezentaciji čvor sa k djece ima polje sa k pointera. Ponekad se čuva i pointer na roditelja. Stablo *bilo kojeg stupnja* može se prikazati kao binarno stablo: lijevi pointer je na prvo dijete, desni pointer je na prvog siblinga. Siblinzi formiraju vezanu listu. **Slika 3.6** prikazuje takvu reprezentaciju stabla sa **slike 3.5**.



Slika 3.6 Stablo višeg stupnja kao binarno

U **implicitnoj** reprezentaciji se koristi polje, a veze su implicirane pozicijom u polju. Za binarna stabla se najčešće koristi sljedeći prikaz u nekom polju `A`:

1. Korijen je na `A[1]`.
2. Lijevo dijete čvora `A[i]` je na `A[2*i]`.
3. Desno dijete čvora `A[i]` je na `A[2*i+1]`.

Dimenzije je potrebno unaprijed znati. Ako je stablo dobro balansirano štedi se na pokazivačima, inače ima puno praznog prostora.

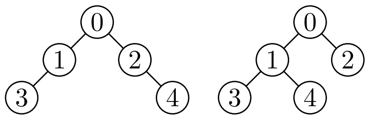
Druga varijanta implicitnog prikaza korištena je u **odjeljku 2.6** prilikom formiranja klasa ekvivalencije. Tamo je također stablo spremljeno u polje, ali lokacija `A[i]` čuva *roditelja* čvora `i`. Čvor je korijen ako je `A[i] == i`.

3.2.3 Binarna stabla

Za razliku od "običnih" stabala, kod binarnih stabala *razlikujemo* lijevo i desno dijete.

Potpuno binarno stablo Binarno stablo je **potpuno** ukoliko su svi nivoi stabla puni, osim možda zadnjeg. Na zadnjem nivou se djeca popunjavaju s lijeva na desno bez praznina.

PRIMJER



nije potpuno   potpuno

Slika 3.7 Stabla

Na slici 3.7 prikazano je potpuno i nepotpuno stablo sa 5 čvorova. Pri umetanju elemenata u stablo, visina stabla se povećava jedino ako je to nužno; ako se može umetnuti element da se ne poveća visina stabla, tada se i mora tako umetnuti. Štoviše nivoi u stablu se popunjavaju s lijeva na desno. Zbog toga lijevo stablo na slici 3.7 nije potpuno jer je element 4 umetnut kao lijevo dijete elementa 2 umjesto kao desno dijete elementa 1.

Potpuno binarno stablo od n elemenata je *uvijek* minimalne visine v i ta visina iznosi $v = \lfloor \lg n \rfloor$ (uzima se da binarno stablo od samo jednog elementa ima visinu 0). Binarno stablo visine v može sadržavati najviše $2^{v+1} - 1$ elemenata.

3.2.3.1 Umetanje i traženje

Kako binarno stablo ima najviše dvoje djece, možemo djecu staviti u strukturu čvora stabla. Djeca se tradicionalno zovu **lijevo** i **desno** dijete. Ako neko dijete ne postoji, tada će odgovarajući pointer biti `NULL`.

```
typedef struct bt_node {
  int data;               /* neki podatak */
  struct bt_node *left;   /* lijevo dijete */
  struct bt_node *right;  /* desno dijete */
} bt_node;
```

Sortirano binarno stablo (binarno stablo traženja) Kažemo da je binarno stablo **sortirano** ako je lijevo dijte *manje* od svog roditelja, a desno dijete *veće* od svog roditelja.

Postoje jednostavni (i, ako se radi sa slučajnim podacima, efikasni – reda $O(\lg n)$) algoritmi za umetanje i traženje elemenata u sortiranom binarnom stablu.

Prvo će se obraditi traženje elementa jer je to prvi korak pri umetanju. Funkcija vraća pointer na čvor koji sadrži traženi podatak (`val`) ili NULL ako podatak ne postoji. `root` je pointer na korijen stabla. `prev`, ako je različit od `NULL` pointera, će uvijek pokazivati na vrijednost `root` varijable iz prethodnog koraka petlje. To će kasnije trebati za umetanje.

U svakom koraku petlje se uspoređuje vrijednost koja se traži sa podatkom u trenutnom čvoru. Ukoliko su jednake, izlazi se iz petlje i vraća se pointer na taj čvor. Ukoliko je `val` manji od vrijednosti u čvoru, tada se mora pretražiti lijevo podstablo (jer su u desnom vrijednosti isključivo veće od `val`, zbog definicije sortiranog binarnog stabla). Analogno, ako je veći, pretražuje se desno podstablo.

```
bt_node *bt_find(const bt_node *root, int val,
  bt_node **prev)
{
  while(root) {
    if(root->data == val) break;
    if(prev) *prev = root;
    if(val < root->data) root = root->left;
    else root = root->right;
  }
  return root;
```

```
}
```

Prilikom umetanja imamo dva slučaja: ako element *postoji*, neće se umetnuti. Ako *ne postoji*, treba ga umetnuti na odgovarajuće mjesto – zadnji posjećeni čvor prije nego što funkcija `bt_find` vrati `NULL`. To je upravo vrijednost na koju se postavi `prev` u funkciji `bt_find` ako je `prev != NULL`. `bt_insert` vraća pointer na novo umetnuti čvor.

```
struct bt_node *bt_insert(bt_node *root, int val)
{
  bt_node *where, *new_n;
  if(!(new_n = bt_find(root, val, &where))) {
    new_n = malloc(sizeof(bt_node));
    new_n->left = new_n->right = NULL;
    new_n->data = val;
    if(val < where->data) where->left = new_n;
    else where->right = new_n;
  }
  return new_n;
}
```

—— Primjer ——

`bt_insert` se prvi put pozove sa `NULL` kao `root` argumentom – to označava prazno stablo te se u tom slučaju stvara korijen stabla. Svaki idući put se `bt_insert` poziva sa vrijednošću vraćenom iz prvog poziva.

```
/* primjer umetanja 1, 2 i 3 u prazno stablo */
struct bt_node *tree = bt_insert(NULL, 1);
bt_insert(tree, 2);
bt_insert(tree, 3);
```

Moguće je napraviti binarno stablo koje omogućava više pojavljivanja istog elementa, no to se neće ovdje obrađivati.

3.2.3.2 Obilasci

Sva tri načina obilaska stabala su varijante obilaska po dubini i mogu se elegantno iskodirati rekurzivno; za obilazak po širini potreban je red. Inorder obilazak binarnog stabla traženja ima lijepo svojstvo da ispisuje elemente stabla *sortiranim* redoslijedom.

```
void inorder(bt_node *root)
{
  if(root) {
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
  }
}
```

3.2.3.3 Brisanje

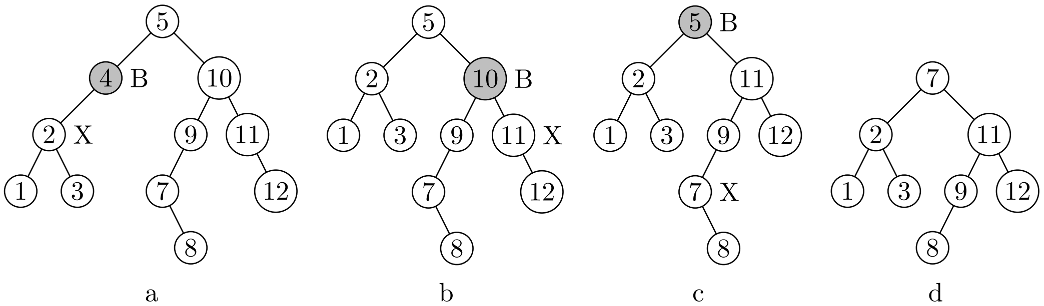
Brisanje čvora je složeno zato što treba paziti da nakon brisanja ostane zadovoljen uvjet stabla. Brisani čvor B se zamjenjuje čvorom X koji treba pronaći u stablu. Postoje tri slučaja:

1. B nema desno dijete. Traženi čvor X je lijevo dijete od B (odn. nema ga ako B nema ni lijevo dijete). U oba slučaja se B briše iz stabla i zamjenjuje sa X koji nasljeđuje njegovo lijevo i desno podstablo.
2. B ima desno dijete R, ali R nema lijevo dijete. U tom slučaju je R upravo traženi čvor X. B se briše i zamjenjuje sa X koji nasljeđuje njegova podstabla.
3. B ima desno dijete R i R ima lijevo dijete. Treba naći najmanju vrijednost X veću od B – "sljedbenika" od B u inorder obilasku. Taj čvor je "najljevije" dijete od R (tj. čvor do kojega se dolazi praćenjem isključivo lijevih linkova počevši od čvora R).

Desno podstablo od X postaje *lijevo* podstablo *roditelja* od X. X zamjenjuje B i nasljeđuje njegova podstabla.

— Primjer —

Sljedeća slika prikazuje sve slučajeve koji mogu nastupiti. Sa B je označen čvor koji se briše (također je i zasivljen) te sa X čvor koji ga zamjenjuje. Na slikama a, b i c nastupaju prvi, drugi i treći slučaj prilikom brisanja; svako iduće stablo ujedno je i rezultat brisanja čvora iz prethodnog stabla.



Slika 3.8 Brisanje čvorova iz binarnog stabla

3.2.3.4 Složenost

Binarna stabla su korak prema strukturama podataka koje daju složenost traženja ključa $O(\lg n)$. Ta složenost se postiže sortiranim binarnim stablom ako su ulazni podaci *slučajni*. Ako su ulazni podaci sortirani ili "cik-cak", stablo degenerira u listu pa je stoga najgori mogući slučaj opet $O(n)$. Slučajna ubacivanja i brisanja daju stablo visine $O(\sqrt{n})$. Balansirana binarna stabla *garantiraju* logaritamsku složenost traženja i umetanja za *sve* podatke.

3.2.4 Heap

Heapom se vrlo efikasno implementiraju operacije prioritetnog reda iz **odjeljka 1.2.9**.

Heap je potpuno binarno stablo u kojem je roditelj npr. veći od svoje djece; taj uvjet se zove **uvjet heapa**.[15]

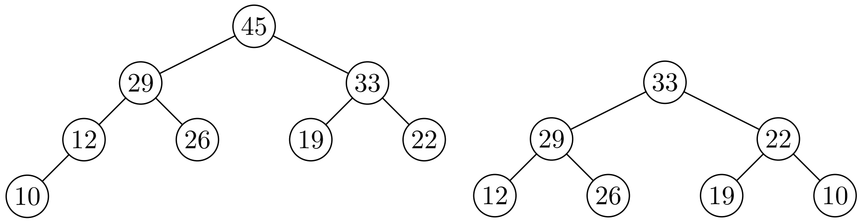
Potpuno binarno stablo se može spremiti u *jednodimenzionalan niz* bez potrebe za pamćenjem veza pomoću pointera: n elemenata stabla spremamo u niz na indekse $1 \ldots n$. Korijen stabla dolazi na indeks 1. *Djeca* čvora sa indeksom i nalaze se na indeksima $2i$ i $2i+1$, dok se njegov *roditelj* nalazi na indeksu $\lfloor i/2 \rfloor$. (Tako potpunom stablu sa **slike 3.7** odgovara niz: 0 1 2 3 4.)

— Primjer —

Pokazat će se umetanje elemenata 12, 26, 19, 29, 33, 22, 45 i 10 u heap u kojem je roditelj veći od djece (sva dijeljenja se uzimaju cjelobrojno). Nakon konstruiranja heapa, efikasna operacija je brisanje najvećeg[16] elementa.

Rezultirajuća stabla nakon konstrukcije heapa i brisanja najvećeg elementa prikazana su na **slici 3.9**.

[15] Uvjet heapa može biti bilo koja *relacija totalnog poretka*; npr. heap se može napraviti i tako da je roditelj manji od svoje djece.
[16] ili najmanjeg, ovisno o uvjetu heapa



heap

nakon izbacivanja najvećeg elementa (45)

Slika 3.9 Primjer heapa

1	2	3	4	5	6	7	8	komentar
12								Umeće se 12 na indeks 1.
12	26							Umeće se 26 na indeks 2. kako je njegov roditelj na indeksu 1 (12) manji, zamjenjuju se.
26	12	19	29					Dodaje se 19 na indeks 3. njegov roditelj je na indeksu $\lfloor 3/2 = 1 \rfloor$ (26); veći je i ne radi se ništa. Odmah potom se dodaje 29 na indeks 4. Njegov roditelja na indeksu 2 (12) je manji pa se zamjenjuju.
26	29	19	12					29 je doputovao do indeksa 2 nakon zamjene sa svojim roditeljem (12). Međutim, još uvijek je manji od svog roditelja (26 na indeksu 1) pa se zamjenjuju.
29	26	19	12	33				Dodaje se 33 na indeks 5.
33	29	26	19	12				33 je veći od svog roditelja 26 na indeksu $\lfloor 4/2 = 2 \rfloor$ pa se zamjenjuju. Nakon zamjene je opet veći od svog roditelja 29 na indeksu 1 pa dolazi opet do zamjene.
33	29	19	12	26	22			33 je doputovao do korijena kao najveći element. Dodaje se 22 na kraj (indeks 6). Veći je od svog roditelja 19 na indeksu 3 pa se zamjenjuju.
33	29	22	12	26	19	45		Nakon zamjene 19 i 22 dodaje se 45 na kraj niza (indeks 7). On putuje do korijena redom se zamjenjujući sa svojim roditeljima, to su indeksi 3 (na kojem se nalazi 22) i 1 (na kojem je 33).
45	29	33	12	26	19	22	10	Dodaje se 10 na indeks 8. Kako je manji od svog roditelja na indeksu 4 (broj 12), ne treba ništa raditi.

Brisanje najvećeg elementa ide ovako:

1	2	3	4	5	6	7	komentar
10	29	33	12	26	19	22	Najmanji element (uvijek se nalazi na kraju niza) se stavi na početak (indeks 1) i smanji se za 1 broj elemenata u nizu.
33	29	10	12	26	19	22	10 je manji od oba svoja djeteta (na indeksima $2 \cdot 1 = 2$ (29) i $2 \cdot 1 + 1 = 3$ (33)) te zamjenjuje mjesto sa *većim* djetetom (33).
33	29	22	12	26	19	10	Opet je manji od oba svoja djeteta (na indeksima $2 \cdot 3 = 6$ (19) i $2 \cdot 3 + 1 = 7$ (22)) te zamjenjuje mjesto sa većim djetetom (22). Došao je do indeksa 7. Kako je $2 \cdot 7 = 14$ izvan granica niza, element na indeksu 7 više nema djece i došao je do svoje konačne pozicije.

3.2.4.1 Umetanje i brisanje

Umetanje i brisanje će se prikazati na primjeru heapa gdje su prioriteti cijeli brojevi. Pretpostavlja se postojanje `int` niza `heap` te varijable `hsz` koja drži trenutan broj elemenata u heapu (to je istovremeno i zadnji iskorišteni indeks u nizu).

