### Zapiski s priprav

# Podatkovne strukture

ACM priprave na tekmovanja

Različica z dne 12. april 2025

## Kazalo

Pogosto si moramo za rešitev različnih problemov podatke organizirati drugače, kot da jih le shranimo v običajen seznam. Z uvedbo posebnih *podatkovnih struktur* lahko hitreje odgovarjamo na različna vprašanja o shranjenih informacijah, kar lahko uporabimo kot sestavno komponento v raznih algoritmih. Poglejmo si primer. Recimo, da imamo dan seznam števil dolžine $n$, nato pa bi radi $q$-krat ugotovili, če je neko število v teh seznamu.

```
#include <stdio.h>

int arr[1000];

int main() {
    int n, q;
    scanf("%d%d", &n, &q);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    for (int j = 0; j < q; j++) {
        int k;
        scanf("%d", &k);
        bool najdeno = false;
        for (int i = 0; i < n; i++) {
            if (arr[i] == k) {
                printf("Stevilo %d se pojavi v seznamu.\n", k);
                najdeno = true;
                break;
            }
        }
        if (!najdeno) {
            printf("Stevila %d ni v seznamu.\n", k);
        }
    }
    return 0;
}
```

Če si podatke le shranimo v seznam, kot zgoraj, potem bomo v najslabšem primeru za vsako od poizvedb morali preiskati celoten seznam, torej je naša rešitev $O(nq)$. Za hitrejšo rešitev bi lahko seznam na začetku uredili, nato pa pojavitve števil $k$ iskali z bisekcijo:

```
#include <stdio.h>
#include <algorithm>
```

```
3  using namespace std;
4
5  int arr[1000];
6
7  int main() {
8      int n, q;
9      scanf("%d%d", &n, &q);
10     for (int i = 0; i < n; i++) {
11         scanf("%d", &arr[i]);
12     }
13
14     sort(arr, arr+n);
15
16     for (int j = 0; j < q; j++) {
17         int k;
18         scanf("%d", &k);
19         int idx = lower_bound(arr, arr+n, k) - arr;
20         if (arr[idx] == k)
21             printf("Stevilo %d se pojavi v seznamu.\n", k);
22         else
23             printf("Stevila %d ni v seznamu.\n", k);
24     }
25
26     return 0;
27 }
```

Takšna rešitev ima časovno zahtevnost $O((n+q)\log n)$, saj za urejanje porabimo $O(n \log n)$ časa, vsaka izmed $q$ poizvedb pa se izvede v $O(\log n)$ korakih. Povedano drugače smo v zgornjem primeru opazili naslednje dejstvo: Operacija „preveri, ali se dan podatek nahaja v podatkovni strukturi“ ima zahtevnost $O(n)$ v sezamu in $O(\log n)$ v urejenem seznamu. Kaj pa lahko povemo o drugih operacijah, ki bi si jih želeli uporabljati v algoritmih?

V nadaljevanju bomo za različne podatkovne strukture obravnavali časovno zahtevnost naslednjih operacij:

(O1) dodaj podatek v podatkovno strukturo,

(O2) odstrani podatek iz podatkovne strukture,

(O3) preveri, če je podatek v podatkovni strukturi.

Število podatkov, shranjenih v podatkovni strukturi, bomo označili z $n$.

# 1 Seznami

Najbolj osnovna podatkovna struktura je običajen seznam. V računalniškem spominu je predstavljen z enim velikih blokom spomina, ki si ga lahko predstavljamo kot $n$ spremenljivk istega tipa, položenih eno za drugo. Ko deklariramo seznam, moramo nujno podati njegovo dolžino, da lahko prevajalnik rezervira blok primerne velikosti. Kakor smo povedali v uvodnem razdelku poglavja, za operacijo (O3) potrebujemo $O(n)$ časa. Pri določanju časovne zahtevnosti za operaciji (O1) in (O2) pa naletimo na težavo. Seznami namreč ne podpirajo dodajanja ali odstranjevanja podatkov — število elementov seznama je fiksno in določeno v kodi. Lahko pa se pretvarjamo, da seznam podpira ti operaciji tako, da si zapomnimo trenutno število podatkov v seznamu:

```
// V spremenljivki arr hranimo elemente seznama (in nekaj praznega
// prostora), v spremenljivki velikost pa število teh elementov.
// Začnemo s praznim seznamom
int arr[1000];
int velikost = 0;

// Funkcija doda element k na i-to mesto našega seznama
void dodaj_element(int k, int i) {
    // Zapisati želimo arr[i] = k, a tega še ne smemo narediti,
    // ker bomo s tem prepisali element, ki je tam trenutno shranjen.
    // Zato moramo prvo prekopirati vse elemente na eno mesto desno.
    for (int j = velikost; j > i; j--) {
        arr[j] = arr[j-1];
    }
    arr[i] = k;
    velikost++;  // dodali smo element, zato je naš seznam za 1 daljši
}

// Funkcija odstrani element na indeksu i iz našega seznama
void odstrani_element(int i) {
    // Če želimo odstraniti i-ti element, moramo poskrbeti, da bodo
    // vsi kasnejši elementi na pravem mestu, torej da bo prvi "prazen"
    // prostor na mestu arr[velikost], in ne prej
    for (int j = i; j < velikost; j++) {
        arr[j] = arr[j+1];
    }
    velikost--;
}
```