Kao što je pokazano u primjeru, element se u heap uvijek dodaje na kraj i onda se heap *popravlja* zamjenom mjesta elemenata (počevši od upravo dodanog) dok nije zadovoljen uvjet heapa. Tu zamjenu elemenata radi funkcija `upheap` koja kao argument uzima indeks unutar polja od kojeg treba početi zamjenjivati elemente.
Pri pisanju funkcije `upheap` iskorišten je trik: indeks 0 u polju se ne koristi pa se postavlja na maksimalnu vrijednost. Tako se izbjegava dodatan uvjet u uvjetu petlje koji bi bio `k/2 > 0`. Taj slučaj nastupa kada je novi element najveći element u heapu.

```
void upheap(int k)
{
  int v = heap[k];
  for(heap[0] = INT_MAX; heap[k/2] <= v; k /= 2) {
    heap[k] = heap[k/2];
  }
  heap[k] = v;
}
void insert(int v)
{
  heap[++heapsz] = v;
  upheap(heapsz);
}
```

Kako funkcija `upheap` popravlja heap od dna prema vrhu, tako funkcija "downheap" popravlja heap od vrha prema dnu počevši od nekog zadanog elementa. Funkcija `remove` element s kraja niza prebacuje na početak i popravlja heap. Vraća upravo obrisani element.

```
void downheap(int k)
{
  int j, v = heap[k];
  while(k <= heapsz) {
    j = k+k;
    if((j < heapsz) && (heap[j] < heap[j+1])) j++;
    if(v >= heap[j]) break;
    heap[k] = heap[j]; k = j;
  }
  heap[k] = v;
}
int remove(void)
{
  int v = heap[1];
  heap[1] = heap[heapsz--];
  downheap(1);
  return v;
}
```

3.2.4.2 Složenost

Sve operacije na heapu (umetanje i brisanje najvećeg elementa) su garantirano složenosti $O(\lg n)$ gdje je n broj elemenata trenutno u heapu.

3.2.5 Balansirana stabla

Ukoliko prilikom umetanja u binarno stablo elementi ne dolaze slučajnim redoslijedom, moguće je da stablo *degenerira u listu* – najgori mogući slučajevi su:

- Umetanje elemenata sortiranim redoslijedom (npr. 1,2,3,4,5,6).
- Umetanje obrnutim sortiranim redoslijedom (npr. 6,5,4,3,2,1).
- Naizmjenično umetanje velikih i malih elemenata (npr. 1,6,2,5,3,4); stablo će imati "cik-cak" oblik.

U svim tim slučajevima (za n elemenata) stablo ima visinu n umjesto optimalne visine $\lfloor \lg n \rfloor$. To bitno usporava traženje elemenata u stablu jer se stablo sada ponaša kao linearna lista – traženje traje $O(n)$ umjesto $O(\lg n)$ (a jedna od najčešćih primjena stabala je upravo brzo traženje).

U slučajevima kada nije moguće utjecati na redoslijed kojim se elementi umeću u stablo[17] ništa nam ne može garantirati da se neće desiti najgori slučaj. Tada se upotrebljavaju **balansirana stabla** koja *garantiraju* da će vrijeme umetanja i traženja biti $O(\lg n)$, neovisno o redoslijedu kojim se elementi umeću u stablo. Najpoznatije vrste balansiranih stabala su AVL stabla i crveno-crna[18] stabla.

Implementacija balansiranih stabala je znatno složenija od implementacije "običnih" binarnih stabala i neće se ovdje obrađivati. Zbog složenije implementacije, balansirana stabla čak mogu biti *sporija* od običnih binarnih stabala (ukoliko se u binarno stablo elementi umeću "ispravno" ili nema puno elemenata). Međutim, kako balansirana stabla imaju garantirani najgori slučaj $O(\lg n)$, često se koriste za implementaciju ADT mape i skupa.

3.2.5.1 AVL stabla

AVL stabla su osmislili ruski matematičari Adel'son-Vel'skii i Landis 1962. godine. To je prva struktura podataka koja garantira $O(\lg n)$ u najgorem slučaju za *sve* operacije. To se postiže trošenjem dodatnog vremena pri umetanju i brisanju kako bi se stablo izbalansiralo. Vrijeme za balansiranje također ne smije prijeći $O(\lg n)$. Čvor stabla dodatno čuva razliku dubina (**faktor balansa**) sa oznakama $-$, $\bullet$ i $+$.

Nedostaci AVL stabala su dodatna (minimalno) 2 bita u čvoru za čuvanje faktora balansa i relativno komplicirana implementacija.

AVL stablo AVL stablo je binarno stablo pretraživanja takvo da je za svaki čvor razlika u visini lijevog i desnog podstabla najviše 1. Visina h AVL stabla sa n unutarnjih čvorova je u sljedećim granicama: $\lg(n+1) < h < 1.4404 \lg(n+2) - 0.3277$.

Kod **optimalnog** stabla svi listovi su na istoj dubini.

Ubacivanje:

1. Element se ubaci na dno kao u obično binarno stablo.
2. Poprave se promijenjeni faktori balansa.
3. Ako je ravnoteža pokvarena, stablo se rebalansira. Rebalansiranje je potrebno ako faktor balansa nije $\bullet$ i ubacivanje povećava visinu na "krivu stranu". Balans se popravlja tzv. **rotacijama**.

Brisanje je složenije jer se ne zna koji se faktori balansa korigiraju. Kreće se od roditelja obrisanog čvora prema vrhu. Možda je potrebno balansirati svaki čvor i poslije brisanja može trebati $O(\lg n)$ rotacija.

[17] Ne zna se unaprijed koji su i koliko ih ima – npr. upisuje ih korisnik umjesto da se čitaju iz unaprijed pripremljene datoteke.
[18] engl. red-black trees

3.2.5.2 Red-Black stabla

Crveno-crno stablo je binarno stablo traženja ali tako da svaki čvor ima "boju" – crvenu ili crnu. Crveno-crno stablo mora poštivati sljedeće uvjete:

1. Crveni čvor ne može imati crveno dijete.
2. Svaki jednostavni put od zadanog čvora do potomaka bez djeteta ili sa jednim djetetom ima isti broj crnih čvorova. Pri tome se NULL čvorovi promatraju kao crni.

Ubacivanje:

1. Korijen stabla je crn. Ostali čvorovi koji se ubacuju su crveni. Pravilo 2 je sigurno zadovoljeno, a pravilo 1 možda nije ako je roditelj ubačenog čvora crven (kod implementacije se može birati koje pravilo će se prekršiti).
2. Ako je roditelj novog čvora crven, stablo se mora popravljati rotacijama (praroditelj je tada sigurno crn).

Brisanje:

- Ako se briše crveni čvor sve je u redu.
- Brisanjem crnog čvora prekrši se barem drugo pravilo, a ako su crveni postali susjedni prekršeno je i drugo pravilo.

3.2.5.3 Složenost

Sljedeća tablica daje usporedbu AVL i crveno-crnih stabala. n je broj elemenata u stablu u trenutku izvođenja operacije.

	AVL	red-black
pretraživanje	$O(\lg n)$	$O(\lg n)$
ubacivanje	$O(\lg n)$ s najviše jednom rotacijom.	$O(\lg n)$
brisanje	$O(\lg n)$ uz najviše $\lg n$ rotacija.	$O(\lg n)$ uz najviše tri rotacije.
visina stabla	Najveća visina AVL stabla s n čvorova manja je od najveće visine crveno-crnog stabla za isti broj čvorova.	

4 Grafovi

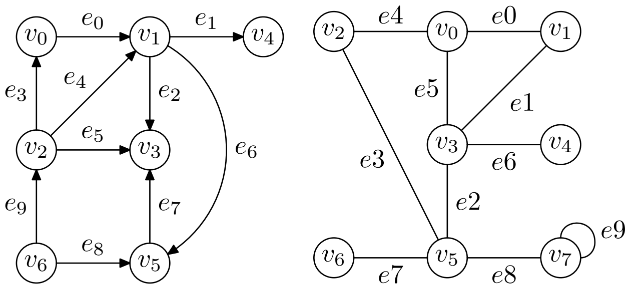
4.1 Definicije

Graf Neka je V ($|V| = \nu$) konačan skup i $E \subseteq V \times V$ ($|E| = \epsilon$). Tada se par (V, E) naziva **usmjereni graf** ili **digraf**[19] (na V) gdje je V skup *vrhova*[20], a E skup *usmjerenih bridova.*[21] Za takav graf pišemo $G = (V, E)$.

Ukoliko smjer brida nije bitan, i dalje pišemo $G = (V, E)$, ali je sada E skup *neuređenih* parova iz V.

Svakom bridu e može biti pridružena brojčana **težina**, oznakom $w(e)$.

U nastavku će se za neusmjereni graf pisati samo "graf", a za usmjereni graf će se pisati "digraf".



Slika 4.1 Primjeri grafova

Planarnost Oba grafa na slici 4.1 su nacrtana tako da se nikoja dva brida međusobno ne sijeku. Takav graf se zove **planaran**. Postoje i grafovi koji nisu planarni (npr. peterokut sa svim dijagonalama).[22]

Incidencija Kažemo da su vrhovi u i v grafa G **incidentni** s bridom koji ih spaja i obratno. Dva vrha incidentna s nekim bridom zovu se **susjedni**.

Stupanj vrha Stupanj vrha $d(v)$ za neki vrh $v \in V$ je broj bridova incidentnih sa v. Ukoliko postoji brid koji spaja v sa samim sobom (**petlja**), tada se taj brid broji dva puta (npr. vrh $v7$ u grafu na slici 4.1).

Šetnja Dodijelimo imena bridovima: neka svaki brid ima oznaku e_i. Tada je **šetnja** u grafu G netrivijalan konačni niz $W = v_0 e_1 v_1 e_2 v_2 \dots e_k v_k$ čiji su članovi naizmjence vrhovi v_i i bridovi e_i tako da su krajevi od e_i vrhovi v_{i-1} i v_i za svako i, $1 \leq i \leq k$.

Kaže se da je W šetnja od v_0 do v_k. Vrhovi v_0 i v_k zovu se redom **početak i kraj** šetnje, a broj k zove se **duljina** šetnje.

Ciklus Šetnja je **zatvorena** ako ima pozitivnu duljinu (> 0), a početak i kraj se podudaraju. Zatvorena šetnja kod koje su početak i unutrašnji vrhovi različiti zove se **ciklus**.

[19] od engl. *d*irected graph
[20] od engl. vertex, vrh
[21] od engl. edge, rub ili brid
[22] Postoje algoritmi kojima se ispituje planarnost grafa, no oni su izvan opsega ove skripte.

— Primjer —

Za digraf na slici 4.1 imamo $V = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6\}$ te $E = \{e_0 = (v_0, v_1),\ e_1 = (v_1, v_4),\ e_2 = (v_1, v_3),\ e_3 = (v_2, v_0),\ e_4 = (v_2, v_1),\ e_5 = (v_2, v_3),\ e_6 = (v_1, v_5),\ e_7 = (v_5, v_3),\ e_8 = (v_6, v_5)\}$.

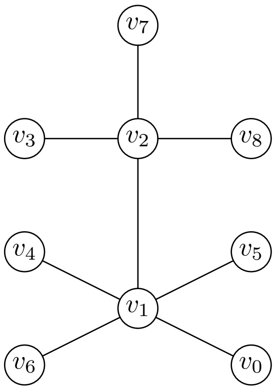
Za graf sa slike 4.1 imamo $V = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$ te $E = \{e_0 = \{v_0, v_1\},\ e_1 = \{v_1, v_3\},\ e_2 = \{v_3, v_5\},\ e_3 = \{v_2, v_5\},\ e_4 = \{v_0, v_2\},\ e_5 = \{v_0, v_3\},\ e_6 = \{v_3, v_4\},\ e_7 = \{v_5, v_6\},\ e_8 = \{v_5, v_7\},\ e_9 = \{v_7, v_7\}\}$.

Primijetite da kod digrafa imamo skup *uređenih parova* (poredak elemenata je bitan i brid "izvire" iz prvog elementa te "ponire" u drugi element), dok kod grafa imamo skup *dvočlanih skupova* (poredak elemenata u skupu nije bitan).

Za graf sa slike 4.1 vrhovi $v6$ i $v5$ su incidentni s bridom $e7$ (kao i npr. $v5$ i $v3$ sa bridom $e2$, itd...).

Jedan od ciklusa u grafu na slici 4.1 je $v_0v_2v_5v_3$.

Poseban slučaj grafa je *stablo*, neke vrste stabala imaju vrlo veliku primjenu u algoritmima.



Slika 4.2 Primjer stabla

Stablo **Aciklički graf** je onaj koji ne sadrži cikluse. **Stablo** je povezani aciklički graf. Primjer stabla prikazan je na slici 4.2.

Teorem Svaka dva vrha u stablu povezana su jednim jedinim putem.

Teorem Ako je G stablo koje ima ϵ bridova i ν vrhova, onda je $\epsilon = \nu - 1$. štoviše, vrijedi i jača tvrdnja: za *svako* stablo je $\chi(G) = 2$ gdje je $\chi(G)$ **Eulerova karakteristika** definirana za bilo kakav graf formulom $\chi(G) = \nu - \epsilon + 1$.

Ovakva stabla su puno općenitija od binarnih stabala opisanih iz poglavlja 3.2.

4.2 Reprezentacija grafa

4.2.1 Matrični prikaz

4.2.1.1 Matrica incidencije

Matrica incidencije Neka graf G ima ν vrhova $(v_0, \ldots, v_{\nu-1})$ i ϵ bridova $(e_0, \ldots, e_{\epsilon-1})$. Matrica incidencije je $\nu \cdot \epsilon$ matrica čiji je element m_{ij} broj koliko puta su vrh v_i i brid e_j incidentni. Taj broj je 0 (nisu susjedni), 1 ili 2 (vrh v_i spaja sam sebe (petlja) bridom e_j). Ako je graf usmjeren, broj može biti +1 (ulazak u čvor) ili -1 (izlazak iz čvora).

— Primjer —

Ovo su matrice incidencije za grafove sa slike 4.1. Gotovo sve operacije su jednostavnije sa matricom susjedstva te se detalji implementacije matricom incidencije neće razmatrati.

$$
\begin{array}{c}
\begin{array}{c} \\ 0\\1\\2\\3\\4\\5\\6 \end{array}
\begin{array}{c}
\begin{array}{cccccccccc} 0&1&2&3&4&5&6&7&8&9 \end{array}\\
\begin{pmatrix}
-1 & 0 & 0 & +1 & 0 & 0 & 0 & 0 & 0 & 0\\
+1 & -1 & -1 & 0 & +1 & 0 & -1 & 0 & 0 & 0\\
0 & 0 & 0 & -1 & -1 & -1 & 0 & 0 & 0 & +1\\
0 & 0 & +1 & 0 & 0 & +1 & 0 & +1 & 0 & 0\\
0 & +1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & +1 & -1 & +1 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1
\end{pmatrix}
\end{array}
\end{array}
\quad
\begin{array}{c}
\begin{array}{c} \\ 0\\1\\2\\3\\4\\5\\6\\7 \end{array}
\begin{array}{c}
\begin{array}{cccccccccc} 0&1&2&3&4&5&6&7&8&9 \end{array}\\
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0\\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0\\
0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2
\end{pmatrix}
\end{array}
\end{array}
\tag{4.1}
$$

4.2.1.2 Matrica susjedstva

Matrica susjedstva Neka graf G ima ν vrhova. Matrica susjedstva je $\nu \cdot \nu$ matrica gdje je element m_{ij} matrice jednak broju bridova koji spajaju vrhove i i j.

Za grafove je matrica susjedstva simetriča s obzirom na glavnu dijagonalu; za digrafove to ne mora biti slučaj.

Prema gornjoj definiciji, ako u vrhu i ne postoji petlja, tada je element m_{ii} u matrici susjedstva 0.[23]

—— Primjer ——

Ovo su matrice susjedstva za grafove sa **slike 4.1**.

$$
\begin{array}{c}
\begin{array}{c} \\ 0\\1\\2\\3\\4\\5\\6 \end{array}
\begin{array}{c}
\begin{array}{ccccccc} 0&1&2&3&4&5&6 \end{array}\\
\begin{pmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 1 & 1 & 1 & 0\\
1 & 1 & 0 & 1 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 1 & 0 & 0 & 0\\
0 & 0 & 1 & 0 & 0 & 1 & 0
\end{pmatrix}
\end{array}
\end{array}
\quad
\begin{array}{c}
\begin{array}{c} \\ 0\\1\\2\\3\\4\\5\\6\\7 \end{array}
\begin{array}{c}
\begin{array}{cccccccc} 0&1&2&3&4&5&6&7 \end{array}\\
\begin{pmatrix}
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0\\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0\\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0\\
1 & 1 & 0 & 0 & 1 & 0 & 0 & 0\\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0\\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 1\\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1
\end{pmatrix}
\end{array}
\end{array}
\tag{4.2}
$$

Matrica susjedstva ima sljedeće zgodno svojstvo:

Broj šetnji Neka je A matrica susjedstva grafa G. Tada je (i, j)-ti član l-te potencije A^l (matrično množenje) jednak broju (v_i, v_j) šetnji u G duljine l. Stoga je broj *svih* šetnji na G duljine l jednak sumi svih članova od A^l.

Matrica susjedstva za graf sa `n` vrhova može se alocirati i inicijalizirati na prazan graf (bez bridova) sljedećim C kodom (radi kratkoće je izostavljena provjera grešaka).

```
int **graph_create(int n)
{
  int **ret = malloc(n*sizeof(int*));
  int i, j;
```

[23] Neki autori (npr. Sedgewick) uzimaju da je $m_{ii} = 1$ čak i kada *nema* petlje u vrhu i zbog jednostavnije implementacije nekih algoritama. Ovdje će se uvijek poštovati dana definicija te će element na glavnoj dijagonali biti veći od 0 samo ako u odgovarajućem vrhu postoji petlja.

Ovaj kod postavlja sve elemente matrice (pa tako i dijagonalne elemente) na 0.

```
  for(i = 0; i < n; i++) {
    ret[i] = malloc(n*sizeof(int));
    for(j = 0; j < n; j++) ret[i][j] = 0;
  }
  return ret;
}
```

Dodavanje brida u *digraf* između vrhova v_i i v_j je jednostavno (pretpostavlja se da brid izvire u vrhu v_i a ponire u vrh v_j): za 1 se *poveća* element matrice na mjestu (i, j). Element se povećava za 1 umjesto da se postavlja na 1 zato što već može postojati brid između ta dva vrha.
Dodavanje brida u *graf* je isto što i dodavanje *dva* brida između tih vrhova u digraf, ali tako da su bridovi suprotne orijentacije.

```
void graph_d_add_edge(
  int **digraph, int i, int j)
{
  ++digraph[i][j];
}
void graph_add_edge(
  int **graph, int i, int j)
{
  graph_d_add_edge(graph, i, j);
  if(i != j) graph_d_add_edge(graph, j, i);
}
```

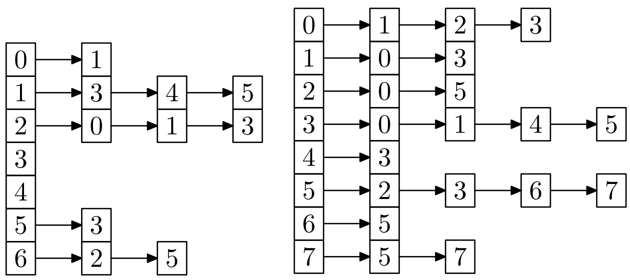
Provjera `i != j` je potrebna za slučaj petlje; kada je ne bi bilo u matrici susjedstva bi na dijagonali za vrh i imali 2 umjesto 1. Također bi i bio dvaput naveden u svojoj listi susjedstva (opisana u idućem odjeljku), što bi moglo dovesti do anomalija kod obilaska grafa.

4.2.2 Lista susjedstva

U ovoj implementaciji (di)grafa od ν vrhova imamo niz od ν pointera; pointer na i-tom indeksu je pointer na početak liste vrhova koji su povezani s vrhom i.

—— PRIMJER

Na **slici 4.3** prikazane su liste susjedstva grafova sa **slike 4.1**.



Slika 4.3 Liste susjedstva

U sljedećem kodu, koji alocira prazan graf od maksimalno `n` vrhova, tip `vertex` je tip na kojem gradimo povezanu listu susjednih vrhova. Graf `g` je niz `vertex` struktura: `g[0]` je pointer na prvi element liste susjednih vrhova vrhu 0.
Pointeri se inicijaliziraju na NULL, što označava praznu listu susjedstva za svaki vrh.

```
struct vertex {
  int id;                /* indeks vrha */
  struct vertex *next;   /* sljedeci u listi */
};
struct vertex **graph_create(int n)
{
  int i;
  struct vertex **g = malloc(n*sizeof(vertex*));
```

```
  for(i = 0; i < n; i++) g[i] = NULL;
  return g;
}
void graph_d_add_edge(
  struct vertex **digraph, int i, int j)
{
  struct *vertex v =
    malloc(sizeof(struct vertex));
  /* oznaci novi vrh */
  v->id = j;
  /* ubaci na pocetak liste susjedstva vrha i */
  v->next = digraph[i]; digraph[i] = v;
}
```

Za dodavanje novog brida između vrhova i i j u *digraf* treba samo dodati j u listu susjeda za vrh i.
Kao i prije, dodavanje brida u *graf* je isto što i dodavanje *dva* brida između tih vrhova u digraf, ali tako da su bridovi suprotne orijentacije. Međutim, sada se vidi prednost apstrakcije i “razbijanja” dodavanja brida u graf na dva dodavanja brida u digraf: iako smo promijenili strukturu podataka, i funkciju za dodavanje brida u digraf, funkcija za dodavanje brida u graf (`graph_add_edge`) ima *identičnu* implementaciju kao i u **odjeljku 4.2.1**.

U dosadašnjoj implementaciji se troši po jedan dodatan pointer za svaki čvor liste susjedstva. Ako ima puno bridova, sami pointeri mogu zauzimati puno mjesta. Stoga se može napraviti sljedeća optimizacija prostora: alocira se niz (nazovimo ga a) od $n+m$ brojeva (gdje je n broj vrhova, a m broj čvorova). Broj a_i na indeksu i za $0 \leq i < n$ sadrži *indeks* gdje u nizu počinju podaci o susjedstvu za vrh i. Podaci o susjedstvu se stavljaju istim redoslijedom kao i vrhovi (dakle prvo lista za vrh 0, pa za 1 itd...). Ako neki vrh nema susjeda, na mjesto a_i se stavlja -1. Broj susjeda od vrha i može se izračunati: $a_k - a_i$ gdje je k najmanji indeks strogo veći od i takav da je $a_k > 0$.

—— PRIMJER

Na **slici 4.4** prikazana je takva reprezentacija usmjerenog grafa sa **slike 4.1**.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a_i	7	8	11	-1	-1	14	15	1	3	4	5	0	1	3	3	2	5

Slika 4.4 Varijanta liste susjedstva

Ovakva reprezentacija ima dva nedostatka:

- Relativno spora i komplicirana izmjena grafa (dodavanje ili brisanje vrhova i bridova).
- Potrebno je izračunavati broj susjeda za svaki vrh, što može biti $O(n)$ operacija.[24]

Posljednji nedostatak se može izbjeći spremanjem podatka o broju susjeda prilikom izgradnje grafa.

4.3 Algoritmi

4.3.1 Obilasci grafa

Kao i kod stabala, **obilazak** grafa je procedura koja posjeti sve vrhove točno jednom. Razlikujemo obilazak po dubini (DFS - depth first search) i obilazak po širini (BFS - breadth first search).

[24] Primjer takvog patološkog slučaja: graf sa n vrhova i *jednim jedinim* bridom koji povezuje vrh 0 sa bilo kojim drugim. Dakle, svi ostali vrhovi imaju zapisano -1 za podatke o susjedstvu. Kada se treba izračunati broj susjeda od vrha 0, traži se indeks $k > 0$ za koji je $a_k > 0$. No takav k ne postoji i potrebno je ispitati indekse početka susjedstva svih $n-1$ vrhova da bi se to ustanovilo.

1. Izbriše se oznaka sa svih čvorova grafa.
2. Izabire se i označi početni vrh v_1 i stavlja se u *popis* P.
3. Izvadi čvor x iz popisa i označi ga.
4. Označi i stavi u popis svakog neoznačenog susjeda y od x.
5. Ponavlja se korak 3 dok popis nije prazan.

Ako se za strukturu podataka *popisa* iz algoritma uzme *stack* tada će obilazak biti *po dubini*, a ako se uzme *red* obilazak će biti *po širini*. Obilazak grafa nije jedinstven i ovisi o izboru početnog vrha i redoslijedu stavljanja susjednih vrhova.

Algoritam svaki posjećeni vrh mora označiti kako se ne bi desila beskonačna petlja. Sa stablima to nije bilo potrebno zato što niti jedna veza iz nekog čvora ne vodi prema roditelju – tako nema načina da se algoritam vrati na već posjećene čvorove.

Složenost obilaska grafa je $O(\nu)$. Ne može manje jer treba posjetiti svaki vrh.

4.3.2 Topološko sortiranje

Topološko sortiranje se u praksi pojavljuje npr. kod određivanja redoslijeda izvođenja međusobno ovisnih zadataka. Zadaci čine čvorove *usmjerenog* grafa. Ako je zadatak i preduvjet za izvršenje zadatka j tada se povlači *usmjereni* brid *iz* vrha i *u* vrh j. Topološko sortiranje nad takvim grafom daje redoslijed[25] izvršenja zadataka. Iz očitog razloga topološko sortiranje nije moguće napraviti ako graf sadrži ciklus.

U općem slučaju imamo skup S nad kojim je definirana relacija parcijalnog poretka. Elementi skupa S su čvorovi usmjerenog grafa, te ako za neke $x, y \in S$ vrijedi $x < y$ onda graf sadrži usmjereni brid od x prema y. Kažemo da je x prethodnik od y.

1. Stavi se $k = 1$.
2. Bira se vrh v_k u grafu takav da nijedan brid ne završava u v_k (tj. vrh nema prethodnika). Iz grafa se briše v_k kao i svi bridovi koji počinju u v_k.
3. Uveća se k za 1. Ako je $k = n$, kraj i imamo poredak $v_1 < v_2 < \ldots < v_n$. Inače se vraća na korak 2.

Praktično se algoritam implementira tako da se za svaki čvor čuva vezana lista njegovih sljedbenika, kao i ukupan broj *prethodnika*. Tada se čvor bez prethodnika lako prepozna (ima 0 prethodnika). Svaki put kad se izbriše neki čvor x, za 1 se umanji broj prethodnika svih sljedbenika od x. Kako bi se izbjeglo traženje čvorova bez prethodnika, oni se čuvaju u redu. Novi čvorovi se dodaju u red pri brisanju postojećih čvorova (kad broj prethodnika nekog čvora postane 0).

4.3.3 Minimalno razapinjuće stablo

Razapinjuće stablo Razapinjuće stablo je podgraf koji je stablo i ostavlja povezanim sve vrhove koji su povezani u početnom grafu.

Minimalno razapinjuće stablo (Minimum Spanning Tree) je razapinjuće stablo najmanje ukupne težine.

Problem minimalnog razapinjućeg stabla pojavljuje se npr. u povezivanju čvorova telefonske mreže sa što manje žice. U ovom slučaju težini brida odgovara fizička udaljenost čvorova.

[25] Jedan od: često ima više rezultata topološkog sortiranja istog grafa.

Kruskalov i Primov algoritam za pronalaženje minimalnog razapinjućeg stabla su **pohlepni** (greedy). Pohlepan algoritam uvijek bira trenutno najbolji korak ne gledajući unaprijed i kad dođe do kraja pretpostavlja da je to rješenje.

Pohlepni algoritmi ne daju uvijek najbolje rješenje – npr. u problemu trgovačkog putnika.

4.3.3.1 Kruskalov algoritam

1. Postavi se brojač $i = 1$ i bira se brid e_1 minimalne težine.
2. Ako su za $1 \leq i \leq n-2$ izabrani bridovi $e_1, e_2, \ldots, e_i$, bira se brid e_{i+1} tako da:
   a. e_{i+1} ima najmanju težinu,
   b. dodavanje tog brida ne čini ciklus uz već odabrane bridove.
3. Poveća se i za 1. Ako je $i = n-1$, kraj. Inače se vraća na korak 2.

Ako je graf bio povezan, označeni bridovi čine minimalno razapinjuće stablo. Ako nije bio povezan, bridovi čine šumu; svako stablo u šumi je minimalno razapinjuće stablo za tu komponentu povezanosti grafa.

Za provjeru ciklusa može se koristiti struktura podataka iz **odjeljka 2.6**. Inicijalno je svaki vrh u svojoj vlastitoj klasi ekvivalencije. Ako se bridom povežu dva stabla, svi vrhovi se prebacuju u istu klasu ekvivalencije i dobivaju istu oznaku (predstavnika). Ako najkraći brid ima iste oznake u oba vrha tada oni pripadaju istom stablu.

4.3.3.2 Primov algoritam

Primov algoritam dijeli vrhove u dva skupa: P - skup obrađenih vrhova (i dodanih u stablo) te N - skup neobrađenih vrhova. Uvijek vrijedi da je $P \cup N = V$. U svakom koraku algoritma se jedan vrh prebacuje iz N u P.

1. Postavi se brojač $i = 1$ i u P se stavlja proizvoljan vrh v_1. Stavi se $N = V - \{v_1\}$ i $T = \emptyset$.
2. Neka je za $1 \leq i \leq \nu - 1$: $P = \{v_1, v_2, \ldots, v_i\}$, $T = \{e_1, e_2, \ldots, e_{i-1}\}$, $N = V - P$. U T se dodaje brid najmanje težine koji povezuje jedan vrh x iz P i jedan vrh $y = v_{i+1}$ iz N. y se stavlja u P i briše iz N.
3. Poveća se i za 1. Ako je $i = n$, kraj. Inače se vraća na korak 2.

Ako je graf bio povezan, označeni bridovi čine minimalno razapinjuće stablo. Inače se dobije stablo komponente povezanosti od početnog vrha.

4.3.3.3 Složenost

Kod Kruskalovog algoritma treba sortirati bridove – to ima složenost $O(\epsilon \lg \epsilon)$. Nakon što su bridovi sortirani, petlja algoritma se izvršava najviše $\epsilon - 1$ puta. Provjeru zatvaranja ciklusa nije moguće napraviti u konstantnom vremenu i može se pokazati da je *ukupna* složenost algoritma (uz provjeru ciklusa) $O(\nu \lg \nu)$. Ako je graf povezan i nije stablo, vrijedi $\nu \leq \epsilon$ te se također i za *ukupnu* složenost može reći da je $O(\epsilon \lg \epsilon)$.