Iz zgornje kode je jasno razvidno, da imata tako operaciji (O2) kot (O3) časovno zahtevnost $O(n)$, saj moramo pri obeh v najslabšem primeru premakniti vse elemente.

Opazimo pa lahko tudi, da potrebujeta operaciji „dodaj na konec seznama“ in „odstrani iz konca seznama“ le konstantno mnogo časa, saj se v obeh primerih zanka takoj konča. To bo pomembno kasneje.

Druga težava z zgornjo implementacijo pa je ta, da naš seznam lahko shrani največ 1000 elementov. Seveda bi lahko pri deklaraciji `int arr[1000]` le povečali številko tako, da bi bila dovolj velika za naše potrebe, vendar tega včasih ne želimo narediti (na primer če imamo veliko majhnih seznamov in nekaj velikih, a ne vemo v naprej, kateri od teh seznamov bodo veliki). Lahko bi si namesto fiksnega seznama `arr` hranili kazalec na del spomina, kjer je seznam shranjen, ter širino tega dela spomina, ter ga po potrebi povečali, kot spodaj:

```
int *arr = nullptr;
int kapaciteta = 0;  // število elementov, ki jih lahko shranimo v arr
int velikost = 0;  // število elementov, trenutno shranjenih v seznamu

void dodaj_element(int k, int i) {
    if (velikost == kapaciteta) {
        // Ne moremo dodati novega elementa, saj imamo rezerviranega
        // premalo spomina.
        // Na začetku si torej sposodimo več spomina, in v nov spomin
        // prekopiramo stare podatke
        int *nov_arr = (int*) malloc((kapaciteta + 1) * sizeof(int));
        for (int j = 0; j < velikost; j++) {
            nov_arr[j] = arr[j];
        }
        kapaciteta++;

        // Starega bloka spomina ne potrebujemo več, zato ga vrnemo
        // operacijskemu sistemu
        if (arr != nullptr) free(arr);
        arr = nov_arr;
    }
    // Nadaljujemo tako kot prej
    for (int j = velikost; j > i; j--) {
        arr[j] = arr[j-1];
    }
    arr[i] = k;
    velikost++;
}
```

Funkcija `odstrani_element` je v tem primeru enaka kot prej. Napisali smo *dinamični seznam*, tj. seznam, ki prilagaja svojo hitrost glede na število shranjenih elementov. S

tem pa smo izgubili zgoraj omenjeno lastnost, da je dodajanje na oz. brisanje iz konca seznama $O(1)$, saj bo tekom zgornjega programa velikost vedno enaka kapaciteti, zato se bo prvi pogojni stavek v funkciji vedno aktiviral. To lastnost pa lahko še vedno v neki obliki obdržimo, če naredimo majhno spremembo:

```
void dodaj_element(int k, int i) {
    if (velikost == kapaciteta) {
        // Ne moremo dodati novega elementa, saj imamo rezerviranega
        // premalo spomina.
        // Na začetku si torej sposodimo več spomina, in v nov spomin
        // prekopiramo stare podatke
        int nova_kapaciteta = 2*kapaciteta;
        if (nova_kapaciteta == 0) nova_kapaciteta = 1;
        int *nov_arr = malloc(nova_kapaciteta * sizeof(int));
        for (int j = 0; j < velikost; j++) {
            nov_arr[j] = arr[j];
        }
        kapaciteta = nova_kapaciteta;

        // Starega bloka spomina ne potrebujemo več, zato ga vrnemo
        // operacijskemu sistemu
        if (arr != nullptr) free(arr);
        arr = nov_arr;
    }
    // Nadaljujemo tako kot prej
    for (int j = velikost; j > i; j--) {
        arr[j] = arr[j-1];
    }
    arr[i] = k;
    velikost++;
}
```