Tipična implementacija Primovog algoritma ima složenost $O(\nu^2)$. Složenije implementacije postižu složenost $O(m \lg \nu)$.

4.3.4 Najkraći put

Neka su težine bridova u usmjerenom grafu brojevi koji predstavljaju udaljenosti između vrhova (u ovom problemu će se zvati duljine). Traži se najkraći put između dva vrha. Duljina puta je suma duljina bridova od kojih se put sastoji.

Postoji nekoliko algoritama najkraćeg puta:

1. Dijkstrin algoritam traži najkraći put od zadanog početnog vrha do svih ostalih vrhova u grafu.
2. Bellman-Fordov algoritam ima sličnu namjenu kao i Dijkstrin algoritam. Međutm, dopušta bridove sa *negativnim* udaljenostima.
3. Floydov algoritam računa najkraći put između *svih parova* vrhova u grafu. Iako se to može postići izvršavanjem Dijkstrinog algoritma $\nu - 1$ puta (složenost $O(\nu^3)$), Floydov algoritam je mnogo brži (iako je iste složenostr $O(\nu^3)$).

4.3.4.1 Dijkstrin algoritam

Osnovna ideja Dijkstrinog algoritma je sljedeća: neka je $S \subseteq V$ tako da je početni vrh $u_0 \in S$ te neka je $\bar{S} = V - S$. Neka je $P = u_0 \dots \bar{u}\bar{v}$ najkraći put od u_0 do $\bar{S}$. Tada je $\bar{u} \in S$ i $(u_0, \bar{u})$-dio od P mora biti najkraći $(u_0, \bar{u})$ put. Stoga je

$$d(u_0, \bar{v}) = d(u_0, \bar{u}) + w(\bar{u}, \bar{v})$$

a udaljenost u_0 od $\bar{S}$ dana je s

$$d(u_0, \bar{S}) = \min_{u \in S, v \in \bar{S}} \{d(u_0, u) + w(u, v)\} \tag{4.3}$$

Formula 4.3 je osnovna formula Dijkstrinog algoritma.

Kako bi se izbjegle mnoge usporedbe i ubrzao algoritam, uvodi se proces *označavanja* grafa. Kroz cijeli algoritam svaki vrh v nosi oznaku $l(v)$ koja je gornja granica od $d(u_0, v)$. Tijekom algoritma se označavanje mijenja.

1. Odabire se početni vrh v_0 i stavlja se $l(v_0) = 0$ i $l(v) = \infty$ za $v \neq v_0$. $S_0 = \{v_0\}$, $i = 0$.
2. U i-tom koraku, za svako $v \in \bar{S}_i$, zamijenimo $l(v)$ sa $\min\{l(v), l(u_i) + w(u_i, v)\}$. Izračuna se $\min_{v \in \bar{S}_i} l(v)$ i neka je u_{i+1} vrh u kojem se postiže taj minimum. Stavlja se $S_{i+1} = S_i \cup \{u_{i+1}\}$.
3. Ako je $i = \nu - 1$, kraj. Inače se uveća i za 1 i vraća se na korak 2.

Kad algoritam završi, udaljenost od v_0 do v dana je konačnom vrijednošću $l(v)$. Algoritam se može i ranije prekinuti ako nas zanima udaljenost do nekog određenog (a ne svih) vrha.

Algoritam kako je napisan samo računa udaljenosti. Ako nas zanima i konkretan put najkraće duljine, algoritam se proširuje u drugom koraku: za svaki novo dodani vrh u skup S_i pamti se njegov prethodnik.

Složenost Dijkstrinog algoritma je $O(\nu^2)$.

5 Sortiranje

Problem sortiranja Zadani su zapisi $R_1, R_2, \ldots, R_n$ s ključevima $K_1, K_2, \ldots, K_n$. Treba naći permutaciju p tako da je $K_{p(1)} <= K_{p(2)} <= \ldots <= K_{p(n)}$. Tada kažemo da su zapisi R_i **uzlazno sortirani**.

Ima mnogo različitih metoda, a koja će se koristiti ovisi o mnogo faktora, poput:

- broj elemenata koji treba sortirati,
- veličina zapisa,
- arhitektura kompjutera.

Ovdje će se razmatrati isključivo **interno sortiranje**: pretpostavlja se da svi zapisi (i eventualno dodatna potrebna memorija) odjednom stanu u radnu memoriju kompjutera.

Prilikom sortiranja zapisi se mogu:

- Fizički micati po memoriji – to je sporo ako su zapisi veliki.
- Napraviti polje pointera na zapise pa se sortiraju pointeri (a uspoređuju se ključevi na koje pokazuju).
- Puniti dodatno polje sa redoslijedom (addres table sorting).
- Zapis se proširi poljem veze. U ovom slučaju se može:
  - Svi zapisi se povežu u (nesortiranu) listu pa se sortira lista preslagivanjem veza.
  - Proces sortiranja postavlja veze tako da rezultat bude sortirana lista (a elementi u nizu ostaju u originalnom redoslijedu).

Stabilno sortiranje Algoritam sortiranja je **stabilan** ako zapisi s jednakim ključevima ostaju u originalnom poretku.

—— PRIMJER ——

Zadan je sljedeći skup zapisa (uređenih parova) gdje je ključ prva komponenta: $(3, a)$, $(1, b)$, $(2, c)$, $(1, c)$, $(4, d)$. On se može sortirati na dva načina:

1. Stabilno: $(1, b)$, $(1, c)$, $(2, c)$, $(3, a)$, $(4, d)$. U originalnom nizu je zapis $(1, b)$ bio prije zapisa $(1, c)$ i taj poredak je očuvan.
2. "Ne-stabilno": $(1, c)$, $(1, b)$, $(2, c)$, $(3, a)$, $(4, d)$. Ako ima više zapisa s istim ključevima, tada raste i broj načina na koji se oni mogu (ne-stabilno) sortirati.

Teorem Može se dokazati da je sortiranje n zapisa usporedbom ključeva[26] $\Omega(n \lg n)$ problem.

5.1 Sortiranje brojanjem

Ako je zapis u sortiranom nizu na poziciji j, znači da je manji od $j-1$ zapisa. Algoritam za svaki ključ prebrojava od koliko preostalih ključeva je manji i sprema taj broj u CNT pol-

```
for(i = N-1; i > 0; i--) {
  for(j = i-1; j >= 0; j--) {
    if(K[i] < K[j]) ++CNT[j];
    else ++CNT[i];
```

[26] Postoje druge metode koje se ne baziraju na usporedbi ključeva, međutim one nisu široko primjenjive u praksi i neće se ovdje razmatrati.

je (koje se na početku mora inicijalizirati na 0). Na kraju algoritma je vrijednost `CNT[j]` jednaka mjestu na koje u sortiranom nizu ide zapis `R[j]`.

```
  }
}
```

5.2 Parcijalno sortiranje brojanjem

Ako su ključevi (ili dio ključa) u malom rasponu $u \leq K_i \leq v$, tada se može prebrojati koliko puta se koji ključ pojavljuje. Polje `CNT` broji koliko puta se pojavljuje koji ključ (i na početku mora biti inicijalizirano na 0).

```
for(i = 0; i < N; i++) ++CNT[K[i]-u];
```

5.3 Insertion sort

Insertion sort simulira način kako se slažu karte po veličini u npr. beli. Za zapis R_j se traži odgovarajuće mjesto među već sortiranim zapisima $R_1, \ldots, R_{j-1}$. Pri umetanju se svi zapisi pomiču jedno mjesto udesno.
Algoritam kako je napisan, sortira elemente na indeksima od 0 do $n-1$. Međutim, petlja se može bitno ubrzati eliminacijom `j>0` uvjeta iz petlje tako da se na `a[0]` stavi najmanji mogući element.

```
void insertion_sort(int *a, int n)
{
int i, j, v;
  for(i = 1; i < n; i++) {
    for(v = a[i], j = i;
        (j > 0) && (a[j-1] > v); j--)
      a[j] = a[j-1];
    a[j] = v;
  }
}
```

Prosječan broj operacija je nešto manji od $2.25N^2 + 7.75N = O(N^2)$. Modifikacija algoritma (**binary insertion**) prilikom traženja mjesta za novi element koristi binarno traženje, ali zbog velikog pomicanja zapisa složenost ostaje $O(N^2)$.

5.4 Shell sort

Pomicanje elemenata za jedno mjesto ne može se postići složenost bolja od $O(n^2)$. Donald L. Shell uvodi metodu kod koje se vanjskim nizom kontrolira duljina "skoka" pri sortiranju elemenata.

— PRIMJER —

Za 16 zapisa i niz brojeva $h_1 = 8$, $h_2 = 4$, $h_3 = 2$, $h_4 = 1$ sortiraju se sljedeće grupe elemenata u 4 prolaza:

1. 8 grupa: (R_1, R_9), (R_2, R_{10}),...,(R_8, R_{16}). Kaže se da je nakon ovog prolaza niz **8-sortiran**.
2. 4 grupe: (R_1, R_5, R_9, R_{13}),...,$(R_4, R_8, R_{12}, R_{16})$.
3. 2 grupe: $(R_1, R_3, \ldots, R_{15})$,...,$(R_2, R_4, \ldots, R_{16})$.
4. 1 grupa – svi elementi: $(R_1, R_2, \ldots, R_{16})$.

Bitno je da m-sortiranje čuva n-sortiranje kad je $m < n$.

Sa h_i je označena duljina skoka. Mora biti $h_0 = 1$ i $h_i < h_{i+1}$ za $i \in \{1, \ldots, s-1\}$. Sortiranje počinje od h_s prema manjim koracima. Postavlja se pitanje kako birati inkremente h_i. Matematička anailza Shell sorta je vrlo složena i optimalna sekvenca za Shell sort do sada *nije poznata.*

```
void shell_sort(int *a, int n)
{
  int i, j, h, v;
  for(h = 1; h <= n/9; h = 3*h + 1) ;
  for(; h > 0; h /= 3) {
    for(i = h; i < n; i++) {
      for(v = a[i], j = i;
          (j >= h) && (a[j-h] > v); j-= h)
        a[j] = a[j-h];
      a[j] = v;
    }
  }
}
```

Međutim, vrijede sljedeće ograde:

- Za $h_s = 2^{s+1} - 1, 0 \leq s < t = \lfloor \lg N \rfloor$ složenost je $O(N^{3/2})$.
- Ako se h_s bira iz skupa svih brojeva oblika $2^p 3^q < N$, tada je složenost $O(N \lg^2 N)$. Međutim, to ne ubrzava bitno sortiranje jer zahtijeva velik broj prolaza.
- Sljedeći niz (R. Sedgewick) ima složenost $O(N^{4/3})$:

$$h_s = \begin{cases} 9 \cdot 2^s - 9 \cdot 2^{s/2} + 1, & \text{ako je } s \text{ paran;} \\ 8 \cdot 2^s - 6 \cdot 2^{(s+1)/2} + 1, & \text{ako je } s \text{ neparan.} \end{cases}$$

- Za mali broj zapisa (npr. manji od 1000), najbolje je koristiti jednostavan niz poput $h_0 = 1$, $h_{s+1} = 3h_s + 1$ i stati na h_{t-1} kada je $h_{t+1} > N$. Ovaj niz je napravljen u danom kodu.

5.5 Selection i heap sort

Selection sort traži najmanji element u nizu i zamijeni ga sa prvim elementom niza. Zatim traži drugi najmanji element i zamijeni ga sa drugim elementom niza itd.

```
void selection_sort(int *a, int n)
{
  int i, j, t, min;
  for(i = 0; i < n-1; i++) {
    for(min = i, j = i+1; j < n; j++)
      if(a[j] < a[min]) min = j;
    t = a[min]; a[min] = a[i]; a[i] = t;
  }
}
```

Prednost običnog (kvadratne složenosti) selection sorta je što svaki element niza pomiče *točno jedanput* što ima primjenu pri sortiranju nizova u kojima su zapisi veliki, a ključevi mali. Ali zbog kvadratne složenosti ipak nije primjenjiv na velike nizove.

Prebacivanjem elemenata niza u heap (za što nije potreban pomoćni niz), može se složenosti svesti na $O(n \log n)$. Ta varijanta se zove heap sort.

5.6 Quick sort

5.6.1 Sortiranje

Odabere se zapis (tzv. pivot) i elementi unutar niza se prebacuju tako da se svi koji su manji od pivota nalaze s lijeve strane, a svi koji su veći od pivota se nalaze s desne strane pivota (funkcija `partition`). Nakon toga se postupak rekurzivno ponovi na dva novodobivena skupa (elementi manji od pivota i veći od pivota).
Česta modifikacija je sljedeća: kad je veličina nekog segmenta manja od M (npr. 9), sortira se insertion sortom.
Prosječna složenost algoritma za $M = 9$ iznosi $11.667(N + 1)\ln N - 1.74N - 18.74$. Najgori slučaj je *već sortirani niz* i tada algoritam degenerira u složenost $O(N^2)$.
Modifikacije u kojima najgori slučaj ostaje $O(N^2)$, ali puno rjeđe:

- Odabir slučajnog pivota u intervalu $[lo_0, hi_0]$.
- Uzme se srednji po veličini (**median**) od 3 elementa na indeksima lo_0, $\lfloor(lo_0 + hi_0)/2\rfloor$ i hi_0.