Ko `arr` povečamo, torej njegovo kapaciteto podvojimo. Kaj pa smo s tem dejansko dosegli? Ko v seznam dodamo prvi element, ne potrebujemo ničesar kopirati. Ko dodamo drugi element, moramo prekopirati prvega. Ko dodamo tretji element, moramo prekopirati prvega in drugega. Pri četrtem elementu pa nam ni treba ničesar kopirati, saj je `arr` že dovolj velik, da lahko četrti element takoj shranimo vanj. Za peti dodan element bomo morali spet prekopirati prve štiri, za šesti, sedmi in osmi element pa nam ne bo treba ničesar kopirati. Zapišimo še nekaj členov tega zaporedja (koliko elementov moramo prekopirati, ko dodamo nov element v seznam):

$$0, 1, 2, 0, 4, 0, 0, 0, 8, 0, 0, 0, 0, 0, 0, 0, 16, \ldots$$

Z nekaj razmisleka vidimo, da je vsota prvih $n$ členov tega zaporedja enaka $2n$, če je $n$ ravno potenca števila 2, oziroma $n$ v vseh ostalih primerih. V povprečju bomo torej za dodajanje enega elementa na konec seznama porabili le $O(1)$ časa. Pravimo, da je ta operacija *asimptotično konstantna* — v povprečju potrebujemo konstanten čas, da jo izvedemo, vsake toliko pa bo operacija trajala dlje.

## 1.1 Vektorji

K sreči ne potrebujemo vsakič znova pisati svojih dinamičnih seznamov, saj so že vključeni v standardni knjižnici C++ pod imenom `vector`. Poglejmo si, kako delujejo.

```
#include <stdio.h>
#include <vector>
#include <algorithm>  // sort
using namespace std;  // nujno za vector

int main() {
    vector<int> seznam;

    for (int i = 0; i < 1000; i++) {
        seznam.push_back(i);
    }

    for (int i = 0; i < seznam.size(); i++) {
        if (seznam[i] > 500) {
            seznam[i] = 7;
        }
    }

    seznam.insert(seznam.begin() + 852, 17);
    printf("%d\n", seznam[852]);
    seznam.erase(seznam.begin() + 852);
    printf("%d\n", seznam[852]);

    sort(seznam.begin(), seznam.end());
    printf("%d\n", seznam[852]);

    seznam.clear();
    return 0;
}
```

Ko deklariramo vektor, v trikotnih oklepajih (znaka za manjše oz. večje) dopišemo še tip podatkov, ki jih bomo v vektor shranjevali. V tem primeru uporabimo

kar `int`, lahko pa bi uporabili kakršenkoli drugi tip, na primer `char`, `long long`*, `vector<vector<int>>` itd. Potem nam je na voljo več metod, ki jih uporabimo tako, da napišemo ime metode za imenom spremenljivke in piko:

- metoda `size()` vrne trenutno velikost vektorja,
- metoda `push_back(x)` doda `x` na konec vektorja v $O(1)$,
- metoda `begin()` vrne iterator na začetek vektorja,
- metoda `end()` vrne iterator tik čez konec vektorja,
- metoda `insert(it, x)` doda `x` tik pred mesto `it` v $O(n)$,
- metoda `erase(it)` izbriše element na mestu `it` v $O(n)$,
- metoda `clear()` izbriše vse elemente vektorja.

*Iterator* je poseben tip podatkov, ki si ga lahko predstavljamo kot kazalec (ker tudi podpira običajne operacije kazalcev). Iterator dobimo iz metod `begin()` in `end()`, moramo pa ga podati kot argument metodama `insert()` in `erase()`. Iteratorjem lahko prištevamo številke, s čimer lahko dostopamo do poljubnega elementa vektorja (ali pa to naredimo tudi z oglatimi oklepaji).