Postoji metoda koja za quicksort garantira $O(N \ln N)$ uz bitno kvarenje konstantnog faktora.
Quick sort *nije stabilan.*
Prikazana funkcija uzima niz te donju i gornju granicu niza (uključivo) unutar kojih će se sortirati elementi.

```
void partition(int *a, int *lo, int *hi, int pivot)
{
  int tmp;
  while(*lo < *hi) {
    while((a[*lo] <= pivot) && (*lo < *hi)) ++*lo;
    while((a[*hi] >= pivot) && (*lo < *hi)) --*hi;
    tmp = a[*lo]; a[*lo] = a[*hi]; a[*hi] = tmp;
  }
}
void quicksort(int *a, int lo0, int hi0)
{
  int lo = lo0, hi = hi0, pivot, tmp;
  if(lo >= hi) return;
  if(lo == hi-1) {
    tmp = a[lo]; a[lo] = a[lo+1]; a[lo+1] = tmp;
    return;
  }
  pivot = a[(lo+hi)/2];
  a[(lo+hi)/2] = a[hi];
  a[hi] = pivot;
  partition(a, &lo, &hi, pivot);
  a[hi0] = a[hi];
  a[hi] = pivot;
  quicksort(a, lo0, lo-1);
  quicksort(a, hi+1, hi0);
}
```

5.6.2 k-ti po veličini

Nekad je potrebno pronaći k-ti najmanji element. Ako je k blizu 1 ili N, traži se najmanji element $k(N - k)$ puta. Takav algoritam zahtijeva oko kN usporedbi.
Drugi način je sortirati pa uzeti k-ti element; to je složenosti $O(N \log N)$.
Također, može se postupiti kao u quicksortu: niz se podijeli pivotom i odredi se u kojem je podnizu k-ti element. Međutim, za razliku od quicksorta, sada se sortira samo taj podniz. U prosjeku je složenost ovakve procedure $O(n)$, međutim loš odabit pivota, kao i kod quicksorta, daje složenost $O(n^2)$.

```
int select_k(int *R, int lo0, int hi0, int k)
{
  int lo = lo0, hi = hi0, pivot;
  if(lo >= hi) return lo;
  if(lo == hi - 1) {
    mswap(R+lo, R+lo+1, sizeof(*R));
    return;
  }
  pivot = R[(lo+hi)/2];
  R[(lo+hi)/2] = R[hi];
  R[hi] = pivot;
  partition(R, &lo, &hi, pivot);
  R[hi0] = R[hi];
  R[hi] = pivot;
```

```
  if(lo - lo0 + 1 >= k)
    return select_k(R, lo0, lo-1, k);
  return select_k(R, hi+1, hi0, k - (lo - lo0 + 1));
}
```

5.7 Merge sort

Merge sort je “divide and conquer” (podijeli pa vladaj) algoritam: niz koji treba sortirati podijeli na dva dijela, svaki dio se posebno sortira te se dva sortirana niza spajaju u jedan u $O(n)$ koraka. Spajanje sortiranih nizova (merge) je slično operaciji unije iz **odjeljka 2.7.1**; razlika je što se moraju sačuvati ponovljeni elementi.
Ova funkcija uzima lijevu i desnu granicu (uključivo) niza `a` koji treba sortirati. Parametar `b` je unaprijed alocirani pomoćni prostor koji mora imati mjesta za barem `r-l+1` elemenata.

```
void mergesort(int *a, int *b, int l, int r)
{
  int i, j, k, m;
  if(l < r) {
    m = (r+l) / 2;
    mergesort(a, b, l, m);
    mergesort(a, b, m+1, r);
    i = l; j = m+1; k = l;
    while((i <= m) && (j <= r))
      b[k++] = a[i] < a[j] ? a[i++] : a[j++];
    while(i <= m) b[k++] = a[i++];
    while(j <= r) b[k++] = a[j++];
    for(k = l; k <= r; k++) a[k] = b[k];
  }
}
```

Osim što je u prosjkeu sporiji od quicksorta, merge sort koristi i pomoćni niz (`b`) iste veličine kao ulazni niz (`R`). Međutim, mergesort ima garantiran najgori slučaj $O(n \lg n)$ i *stabilan* je. Još jedna prednost merge sorta je to što ga je moguće napraviti nad vezanom listom tako da ne treba zamjenjivati elemente nego se samo manipulira linkovima čvorova. U tom slučaju ne treba ni dodatan prostor.

6 Ostali algoritmi

6.1 Podijeli pa vladaj

"Podijeli pa vladaj" algoritmi podijele problem na manje instance istog problema i zatim rješenja manjih potproblema "spoje" u rješenje početnog problema. Neke osnovne primjene tehnike "podijeli pa vladaj" već su prikazane na sortiranju nizova u **poglavlju 5** (quicksort, k-ti element i mergesort). Ovdje će se prikazati još neke primjene.

6.1.1 Minimum i maksimum skupa

"Klasično" rješenje traži maksimum ($n-1$ usporedbi) te iz ostatka traži minimum ($n-2$) usporedbi što daje ukupno $2n-3$ usporedbi.
Neka je skup S veličine $|S| = n = 2^m$. S se može podijeliti u dva skupa S_1 i S_2 jednake veličine. Minimum i maksimum od S se dobije iz minimuma i maksimuma S_1 i S_2 sa dvije usporedbe. To daje sljedeću rekurziju:

$$T_n = \begin{cases} 1, & n = 2 \\ 2 - T_{n/2} + 2, & n > 2 \end{cases}$$

Rješenje ove rekurzije je $T_n = \frac{3}{2}n - 2$, dakle ušteda od $n/2$ usporedbi.
Isti rezultat se može dobiti i bez rekurzije: pretpostavimo da je problem riješen za $n-2$ elemenata i promatraju se sljedeća dva elementa. Oni se usporede međusobno, veći se usporedi sa dosadašnjim maksimumom, manji se usporedi sa dosadašnjim minimumom.

```
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#define MAX(a, b) ((a) > (b) ? (a) : (b))
void minmax(int *K, unsigned int n, int *min, int *max)
{
  register int mi, ma;
  for(mi = ma = *K; n >= 2; n -= 2, K += 2) {
    if(K[0] <= K[1]) {
      mi = MIN(mi, K[0]);
      ma = MAX(ma, K[1]);
    } else {
      mi = MIN(mi, K[1]);
      ma = MAX(ma, K[0]);
    }
  }
  /* neparan broj elemenata; n je tada tocno 1 */
  if(n) {
    mi = MIN(mi, *K);
    ma = MAX(ma, *K);
  }
  *min = mi; *max = ma;
}
```

6.1.2 Brzo potenciranje

Zadatak je izračunati x^α. Postoje dva slučaja: potencija α je cjelobrojna i pozitivna, te α je proizvoljan realan (ili kompleksan) broj.

Ako je potencija pozitivan cijeli broj, tada je postupak brzog potenciranja primjenjiv u svakom skupu nad kojim je definirana *asocijativna* binarna operacija među elementima i koji je zatvoren s obzirom na tu operaciju; x je tada element tog skupa.

6.1.2.1 Cjelobrojne pozitivne potencije

Neka je zadan $x \in R$; treba izračunati x^n.[27] "Očit" algoritam treba $n-1$ množenja: redom se računa $x^2 = x \cdot x$, $x^3 = x \cdot x^2$, … Međutim, moguće je izračunati n-tu potenciju sa *logaritamskim*

[27] U ovom odjeljku će se potencija označavati sa n kako bi se naglasilo da je to pozitivan cijeli broj.

brojem množenja. To je pogotovo važno kad se radi sa velikim potencijama i/ili skupovima u kojima je množenje elemenata komplicirano (npr. matrice ili polinomi).

n se zapiše u binarnom brojnom sustavu i tada se svaka jedinica zamijeni nizom slova "SX", a a svaka 0 sa "S"; početni "SX" se izbriše. Rezultat je string kako izračunati x^n gdje "S" označava kvadriranje postojećeg rezultata, a "X" množenje sa x (računanje uvijek počinje od x). Na primjer, za $n = 23 = 10111_2$ se dobije "SSXSXSX" što vodi na uzastopno računanje sljedećih potencija x^2, x^4, x^5, x^{10}, x^{11}, x^{22}, x^{23}. Ovo pravilo je ispravno zbog svojstava binarnog brojnog sustava.

Ovaj "binarni" algoritam zahtijeva da se bitovi od n pregledavaju s lijeva na desno, a u praksi je lakše obrnuto[28] jer operacije cjelobrojnog dijeljenja i ostatka sa 2 daju bitove s desna na lijevo. Zbog toga je često zgodniji prikazani program.

```
unsigned int fastpow(unsigned int x, unsigned int n)
{
  register unsigned int y = 1;
  while(n) {
    if(n & 1) y *= x;
    x *= x;
    n >>= 1;
  }
  return y;
}
```

Iako bitno (za veliki n) smanjuje broj množenja sa $n-1$ na $\lfloor \lg n \rfloor + \nu(n)$ ($\nu(n)$ je broj jedinica u binarnom zapisu od n), ova metoda ne daje uvijek i *najmanji mogući* broj množenja. Najmanji n za koji se to dešava je $n = 15$ gdje binarni algoritam treba 6 množenja, a moguće je sa 5 sljedećim slijedom računanja: $y = x^3$ je moguće izračunati sa dva množenja ($x^2 = x \cdot x$, $x^3 = x \cdot x^2$), a y^5 sa još tri dodatna ($y^2 = y \cdot y$, $y^4 = (y^2) \cdot (y^2)$, $y^5 = y \cdot y^4$).

Generalni problem *minimalnog* broja množenja za računanje potencija je još uvijek neriješen; rješenje je poznato samo za malu klasu potencija sa do 4 jedinice u binarnom zapisu.

6.1.2.2 Realne i kompleksne potencije

Značenje realnih potencija se definira na sljedeći način:

1. Negativne potencije: $x^{-\alpha} = 1/x^{\alpha}$.
2. Racionalne potencije: $x^{m/n} = \sqrt[n]{x^m}$, ako su m i n cijeli brojevi.
3. U općem slučaju je $x^{\alpha} = \exp(\alpha \ln x)$.

Pravilo 1 se može kombinirati sa algoritmom brzog potenciranja jedino ako je α negativan *cijeli* broj. U slučaju 2 najjednostavnije je za praktično računanje iskoristiti definiciju 3 koja vrijedi i u slučaju *kompleksnog* x, α (ili oboje); međutim tada je potenciranje *višeznačna funkcija*.

6.1.3 Nultočke realne funkcije (metoda bisekcije)

Zadatak je naći rješenje jednadžbe $f(x) = 0$ za neku zadanu funkciju f. Metoda bisekcije se temelji na sljedećem teoremu iz matematičke analize: ako je funkcija na intervalu $[x_1, x_2]$

```
float bisect(
  float x1, float x2, float (*f)(float),
  float eps)
{
```

[28] Za varijabilni n. Ukoliko je n fiksan i treba izračunati velik broj istih potencija, lako se unaprijed pripremi potprogram; bilo ručno ili specijaliziranom kompajler-rutinom u programu.

neprekinuta i ako je $f(x_1) * f(x_2) < 0$ (tj. mijenja predznak), tada funkcija u intervalu $[x_1, x_2]$ ima bar jednu nultočku x_0.
Sam algoritam je vrlo sličan binarnom traženju. Funkcija `bisect` uzima kao argumente interval $[x_1, x_2]$ unutar kojeg funkcija f mijenja predznak te samu funkciju f preko pointera na funkciju `f`. Kriterij pronalaska nultočke je da je apsolutna vrijednost funkcije dovoljno mala (parametar `eps`). Zbog inherentne nepreciznosti floating-point aritmetike ne može se ispitivati da `f(x0)` bude jednako točno 0.

```
  float x0 = x1, f0 = f(x0);
  /*
    zamjena x1 i x2 tako da je uvijek
    f(x1) <= 0, a f(x2) > 0
  */
  if(f0 >= 0) {
    float tmp = x1; x1 = x2; x2 = tmp;
  }
  while(fabsf(f0) > eps) {
    x0 = (x1 + x2) / 2;
    f0 = f(x0);
    if(f0 < 0) x1 = x0;
    else x2 = x0;
  }
  return x0;
}
```

6.2 Slučajni brojevi

Kompjuterski program *ne može* generirati prave slučajne brojeve. Moguće je dobiti niz brojeva koji *izgleda* slučajno, a zapravo je dobiven determinističkim postupkom. Zbog toga se govori o **pseudo-slučajnim brojevima**. Pravi slučajni brojevi mogu se dobiti tek raznim fizikalnim procesima te naknadnom A/D konverzijom izmjerenih rezultata.

C jezik ima osnovnu podršku za generiranje slučajnih brojeva u obliku sljedećih funkcija deklariranih u `stdlib.h` headeru:

```
int rand(void);
void srand(unsigned int seed);
```

Funkcija `rand` vraća slučajni cijeli broj u intervalu od 0 do konstante `RAND_MAX`. Funkcija `srand` postavlja početnu vrijednost generatora – ista vrijednost daje uvijek isti niz.

Za ozbiljne primjene (npr. Monte Carlo simulacije) ne preporuča se korištenje (bar ne bez detaljnog testiranja) ovog ugrađenog generatora.

6.2.1 Generiranje

Najčešća (i najbolje proučena) metoda generiranja slučajnih je linearni kongruentni generator.[29] Izabere se početna vrijednost X_0 te se idući slučajni broj[30] generira prema sljedećoj formuli:

$$X_{n+1} = (aX_n + c) \bmod m$$

gdje su a, c i m unaprijed izabrane konstante. Ovako definiran niz brojeva je *periodičan* s periodom najviše m. Ako su konstante loše izabrane, period može biti i manji od m.

Pri izboru konstanti i korištenju ovakvog generatora bi se trebalo držati sljedećih principa:

[29] engl. LCM; Linear Congruential Method
[30] U nastavku će se pisati "slučajni broj", ali će se zapravo podrazumijevati "pseudo-slučajni uniformno distribuirani broj".

1. Početna vrijednost X_0 se može izabrati proizvoljno. Ista početna vrijednost daje uvijek isti niz slučajnih brojeva.
2. m treba biti velik. Pogodno je uzet da m bude jednak 2^{32} na 32-bitnim binarnim procesorima jer je tada `unsigned int` aritmetika automatski modulo 2^{32}.
3. Ako je m potencija broja 2, a treba izabrati tako da vrijedi $a \bmod 8 = 5$. Također, a bi trebao biti unutar intervala $0.01m$ i $0.99m$ i ne bi trebao imati neki jednostavan uzorak znamenaka bilo u decimalnom bilo u binarnom sustavu.
4. Konstanta c *ne smije* imati zajedničkih faktora sa m.
5. Najdesnije znamenke generiranih brojeva nisu "jako slučajne" i stoga je najbolje generirani broj X promatrati kao slučajni realni broj X/m u intervalu $[0, 1)$.
   Za dobivanje slučajnih cijelih brojeva u intervalu $[0, k-1]$, X/m treba pomnožiti sa k i odbaciti decimale. Računanje po formuli $X \bmod k$ *nije dobro* jer ne daje "dovoljno slučajne" brojeve.
6. Sa istim konstantama (pogotovo a) ne bi trebalo generirati više od oko $m/1000$ slučajnih brojeva inače će se budući brojevi ponašati sve sličnije prošlima.

Ova pravila proizlaze iz duboke teorije. Pri korištenju u ozbiljnim primjenama, generator slučajnih brojeva se treba i *testirati* određenim procedurama.

6.2.2 Slučajne permutacije

Svih permutacija od n elemenata ima $n!$. Jedna metoda je generirati slučajni broj $k \in [0, n! - 1]$ te algoritmom iz odjeljka ?? generirati k-tu po redu permutaciju. Međutim, za velike n-ove to zahtijeva aritmetiku s velikim brojevima (npr. običan špil karata ima 52 karte i $52! \sim 8.06 \cdot 10^{67}$).

Dani kod "miješa" ulazni niz. `a` je ulazni niz, `n` je broj elemenata u nizu. Međutim, ovaj algoritam ne može dati više od m različitih permutacija. Taj problem se može riješiti povećanjem perioda generatora slučajnih brojeva.

```
void rand_shuffle(int *a, int n)
{
  int k, tmp;
  while(n) {
    /* slucajni broj izmedju 0 i n-1 ukljucivo */
    k = n-- * ((float)rand() / RAND_MAX);
    tmp = a[k]; a[k] = a[n]; a[n] = tmp;
  }
}
```

6.2.3 Slučajni uzorak

Potrebno je izabrati m slučajnih zapisa iz skupa veličine n. Ako je n dovoljno mali tako da svi zapisi stanu u memoriju, problem se jednostavno riješi: generira se m slučajnih brojeva i ti zapisi se izaberu.

Međutim, postoji elegantan "on-line" algoritam koji za svaki zapis koji dođe odlučuje da li će se prihvatiti ili neće u uzorak. Takav algoritam je npr. primjenjiv na izbor slučajnih zapisa iz datoteke: nije potrebno učitati cijelu datoteku nego se sekvencijalno čita zapis po zapis.
Ako je t broj svih pročitanih zapisa i k broj izabranih zapisa, tada se $t+1$-vi zapis bira s vjerojatnošću $\frac{m-k}{n-t}$. Naime, od svih načina za biranje

```
void rand_sample(unsigned int n, unsigned int m)
{
  unsigned int t = 0, k = 0, elt = 0;
  while(k < m) {
    if((n-t) * ((float)rand() / RAND_MAX) >= m - k)
++t;
    else {
      printf("%d\n", elt);
      ++k; ++t;
```

m od n zapisa tako da se k zapisa pojavljuje u prvih t, točno $\binom{n-t-1}{m-k-1}/\binom{n-t}{n-k} = (m-k)/(n-t)$ njih bira $t+1$-vi element.
Radi jednostavnosti, funkcija `rand_sample` je napisana da bira slučajan uzorak od brojeva do n; izabrani brojevi se ispisuju.

```
      }
      ++elt;
    }
  }
```

7 Reference

Knjige:

1. Steven S. Skiena: The Algorithm Design Manual
2. Robert Sedgewick: Algorithms in C
3. David R. Hanson: C Interfaces and Implementations
4. Richard Heathfield, Lawrence Kirby et al.: C Unleashed
5. Steven S. Skiena, Miguel A. Revilla: Programming Challenges
6. D.E.Knuth: The Art of Computer Programming, vol. 1-3
7. D. Veljan: Kombinatorika s teorijom grafova

Resursi dostupni na internetu:

1. AT&T alati. Između ostaloga ima biblioteka ATP-ova (Cdt), custom memory allocator (vmalloc) the napredni I/O library (Sfio). http://www.research.att.com/sw/tools/
2. glib: ATP-ovi te razne rutine portabilne na različite OS-ove (threadovi, socketi i sl.). Nalazi se u sklopu GTK projekta. http://www.gtk.org

1 Pripreme za vježbe

1.1 Vrijeme izvršavanja

Kako je teoretsko predviđanje vremena izvršavanja algoritma često matematički vrlo složeno, u ovoj vježbi će se pokazati kako se empirijski može *izmjeriti* koliko traje izvršavanje nekog dijela programa.

Za mjerenje *intervala* vremena će nam poslužiti funkcija `clock_t clock(void)` koja se nalazi u standardnom headeru `<time.h>`. Da bismo izmjerili trajanje nekog segmenta programa u sekundama, napravimo kod kao desno. `CLOCKS_PER_SEC` je konstanta također definirana u `<time.h>` headeru i odgovara broju za koji se "interni sat" programa poveća u jednoj sekundi.

```
clock_t c1, c2;
float duration;
c1 = clock(); /* pocetak mjerenja */
/* kod koji se mjeri */
c2 = clock(); /* kraj mjerenja */
duration = (float)(c2 - c1) / CLOCKS_PER_SEC;
```

Budući da se kod koji se mjeri može jako brzo izvršiti (brže nego što je potrebno da se interni sat uveća za barem 1, pa ispada da traje 0 sekundi), uputno je mjeriti koliko traje *petlja izvršavanja tog segmenta više puta zaredom* te ukupno vrijeme podijeliti brojem izvršavanja.

```
clock_t c1, c2;
float duration;
int count;
c1 = clock(); /* pocetak mjerenja */
/* izvrsimo kod koji mjerimo N puta */
for(count = 0; count < N; count++) {
  /* kod koji se mjeri */
}
c2 = clock(); /* kraj mjerenja */
duration = (float)(c2-c1) / CLOCKS_PER_SEC / N;
```

U ovoj vježbi će se uspoređivati vrijeme izvršavanja bubble sorta i quick sorta (o sortiranju će kasnije biti još govora). U fajlu `vj1.c` nalazi se kostur programa u koji radi sljedeće: Sa komandne linije uzima koliko elemenata tipa `struct entry` treba učitati iz fajla `vj1.dat` u polje `original`, te broj ponavljanja mjerenja. Fajl `vj1.dat` treba se nalaziti u istom direktoriju kao i exe.

Za rješenje zadatka bitne su sljedeće funkcije i tipovi iz programa:

```
typedef int (*cmp_fn)(const void *a, const void *b);
void bubble(void *arr, size_t n, size_t sz, cmp_fn cmp);
void qsort(void *arr, size_t n, size_t sz, cmp_fn cmp);
struct entry {
  char name1[25], name2[25];
  int key;
};
```

- `cmp_fn`: Ovo je `typedef` na tip pointera na *funkciju usporedbe*. Funkcija tog tipa vraća < 0 ukoliko je element `a` manji od elementa `b`, 0 ako su jednaki te > 0 ako je `a` veći od `b` (vraćena vrijednost nije bitna; bitan je samo predznak). (Također pogledati dokumentaciju za `qsort`.)
- `bubble, qsort`: Ovo su funkcije koje sortiraju niz elemenata `arr`. `n` je broj elemenata u nizu, `sz` je veličina pojedinog elementa, a `cmp` je pointer na funkciju usporedbe.

Napomena: `qsort` je standardna C funkcija definirana u headeru `<stdlib.h>`. Zbog toga njena implementacija "fali" u `vj1.c`.

Uputa:

1. Napisati dvije funkcije usporedbe i to tako da u jednom slučaju elementi budu sortirani po *rastućim* vrijednostima polja `key`, a u drugom po *padajućim* vrijednostima. Svaka funkcija posebno mora moći brojati koliko puta je pozvana, a također se mora moći brojač postaviti na 0.

2. Glavni program u varijablu `num` učita broj elemenata niza koji treba sortirati, a u varijablu `repeat` koliko puta treba ponavljati sortiranje. Napisati kod koji će `repeat` puta ponavljati poziv funkcije sortiranja nad nizom od `num` elemenata. Prije svakog poziva funkcije sortiranja treba niz `original` kopirati u niz `sorted` te sortirati niz `sorted`.[1]

3. Kod iz prethodne točke treba mjeriti posebno `repeat` puta sortiranje u *uzlazni* poredak `bubble` funkcijom, a posebno `repeat` puta sortiranje `quick` funkcijom. Na kraju izvršavanja, posebno za svaku funkciju `bubble` i `quick` treba ispisati:

    1. Ukupno vrijeme izvršavanja.

    2. Vrijeme izvršavanja *jednog prolaza* kroz petlju.

    3. Broj usporedbi prilikom jednog poziva funkcije sortiranja.[2]

4. Iste podatke kao u prethodnoj točki, ali za sortiranje u *silazni* poredak.

5. Izvršiti program za num=10, 25, 50, 100, 250, 500 i 1000 elemenata. Broj ponavljanja neka bude 100. Koliki su relativni odnosi vremena izvršavanja i usporedbi?

Napomena: Iako jednostavan, ovakav način mjerenja izvršavanja programa je prilično nepraktičan i, što je važnije, neprecizan u većim programima. Zato uz većinu dobrih kompajlera dolazi tzv. **profiler**, program koji puno preciznije mjeri trajanje programa, a može čak i brojati koliko puta se koja linija koda izvršila. Detalji ovise o kompajleru i operacijskom sustav pa je potrebno pogledati dokumentaciju koja dolazi uz kompajler.

1.2 Stack

Tradicionalno zapisujemo aritmetičke izraze tako da između dva operanda dođe operator, npr. `2+3`. U RPN-u *prvo dolaze operandi, a zatim operator*. Prethodni primjer tako postaje `2 3 +`. Prednost RPN-a je da *nisu potrebne zagrade*; izraz `2*(6-4)`[3] u RPN-u postaje `2 6 4 - *` ili, ekvivalentno, `6 4 - 2 *`.

RPN izrazi se jednostavno računaju pomoću stacka: ako se sa ulaza pročita operand, tada ga stavimo na stack. Ako se pročita operator, sa stacka se skine onoliko operanada koliko treba za obavljanje operacije, izračuna se rezultat i rezultat se stavi na stack.

[1] Trajanje sortiranja ovisi i o *redoslijedu* elemenata u nizu. Zato uvijek sortiramo početni niz, a ne već sortirani niz. Naravno, i kopiranje traje neko vrijeme što unosi pogrešku u mjerenje, ali relativni odnosi vremena ostaju isti.

[2] Budući da uvijek sortiramo iste podatke, i broj usporedbi je u svakom prolazu petlje isti: treba podijeliti ukupan broj usporedbi sa brojem ponavljanja petlje.

[3] Ovdje je potrebna zagrada jer množenje ima veći prioritet od zbrajanja.

—— PRIMJER ——

Računanje izraza `6 4 - 3 *`

```
6
6 4
2
2 3
6
```

Za svaki pojedinačni ulaz ispisan je sadržaj stacka u jednom redu i to tako da je *vrh stacka najdesnije.*

Napisati program koji će simulirati jednostavan RPN kalkulator uz sljedeće zahtjeve:

- Program mora raditi sa `float` brojevima.
- Operatori koje mora podržavati: `+ - * /`, `n` (negacija: promjena predznaka elementu na vrhu stacka), `p` (ispis stacka na ekran), `q` (izlaz iz programa).
- Program mora prijaviti grešku ukoliko na stacku nema dovoljno elemenata za obavljanje operacije (npr. ne može se napraviti `+` sa samo jednim elementom na stacku). Također treba prijaviti grešku ukoliko se upiše nepostojeći operator.

—— PRIMJER ——

Ovako bi trebao raditi programa (? je prompt kalkulatora):

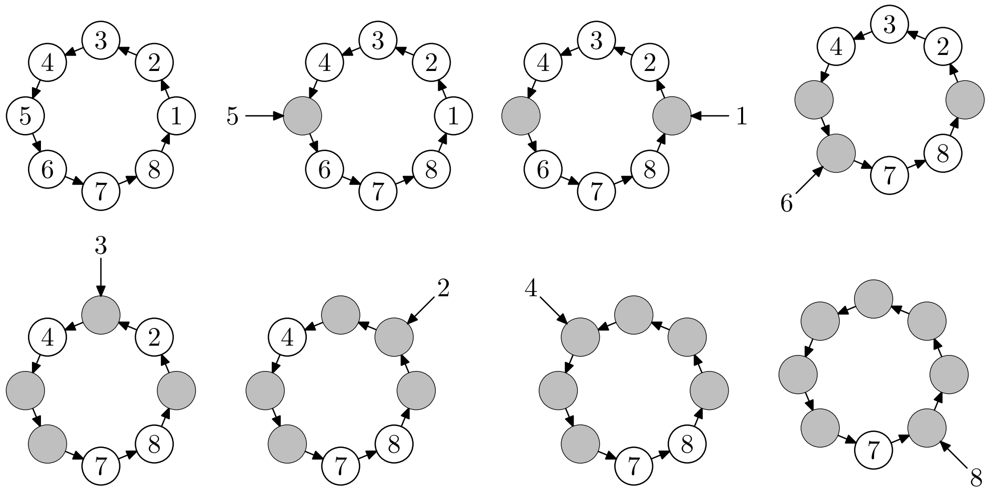
```
? 2
? 3
? p
3
2
? +
? p
5
? n
? p
-5
? +
**GRESKA: nema dovoljno argumenata
? q
```

Uputa: Ulaz sa tastature učitavati pomoću `fgets` funkcije. Tako učitani string se u float broj može pretvoriti pomoću `sscanf` funkcije i `%f` formata. Vrijednost koju vrati `sscanf` (obavezno pogledati dokumentaciju!) govori da li je pretvorba stringa uspjela (tj. da li je učitan float broj ili operator).

1.3 Liste

Prema legendi, židovski povjesničar Flavius Josephus (cca. 37.–100. n.e.) našao se u bitci između Rimljana i Židova te se sakrio u nekoj pećini sa još 40-ak ljudi. Kako su pećinu opkolili Rimljani, zarobljeništvo je izgledalo neizbježno. Većina je radije htjela umrijeti nego se predati Rimljanima pa su odlučili počiniti kolektivno samoubojstvo. Tako su svi stali u krug i svaka treća osoba u krugu je bila ubijena. Zadnji koji preostane je trebao počiniti samoubojstvo. Međutim, Josephus i njegov prijatelj su brzo izračunali kamo trebaju stati u krug da bi ostali zadnja dvojica na životu. Tako su bili zarobljeni (i ispričali cijeli događaj).

Neka je zadano n elemenata poredanih u krug. Počevši od nekog, unaprijed zadanog, elementa k odbrojimo m elemenata i izbacimo ga (elemente brojimo npr. u *pozitivnom* smjeru (suprotno od kazaljke na satu)). Nakon izbacivanja elementa, brojanje nastavljamo od 1 sa prvim sljedećim neizbačenim elementom. Brojanje započinje od elementa k, ali on *se ne izbacuje* odmah na početku; ostavlja se za poslije. Na **slici 1.1** prikazan je slučaj za $n = 8$, $m = 4$ i $k = 1$. Redoslijed izbacivanja je: 51632487.



Slika 1.1 Redoslijed izbacivanja za $n = 8$, $m = 4$, $k = 1$

Napisati program koji će uzeti n, m i k sa komandne linije te ispisivati redoslijed kojim su elementi izbacivani iz kruga. Primjer ($ je prompt komandne linije):

```
$ joseph 8 4 1
5 1 6 3 2 4 8 7
```

Uputa: Umjesto stvarnog izbacivanja elementa iz kružne liste, lakše ga je fizički ostaviti u listi, a samo ga označiti kao izbačenog. Tada se prilikom brojanja svakog m-tog elementa preskaču elementi označeni kao izbačeni.

1.4 Bit vektori

U oba zadatka obavezno koristite bit vektor za predstavljanje liste brojeva, odn. zauzetih sektora!

1.4.1 Eratostenovo sito

Eratostenovo sito je način za izračunavanje prostih brojeva $\leq n$ gdje je n unaprijed zadan. Postupak “na papiru” je sljedeći:

1. Ispišu se svi brojevi od 2 do n te se zaokruži 2 kao najmanji prosti broj. Zatim se križaju svi višekratnici od 2 (4, 6, 8, ...).
2. Pronađe se idući najmanji neprecrtani broj (3) te se on zaokruži. Zatim se križaju svi njegovi višekratnici (nužno će se sresti i već precrtani brojevi).
3. Sve dok ima neprecrtanih i nezaokruženih brojeva u listi se ponavlja postupak pronalaženja idućeg neprecrtanog broja, njegovog zaokruživanja i precrtavanja svih njegovih višekratnika.
4. Na kraju postupka su svi zaokruženi brojevi prosti.

Napišite program koji sa komandne linije učitava n te Eratostenovim sitom računa i na standardni izlaz ispisuje proste brojeve $\leq n$.

1.4.2 Alokacija diska

Neki filesystemi koriste bit vektor za označavanje koji sektori na disku su zauzeti (označeni sa 1), a koji slobodni (označeni sa 0). Ponekad je potrebno alocirati određen broj (n) *uzastopnih* sektora (tzv. blok sektora) na disku.

Napišite program koji sa komandne linije učitava ime ulazne datoteke i n te ispisuje početak i duljinu svakog bloka slobodnih sektora čija je duljina $\geq n$. Ulazna datoteka se sastoji od jednog reda 0 i 1 gdje svaka znamenka predstavlja jedan sektor, npr:

```
01000101001000011111010111101001000011
```

Bitovi se broje tako da je prvi bit 0.

Kad bi se u ovom primjeru tražili svi blokovi slobodnih sektora duljine 3 ili više, program bi ispisao sljedeće parove početka i duljine: $(2, 3)$, $(11, 4)$, $(32, 3)$.