## 1.2 Skladi

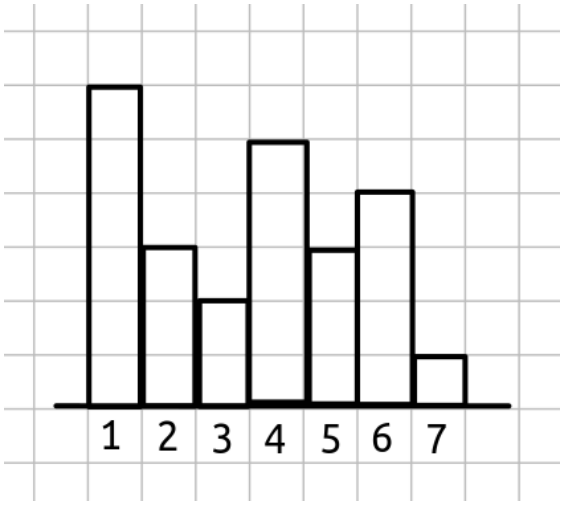
Sklad je podatkovna struktura, ki podpira le dve operaciji: „dodaj element na vrh sklada“ in „odvzemi element z vrha sklada“. Predstavljamo si, da elemente zlagamo na kup, iz katerega lahko jemljemo le vrhnji element, ker bi se kup sicer prevrnil.

```
#include <stdio.h>
#include <stack>
using namespace std;

int main() {
    stack<int> sklad;

    // Za dodajanje na sklad uporabimo metodo push()
    sklad.push(3);
    sklad.push(7);
    sklad.push(9);
    sklad.push(4);

    // Z metodo top() lahko pogledamo vrhnji element sklada
    printf("%d\n", sklad.top());

    // Za odstranjevanje s sklada uporabimo metodo pop()
```

```
    sklad.pop();
    sklad.pop();

    printf("%d\n", sklad.top());
    return 0;
}
```

Poleg metod `push()` in `pop()` za dodajanje oz. odstranjevanje elementov imamo na voljo tudi metodi `top()`, ki pogleda, kaj je shranjeno na vrhu sklada, in `size()`, ki vrne število elementov na skladu. Vse metode imajo časovno zahtevnost $O(1)$.

Poglejmo si primer naloge, kjer v rešitvi uporabimo sklad. Zadan nam je naslednji problem: Dan je seznam števil, ki opisuje višino stolpnic, postavljenih v vrsto ena za drugo. Za vsako stolpnico ugotovi, katera je njej najbližja stolpnica na levi, ki je višja od nje. Za tretjo stolpnico na spodnji sliki je to stolpnica številka 2, za četrto stolpnico pa je odgovor enak 1, saj sta stolpnici 2 in 3 nižji od stolpnice 4.




```
#include <stdio.h>

int arr[1000];

int main() {
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
        bool najdena = false;
```

```
        for (int j = i-1; j >= 0; j--) {
            if (arr[j] > arr[i]) {
                printf("%d\n", j);
                najdena = true;
                break;
            }
        }
        if (!najdena) {
            printf("Ni visje stolpnice na levi.\n");
        }
    }
    return 0;
}
```

Zgornji program bo rešil dan problem na očiten način v $O(n^2)$. Obstaja pa tudi linearna rešitev, ki se poslužuje dveh skladov. Osnovna ideja je, da si hranimo padajoče zaporedje stolpnic, ki so kandidati za najbližjo višjo stolpnico na levi. Ko pridemo do stolpnice, ki je nižja od najnižjega kandidata, lahko za njo takoj zatrdimo, da je najnižji kandidat ravno iskani odgovor. Če pa je najnižji kandidat višji od stolpnice, ki jo trenutno obravnavamo, vemo, da ne bo ta kandidat nikoli več najbližja višja stolpnica, saj je trenutna stolpnica hkrati višja od kandidata in bližje preostalim stolpnicam. Torej lahko kandidate, ki so nižji od trenutne stolpnice, kar odstranimo iz sklada.

V rešitvi imamo dve gnezdeni zanki (`while` znotraj `for`). Zakaj je potem še vedno linearna? Enostaven argument je, da bomo vsako stolpnico enkrat dodali v sklad, in jo zato največ enkrat odstranili iz sklada, torej bomo v najslabšem primeru opravili $2n$ operacij na skladu.

```
#include <stdio.h>
#include <stack>
using namespace std;

stack<int> sklad, indeksi;

int main() {
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        int t;
        scanf("%d", &t);
        while (sklad.size() > 0 && sklad.top() < t) {
            sklad.pop();
            indeksi.pop();
```

```
        }
        if (sklad.size() == 0) {
            printf("Ni visje stolpnice na levi.\n");
        } else {
            printf("%d\n", indeksi.top());
        }
        sklad.push(t);
        indeksi.push(i);
    }
    return 0;
}
```

## 1.3 Vrste

Skladom sorodne podatkovne strukture so *vrste*. Vrsta podpira le dodajanje elementov na konec in odvzemanje elementov z začetka. Tako v nasprotju v skladom pri vrstah elemente odstranjujemo v istem vrstnem redu, v katerem smo jih dodali.