1.5 Hash tablice

U ovoj vježbi će se simulirati jednostavna baza podataka. Svaka tablica je zadana tekstualnom datotekom u kojoj je svaki redak jedan zapis, a polja zapisa su međusobno odvojena znakom `#`. Prvo polje je *ključ* retka i smije biti više različitih redaka sa istim ključem. Retci mogu imati različit broj polja. Sadržaj svih polja je tekstualni i u sadržaju se neće pojaviti `#` (dakle, svaki `#` uvijek započinje novo polje).

Primjer izgleda datoteke s 3 retka:

```
polje1#drugo polje#trece polje
polje1#polje 2
drugi redak#polje 2#polje 3
```

Prvi i treći redak imaju 3 polja, a drugi dva. Također prva dva retka imaju isti ključ `"polje1"`.

Napišite program koji će s komandne linije uzeti ime dvije datoteke te ispisati sve retke koji imaju iste ključeve. Retci trebaju biti ispisani povezani, kao da su jedan redak (s time da se prvo ispisuju podaci iz prve tablice), a zajednički ključ se ispisuje samo jednom. Redoslijed ispisa nije bitan.

—— PRIMJER ——

Za datoteke:

```
-- prva datoteka
A#b1#c1
B#b2#c2
B#b3
C#b4#c4
D#b5#c5

-- druga datoteka
B#q1
C#q2
C#q3
E#q4
F#q5
```

izlaz (jedan od mogućih, može se razlikovati redoslijed redaka) treba biti:

```
B#b2#c2#q1
B#b3#q1
C#b4#c4#q2
C#b4#c4#q3
```

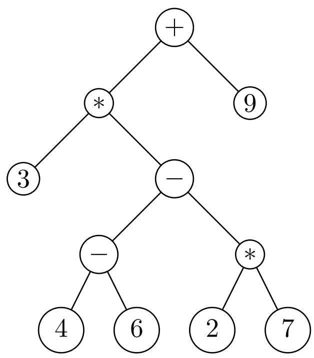
Uputa: Sadržaj pojedinih polja u retku najlakše se izdvaja funkcijom `strtok`. Odgovarajuće retke pospremite u hash tablicu (pri tome sami izaberite prikladnu hash funkciju) i to tako da je ključ hash tablice ključ retka, a vrijednost u hash tablici pointer na početak vezane liste *offseta u datoteci* gdje počinju retci s tim ključem.

1.6 Stabla

U LISP-u i njegovim derivatima (npr. Scheme), binarno stablo se može prikazati kao *lista* od 3 elementa: `(root left right)` gdje je `root` sadržaj čvora stabla, a `left` i `right` su lijevo i desno podstablo prikazani listom. Uvodimo i *prazno stablo* koje se označava sa `()`.
Binarna stabla su pogodna za izračunavanje aritmetičkih izraza.

— Primjer —

Stablo na **slici 1.2** odgovara sljedećem aritmetičkom izrazu: $3*(4-6-2*7)+9$. Ono se listom može prikazati kao `(+ (* (3 () ()) (- (- (4 () ()) (6 () ())) (* (2 () ()) (7 () ())))) (9 () ()))`. Ova forma, iako u potpunosti odgovara definiciji stabla, je vrlo nespretna kod unosa brojeva jer *svaki* broj ima *prazno* i lijevo i desno podstablo.
Definiramo *skraćeni* zapis binarnog stabla za aritmetički izraz gdje se za broj x ne piše `(x () ())` već samo `x`. Tako ovaj zapis postaje `(+ (* 3 (- (- 4 6) (* 2 7))) 9)`.



Slika 1.2 Aritmetički izraz prikazan stablom

Napišite program koji:

a. Učitava aritmetički izraz zadan skraćenom listom binarnog stabla i na temelju nje u memoriji izgrađuje binarno stablo izračunavanja.

b. Ispisuje binarno stablo u inorder obilasku; kako bi ispisani izraz bio aritmetički točan, svako podstablo (osim brojeva) treba u ispisu ograditi zagradama, npr: `((3 * ((4-6) - (2*7))) + 9)`.[4]

c. Izračunava i ispisuje vrijednost aritmetičkog izraza.

Uputa: Ovako zadano stablo najlakše se učitava rekurzivnom funkcijom.

1.7 Grafovi 1

Proširiti strukturu vrha sa oznakom (`label`) koja prilikom kreiranja vrha mora biti inicijalizirana na 0.

```
struct vertex {
  int id;               /* broj vrha */
  int label;            /* oznaka vrha */
  struct vertex *next; /* sljedeci u listi */
};
```

Napisati funkciju koja će iz datoteke učitati opis grafa i vratiti graf implementiran listom susjedstva kako je opisano **odjeljku 4.2.2**. Funkcija ima sljedeći prototip:

```
struct vertex **graph_load(const char *fname);
```

gdje je `fname` ime datoteke koja sadrži opis grafa. Funkcija neka vrati NULL u slučaju greške (npr. ne može se otvoriti datoteka).

Format ulazne datoteke je sljedeći:

```
<V>[D|U]<E>
# ovo je komentar
E parova bridova svaki u svojoj liniji
```

`<V>` je broj vrhova, `D` ili `U` označava usmjereni (D) ili neusmjereni (U) graf, a `<E>` je broj bridova. Ako je graf neusmjeren, tada u datoteci svaki brid treba pisati *jedanput*, ali se i dalje dodaje dvaput u graf. Linija koja počinje sa `#` je komentar i treba ju preskočiti.

—— Primjer ——

Za grafove sa **slike 4.1** datoteke bi izgledale ovako:

```
# usmjereni graf: prvi vrh je izvor, drugi je ponor
7D10
6 2
6 5
2 3
2 0
2 1
0 1
1 3
1 4
```

[4] Stavljanje zagrada u ispisu samo tamo gdje je to nužno radi prioriteta računskih operacije je dosta teže isprogramirati.

```
1 5
5 3

# neusmjereni graf: redoslijed vrhova u bridu nije bitan
8U10
6 5
7 5
7 7
5 2
5 3
2 0
3 4
3 0
3 1
1 0
```

1.8 Grafovi 2

Implementirati funkcije za DFS i BFS obilazak grafa. Program neka učita iz datoteke graf i neka pita za početni vrh obilaska. Ispisati DFS i BFS obilazak grafa.

Obje funkcije neka imaju sljedeći prototip:

```
typedef void (*visit_fn)(const struct vertex*);
void graph_bfs(const struct vertex **graph, const struct vertex *start, visit_fn visit);
void graph_dfs(const struct vertex **graph, const struct vertex *start, visit_fn visit);
```

Funkcije obilaska, umjesto "hard-kodirane" akcije pri obilasku vrha (npr. ispis), uzimaju pointer na funkciju (`visit_fn` typedef) koja obavlja neku radnju. `start` argument je pointer na početni vrh obilaska grafa.

1.9 Sortiranje

Napisati program koji iz datoteke koja se zadaje u komandnoj liniji učitava cijele brojeve (svi se nalaze u jednom redu) te na standardni izlaz ispisuje dvije linije:

- Brojeve sortirane *silaznim* poretkom, bez duplikata.
- Brojeve zajedno sa brojem ponavljanja. Ponovljeni brojevi se ispisuju samo jednom i to redoslijedom kako su se pojavili u ulaznoj datoteci.

—— PRIMJER ——

Za ulaz

```
3 1 2 2 -2 1 3 0 5 3 3 2
```

program ispisuje

```
5 3 2 1 0 -2
(3,3) (1,2) (2,3) (-2,1) (0,1) (5,1)
```

---

2 Domaće zadaće

Zadaci u svakoj skupini namijenjeni su za rješavanje prije svakog kolokvija i moraju se u *pismenom* obliku donijeti na kolokvij.

Svaki student rješava broj zadatka koji se računa na temelju broja indeksa prema sljedećem postupku: prva znamenka broja se pomnoži sa 1, druga sa 2, … te se svi ti umnošci pozbrajaju. Od rezultata se uzme ostatak pri dijeljenju sa n i doda 1 (gdje je n broj zadataka u skupini). Sljedeći program računa broj zadatka:

```
#include <stdio.h>

unsigned int brzad(const char *bridx, unsigned int n)
{
  unsigned int i, sum = 0;

  for(i = 1; *bridx; i++, bridx++) sum += i * (*bridx - '0');
  return sum % n + 1;
}

int main()
{
  char bridx[256];
  unsigned int n;

  printf("broj indeksa : "); scanf("%s", bridx);
  printf("broj zadataka: "); scanf("%u", &n);
  printf("**ZADATAK    : %u\n", brzad(bridx, n));
  return 0;
}
```

2.1 Prva skupina

1. Proširiti implementaciju vektora tako da za svaki element vektora prati je li inicijaliziran ili nije. Proširiti `vector_get` tako da javi grešku kod pristupa neinicijaliziranom elementu (slično kao što javlja ako se pristupa elementu izvan granica niza).

   Treba sasvim ukinuti default vrijednosti u vektoru.

2. Proširiti implementaciju nizom iz **odjeljka 2.3.1** tako da umjesto prijavljivanja "STACK OVERFLOW" greške realocira veći niz. Prijaviti grešku tek ako realokacija ne uspije.

3. Implementirati sljedeće funkcije nad jednostruko povezanom *kružnom listom* s dummy čvorom:

   - Računanje broja elemenata u jednostruko povezanoj listi.
   - Vraćanje k-tog elementa liste ili NULL ako lista ima manje od k elemenata.
   - Napišite funkciju za izbacivanje elementa iz jednostruko povezane liste. Funkcija neka ima prototip

     ```
     Element_t list_remove(struct node *n);
     ```

`n` je neposredni prethodnik čvora koji treba izbaciti. Funkcija neka dealocira izbrisani čvor te neka vrati podatak koji je bio pohranjen u čvoru.
- Napišite funkciju koja uzima pointer na čvor jednostruko povezane liste i vraća njegovog neposrednog prethodnika.
- Implementirajte funkciju koja radi operaciju "splice".
- Implementirajte funkciju `list_delete`.

Osmislite i napišite program kojim će se testirati ispravnost rada svih funkcija.

4. Koristeći funkcije i naputke iz **odjeljka 2.5** napisati potpunu implementaciju ATP-a set. Svi prototipovi (osim `set_new`) funkcija moraju odgovarati onima iz **odjeljka 1.2.3**. Proširiti implementacijske detalje iz **odjeljka 2.5** tako da podržava umetanje cijelih (pozitivnih i negativnih) brojeva u intervalu $[m, n]$ u skup. Interval neka se specificira u konstruktoru ATP-a (`set_new`).

   Osmislite i napišite program kojim će se testirati ispravnost rada ADT-a.

5. Modificirajte implementaciju iz **odjeljka 2.7.1** tako da umjesto arraya i pointer aritmetike rade sa jednostruko vezanim listama.

   Osmislite i napišite program kojim će se testirati ispravnost rada obje funkcije.

6. Kao što je rečeno u **odjeljku 2.7.2.1**, u slučaju istih elemenata nije specificirano na koji će funkcija `bsearch` vratiti pointer. Napišite vlastitu implementaciju binarnog traženja tako da u prisutnosti više istih elemenata, *uvijek* vrati pointer na *prvi* element u nizu (više njih istih). Prototip vlastite implementacije mora se slagati s prototipom C rutine `bsearch`.

   Osmislite i napišite program kojim će se testirati ispravnost rada funkcije.

7. Implementirati sljedeće funkcije nad binarnim stablom:

   - Dealociranje svih čvorova.
   - Računanje visine.
   - Računanje broja elemenata.
   - Ispis obilaska u preorder i postorder obilasku.
   - Brisanje zadanog čvora.

   Osmislite i napišite program kojim će se testirati ispravnost rada svih funkcija.

8. Na temelju koda iz **odjeljka 3.2.4** implementirajte ADT prioritetnog reda. Svi prototipovi moraju odgovarati onima iz **odjeljka 1.2.9**.

   Osmislite i napišite program kojim će se testirati ispravnost rada ADT-a.

9. Napisati program koji broji učestalost pojavljivanja riječi u tekstualnoj datoteci. Podatak u čvoru neka pointer na sljedeću strukturu (pretpostavlja se da nema riječi dulje od 64 znaka):

```
struct word {
  char w[64];  /* riječ */
  int  freq;   /* broj pojavljivanja */
};
```

   Strukture spremajte u sortirano (prema riječi) binarno stablo.

10. Kao prethodni zadatak, ali koristeći hash.

11. Modificirajte implementaciju prioritetnog reda iz **odjeljka 3.2.4** tako da *najmanji* element bude korijen stabla (također se mijenja uvjet heapa: korijen je manji od oba djeteta).

    Modifikaciju isprobajte na sljedeći način: učitava se n brojeva i svi se stavljaju u heap. Skidaju se s heapa jedan po jedan i ispisuju. Ako je implementacija ispravna, brojevi će biti ispisani sortirani *uzlaznim* poretkom. Takav sort se zove **heap sort**.

12. Proširite algoritme iz **odjeljka 2.7.1** tako da rade sa više od dva sortirana niza. Algoritmi također moraju kroz svaki niz proći najviše jednom.

2.2 Druga skupina

1. Napisati program koji izračunava broj bridova u grafu.

2. Ispitati da li je graf povezan. *Uputa:* Napraviti DFS po grafu počevši od proizvoljnog vrha. Ako ostane neki neoznačen vrh, graf *nije* povezan.

3. Implementirati Dijkstrin algoritam.

4. Implementirati Kruskalov algoritam. Za ispitivanje ciklusa se mogu (ali nije nužno) koristiti tehnike opisane u **odjeljku 2.6**.

5. Implementirati Primov algoritam.

6. Implementirati topološko sortiranje.

7. Implementirati heap sort (3/4 bodova). Dodatnih 1/4 boda ako se ne koristi pomoćni niz za izgradnju heapa.

8. U **poglavlju 5** svi algoritmi (radi jednostavnosti prezentacije ideje) sortiraju niz cijelih brojeva. Prilagoditi prezentirani kod (za jedan algoritam sortiranja po izboru) tako da radi nad bilo kakvim podacima te da funkcija sortiranja ima prototip kao ugrađena funkcija `qsort`:

    ```
    void sort(void *arr, size_t n, size_t sz, cmp_fn cmp);
    ```

    Za zamjenu sadržaja blokova memorije može se koristiti funkcija `mswap` iz prve vježbe.

    Napisati program koji će provjeriti rad funkcije sortiranjem niza struktura:

    ```
    struct pair {
      int first, second;
    };
    ```

    Usporedba struktura neka bude leksikografska.

9. Prilagoditi implementaciju quicksorta iz **poglavlja 5.6** tako da male podnizove (manje ili jednake 9 elemenata) sortira insertion sortom.

10. Implementirati neki algoritam sortiranja koji će sortirati jednostruko povezanu listu kakva je objašnjena u **odjeljku 2.4.1**.

2.3 "Kazneni" zadaci

2.3.1 Grupe

Grupa je matematička struktura $(G, \bullet)$ gdje je G skup, a $\bullet$ je operacija nad elementima iz G. Grupa ima sljedeća svojstva:

1. Zatvorenost: za sve $a, b \in G$ je i $a \bullet b \in G$.
2. Asocijativnost: za sve $a, b, c \in G$ je $a \bullet (b \bullet c) = (a \bullet b) \bullet c$.
3. Postoji element e (**identitet**) tako da je $a \bullet e = e \bullet a = a$ za sve $a \in G$.
4. Postojanje inverza: za svaki $a \in G$ postoji $b \in G$ (koji ovisi o a) tako da je $a \bullet b = b \bullet a = e$ (identitet).

Za potrebe ovog zadatka će se elementi grupe označavati *slovima*.

—— PRIMJER ——

Neka je $G = \{a, b, c\}$ te operacija $\bullet$ definirana sljedećom tablicom:

$$\begin{array}{c c} \bullet & \begin{array}{ccc} a & b & c \end{array} \\ \begin{array}{c} a \\ b \\ c \end{array} & \begin{pmatrix} a & b & c \\ b & c & a \\ c & a & b \end{pmatrix} \end{array}$$

Provjerite prema definiciji da vrijedi zatvorenost i asocijativnost te da je identitet element a. Inverz elementa c je b zato što je $c \bullet b = b \bullet c = a$. Analogno vrijedi i za inverze ostalih elemenata.

U C-u se ovakva tablica operatora može prikazati na sljedeći način:

```
char g_op[3][3] = {
  { 'a', 'b', 'c' },
  { 'b', 'c', 'a' },
  { 'c', 'a', 'b' }
};
```

Konverzija iz elementa prikazanog jednim znakom (slovom) u indeks u tablici te izračunavanje operacije nad dva elementa se lako napravi sljedećim makroima:

```
#define ETOI(x) (x - 'a')
#define OP(t, x, y) t[ETOI(x)][ETOI(y)]
```

Npr. rezultat operacije $c \bullet a$, gdje je operacija definirana u tablici `g_op`, se dobije izrazom `OP(g_op, 'c', 'a')`.

Napišite funkciju

```
char **group_read(const char *f, unsigned int *n);
```

koja će iz datoteke pročitati definiciju grupe. U varijabli `n` treba vratiti broj elemenata grupe; elementi grupe su prvih n slova abecede. Pretpostaviti da n neće biti veći od 26 (dakle, slova a-z). Rezultat funkcije treba biti dinamički alocirano polje u kojem će biti definirana operacija nad elementima.

Format ulazne datoteke započinje brojem elemenata grupe n u prvom redu, te po jedan redak tablice operacije. Elementi su prvih n slova engleske abecede, a tablica operacije ne mora nužno predstavljati grupu. U tablici se na presjeku i-tog retka i j-tog stupca nalazi rezultat operacije i-tog i j-tog elementa.

—— PRIMJER ——

Za grupu iz gornjeg primjera datoteka izgleda ovako:

```
3
abc
bca
cab
```

Neka druga grupa bi se mogla ovako zadati:

```
3
bca
cab
abc
```

Iz ove tablice se može očitati npr. da vrijedi $a \cdot b = c$ (prvi redak odgovara elementu a, drugi stupac odgovara elementu b). Također se vidi da je u ovako zadanoj tablici c identitet, te da je inverz od $a^{-1} = b$ jer vrijedi $a \cdot b = c$.

Za zadanu tablicu operacije napisati sljedeće funkcije:

1. Provjera zatvorenosti.
2. Provjera asocijativnosti.
3. Računanje elementa koji je identitet. Ako identitet ne postoji, funkcija neka vrati 0.
4. Izračunavanje inverza za zadani element. Funkcija neka vrati 0 ako element nema inverz.
5. Provjerava je li operacija **komutativna**, tj. da li za sve $a, b \in G$ vrijedi $a \bullet b = b \bullet a$.

Napisati i glavni program te osmisliti nekoliko tablica operacije kojima se može testirati ispravan rad svih funkcija.

2.3.2 (De)konvolucija

Zadana su dva niza realnih brojeva duljine m i n: $a = (a_0, a_1, \ldots, a_{m-1})$ i $b = (b_0, b_1, \ldots, b_{n-1})$. Definiramo njihovu **konvoluciju** kao niz $c = a \star b$ od $m + n - 1$ članova gdje je element c_k, $k \leq m + n - 1$, zadan formulom:

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}$$

Eksplicitan isips prvih par članova niza c glasi:

$$\begin{aligned}
c_0 &= a_0 b_0 \\
c_1 &= a_0 b_1 + a_1 b_0 \\
c_2 &= a_0 b_2 + a_1 b_1 + a_2 b_0 \\
&\vdots
\end{aligned}$$

Pri računanju sume se uzima da je $a_i = 0 \quad \forall i \geq m$ i $b_i = 0 \quad \forall i \geq n$.

Ovaj postupak je moguće obrnuti te napraviti **dekonvoluciju**: ako su zadani nizovi b i $c = a \star b$ treba izračunati niz a. Članovi niza a računaju se pojedinačno, počevši od a_0 prema sljedećim formulama:

$$\begin{aligned}
a_0 &= \frac{1}{b_0} c_0 \\
a_1 &= \frac{1}{b_0}(c_1 - a_0 b_1) \\
a_2 &= \frac{1}{b_0}(c_2 - a_0 b_2 - a_1 b_1) \\
&\vdots \\
a_k &= \frac{1}{b_0}(c_k - \sum_{i=0}^{k-1} a_i b_{k-i})
\end{aligned}$$

Pri računanju a_k, svi prethodni a_i za $i < k$ su već poznati.

Napišite program koji sa komandne linije uzima ime ulazne datoteke u sljedećem formatu:

```
r
x1 x2 ... xm
y1 y2 ... yn
```

r mora biti 1 ili -1, a za svaku drugu vrijednost treba prijaviti grešku. U iduća dva reda se nalaze članovi niza (nizovi ne moraju biti jednake duljine).

Na standardni izlaz ispisati:

- Za $r = 1$: rezultat konvolucije $z = x \star y$.
- Za $r = -1$: rezultat dekonvolucije $y = z \star x$. x i y su zadani u ulaznoj datoteci, a treba izračunati z.

Izlaz treba biti jedna linija u formatu

```
z1 z2 ... zs
```

(s elemenata rezultata).

—— Primjer ——

Za sljedeći ulaz

```
1
7 -2 3 0 -1 4 2 9 6
-3 0 0 2 1
```

program treba izračunati konvoluciju te ispisati sljedeći rezultat:

```
-21 6 -9 14 6 -8 -3 -29 -11 8 20 21 6
```

dok za ulaz

```
-1
-2 -1 -4 1 0 -6
-12 -8 -25 2 7 -33 6 -3 0 18
```

program treba izračunati dekonvoluciju te ispisati:

```
6 1 0 0 -3
```

2.3.3 Bit vektori

Za dva bit-vektora a i b definiramo *dvodimenzonalan* "produkt" na sljedeći način: $c_{ij} = D(a_i, b_j)$, gdje je D funkcija koja uzima dva bita i računa rezultat prema zadanoj tablici. Primijetite da je $D(x, y) = (xy)_2$, tj. vrijednost binarnog broja kojem je prva znamenka x, a druga y.

Napišite program koji s komandne linije uzima ime ulazne datoteke te na standardni izlaz ispisuje rezultat "produkta". U ulaznoj datoteci zadana su dva bit-vektora, po jedan u svakom redu, npr:

```
110
01101
```

Prvi bit u koloni je najmanje težine ("nulti").

Izlaz mora biti u obliku 2-D matrice koja ima onoliko redaka koliko ima bitova *prvi* bit-vektor; za prethodni primjer izlaz je:

x	y	$D(x, y)$
0	0	0
0	1	1
1	0	2
1	1	3

Tablica 2.1 Funkcija D

```
23323
23323
01101
```

Napišite i program koji radi *inverznu* operaciju: sa komandne linije uzima ime datoteke u kojoj je spremljena 2-D matrica te iz nje rekonstruira dva bit-vektora. Program mora detektirati i prijaviti nekonzistentan ulaz.

—— PRIMJER ——

Za ulaz

```
233223
011001
233223
233223
```

program mora dati sljedeće bit-vektore (u tom poretku!) kao izlaz:

```
1011
011001
```

dok za sljedeća dva ulaza (svaki u svojoj datoteci):

```
233203 233223
011001 011001
233223 232223
233223 233223
```

program mora prijaviti nekonzistentnost i to za 0. bit prvog vektora za prvi ulaz i 2. bit drugog vektora za drugi ulaz.

U ulazu će se nalaziti *0, 1 ili više* nekonzistentnosti. Ako se nalazi samo jedna, program ju *mora* moći detektirati; ako ih je više treba samo prijaviti grešku.

3 Pismeni ispiti i kolokviji

3.1 1. kolokvij 2004-03-27

1. a. Nacrtati sortirano binarno stablo nakon umetanja elemenata 7, -16, 2, 9, 3, 4, 8, -15 (tim redoslijedom).
   b. Ispisati preorder, inorder i postorder obilazak tog stabla.
   c. Definirati strukturu čvora binarnog stabla i napisati funkciju koja će u postorderu ispisati elemente stabla.

2. a. Nacrtati heap nakon umetanja elemenata (redoslijedom kako su navedeni) iz zadatka 1a.
   b. Nacrtati heap iz zadatka a nakon brisanja najvećeg elementa.
   c. Za implementaciju heapa nizom, ispisati sadržaj niza za zadatke a i b.

3. a. Definirati potrebne varijable i opisati implementaciju reda kružnim nizom.
   b. Napisati funkciju za brojanje elemenata u redu implementiranom kružnim nizom.

4. a. Definirati strukturu čvora jednostruko povezane liste s jednim krajem.
   b. Napisati funkciju za brisanje svih čvorova liste.

5. Napisati primjer $O(n^2)$ algoritma.

3.2 2. kolokvij 2004-04-24

1. Zadan je skup cijelih brojeva implementiran bit-vektorom.

   a. Napisati funkciju prototipa

   ```
   unsigned int subseq(const unsigned char *set, unsigned int n);
   ```

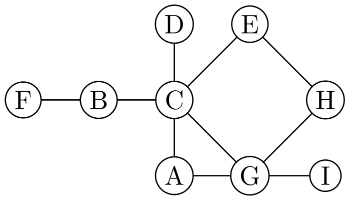
   koja u skupu `set` traži najdulji podniz uzastopnih elemenata. `n` je broj elemenata alociranih za niz; može se pretpostaviti da svi bitovi čine skup. Funkcija vraća duljinu podniza. Npr. za skup $\{7, 8, 16, 17, 18, 22, 23, 24, 26\}$ funkcija vraća 3 (najdulji podnizovi su (16, 17, 18) i (22, 23, 24)).
   b. Koliko minimalno mjesta treba alocirati za niz `unsigned char` elemenata ako će skup sadržavati cijele brojeve u intervalu $[a, b]$ uključivo?

2. Nad sortiranim nizovima (1, 7, 10, 10, 11, 12, 15, 17, 19, 26) i (2, 6, 9, 10, 18, 19, 19) opisati postupak spajanja u jedan sortirani niz (merge).

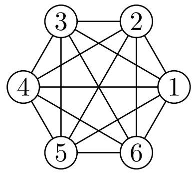
3. a. Napisati funkciju prototipa

   ```
   int merge(int *a, int na, int *b, int nb, int *c);
   ```

   za spajanje dva sortirana niza. Funkcija mora vratiti broj elemenata u izlaznom nizu `c`.
   b. Koliko minimalno mjesta mora biti alocirano za izlazni niz `c`?



zadatak 4



zadatak 5

4. Počevši od vrha A, ispisati DFS i BFS obilazak grafa sa slike.

5. Za graf na slici Kruskalovim algoritmom naći minimalno razapinjuće stablo. Težina bridova između vrhova $1 \le i, j \le 6$ zadana je formulom $d(i,j) = i + j$.

   U svakom koraku napisati koji se brid dodaje u stablo.

3.3 Pismeni ispit 2005-01-10

1. Zadana su dva polinoma:

$$p_1(x) = a_n x^n + \ldots + a_1 x + a_0$$
$$p_2(x) = b_n x^n + \ldots + b_1 x + b_0$$

   Polinomi su prikazani jednostruko vezanom listom koeficijenata, a početak liste pokazuje na koeficijent *najmanje* težine (a_0, odn. b_0). Napisati funkciju

```
struct plist {
  float coef;
  struct plist *next;
};

float prod_coef(struct plist *a, struct plist *b, int i)
```

   koja izračunava koeficijent c_i uz x^i u produktu $p_1(x)p_2(x)$ prema formuli $c_i = \sum_{j+k=i} a_j b_k$.

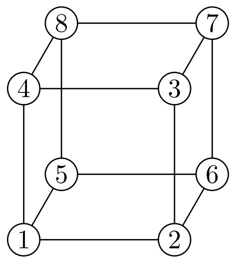
2. Zadan je aritmetički izraz $12 - (3 \cdot 4 + 8)/4 - 6$

   a. Nacrtati binarno stablo koje odgovara tom izrazu.
   b. Definirati strukturu čvora binarnog stabla izračunavanja i napisati funkciju koja izračunava aritmetički izraz zadan takvim stablom.

3. Za zadani graf ispisati:

   a. DFS obilazak stabla počevši iz vrha 1.
   b. BFS obilazak stabla počevši iz vrha 2.

4. Za zadani graf su zadane težine bridova između vrhova i i j formulom $d(i,j) = i + j$.



zadaci 3 i 4

a. Opisati Primov algoritam.
b. Primovim algoritmom izračunati minimalno razapinjuće stablo za zadani graf.

5. Opišite strukturu podataka heap i navedite primjer. Objasnite kako se heap može iskoristiti za sortiranje.

3.4 1. kolokvij 2004-02-19

1. a. (5/4) Napišite funkciju koja obrće redoslijed elemenata u jednostruko vezanoj listi. Nije dozvoljeno alociranje dodatne memorije.

   b. (1/4) Kolika je O-složenost ovog algoritma i o čemu ovisi?

2. a. (1/2) Izračunajte koliko se operacija zbrajanja obavi u sljedećem kodu:

```
for(i = 0; i < N; i++) {
  for(j = 0; j < N; j++) {
    sum += i*j;
  }
  sum += i;
}
```

   Kolika je O-složenost ovog koda i o čemu ovisi?

   b. (1/2) Napišite matematičku definiciju O-složenosti (tj. što znači da je $f(n) = O(g(n))$).

3. (1) Napišite funkciju

```
unsigned int maxel(unsigned char *v, unsigned int n);
```

   koja u bit-vektoru `v` duljine `n` `char`-ova traži bit najveće težine koji je postavljen na 1. Funkcija vraća redni broj bita.

   Funkcije za rad s bit-vektorom, a koje su potrebne za rješenje zadatka, se također moraju iskodirati!

4. (3/4) Zadana je particija skupa $A = \{1, 2, \ldots, 14\}$ u klase ekvivalencija: $A_1 = \{2, 7, 10, 12\}$, $A_2 = \{1, 3\}$, $A_3 = \{4, 6, 8, 9, 14\}$, $A_4 = \{5\}$, $A_5 = \{11, 13\}$. Napišite niz koji odgovara ovoj particiji skupa.

5. (3/4) Zadana je hash tablica veličine 17 elemenata te hash funkcija $h(x) = x \bmod 17$. U tablicu se umeću slova engleske abecede kojima su dodijeljeni kodovi prema principu A=1, B=2,..., Z=26.

   Prikažite sadržaj tablice nakon umetanja elemenata A, B, R, J, U, D. Kolizije se razrješavaju otvorenim adresiranjem i linearnim ispitivanjem.

   Potpuna engleska abeceda glasi: ABCDEFGHIJKLMNOPQRSTUVWXYZ.