V C++-u imamo na voljo podatkovni tip `queue`, ki podaja implementacijo vrste. Za dodajanje elementov v vrsto uporabljamo metodo `push()`, za odvzemanje metodo `pop()`, za dostop do prvega elementa pa metodo `front()`. Vse te operacije imajo časovno zahtevnost $O(1)$.

```
#include <stdio.h>
#include <queue>
using namespace std;

int main() {
    queue<int> vrsta;

    // Za dodajanje v vrsto uporabimo metodo push()
    vrsta.push(3);
    vrsta.push(7);
    vrsta.push(9);
    vrsta.push(4);

    // Z metodo front() lahko pogledamo prvi element vrste
    printf("%d\n", vrsta.front());  // izpiše 3

    // Za odstranjevanje iz vrste uporabimo metodo pop()
    vrsta.pop();
    vrsta.pop();
```

```
    printf("%d\n", vrsta.top());  // izpiše 9
    return 0;
}
```

Razmislimo, kako bi napisali svojo implementacijo vrste. Lahko bi uporabili razširljiv seznam iz začetka poglavja, kjer bi za dodajanje uporabili funkcijo `dodaj_element` in ji vedno podali `i = velikost-1`, za odstranjevanje elementov pa bi uporabili funkcijo `odstrani_element`, ki bi ji vedno podali argument `i = 0`. Problem je, da bi potem za odstranjevanje elementov vedno potrebovali $O(n)$ časa, saj bi vedno morali prekopirati celoten seznam.

V spodnji kodi se poslužimo naslednje ideje: Namesto, da se vrsta vedno začne na indeksu 0 tabele `arr`, bomo pustili, da se tudi indeks začetka spreminja. Potem lahko element vrste „odstranimo“ preprosto tako, da povečamo ta indeks; podatek bo sicer še vedno shranjen v spominu, a bomo mi vedeli, da ga moramo ignorirati.

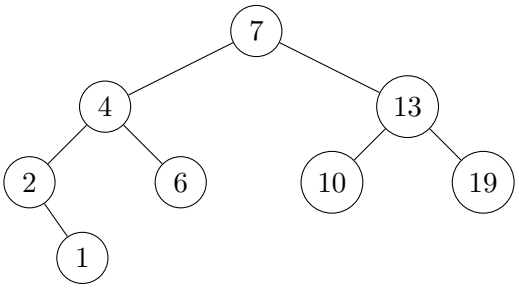
```
int *arr = nullptr;
int velikost = 0;
int kapaciteta = 0;
int zacetek = 0;

int prvi_element() {
    return arr[zacetek];
}

// V vrsto vedno dodajamo elemente na konec
void dodaj_element(int k) {
    if (velikost == kapaciteta) {
        int nova_kapaciteta = 2*kapaciteta;
        if (nova_kapaciteta == 0) nova_kapaciteta = 1;
        int *nov_arr = malloc(nova_kapaciteta * sizeof(int));

        // Prekopiramo seznam na novo lokacijo, pri čemer lahko
        // nastavimo nov začetek na 0
        for (int j = 0; j < velikost; j++) {
            nov_arr[j] = arr[(zacetek+j)%kapaciteta];
        }
        kapaciteta = nova_kapaciteta;
        zacetek = 0;

        if (arr != nullptr) free(arr);
        arr = nov_arr;
```

```
27      }
28
29      // Na kateri indeks dodamo element?
30      int i = (zacetek + velikost) % kapaciteta;
31      arr[i] = k;
32      velikost++;
33  }
34
35  // Iz vrste vedno odstranimo prvi element
36  void odstrani_element() {
37      zacetek++;
38      velikost--;
39  }
```

## 2 Množice

Vrnimo se k osrednjemu vprašanju, hitrosti operacij (O1), (O2) in (O3). V sezamu, vrsti in skladu smo podatke imeli organizirano zaporedoma, eden za drugim (kakor so v seznamu shranjeni v spominu), in s tem uspeli doseči konstantno časovno zahtevnost za dodajanje in odstranjevanje elementov, operacija (O3) pa še vedno traja $O(n)$, ker moramo pogledati vsak element posebej.

Če si podatke organiziramo v drevesno strukturo, lahko dosežemo hitrost $O(\log n)$ na vseh treh operacijah hkrati. Na primer, če želimo shraniti številke $1, 2, 4, 6, 7, 10, 13, 19$, jih lahko organiziramo kakor v spodnji sliki, tako da vsaka črta na levo pomeni manjšo številko, črta na desno pa pomeni večjo številko:




Potem lahko preverimo, če je število $k$ vsebovano v naši strukturi tako, da začnemo na vrhu (v sedmici), in se prestavimo levo, če je $k < 7$, oziroma desno, če je $k > 7$. Potem postopek ponavljamo, dokler lahko, oziroma dokler ne najdemo števila $k$. V najslabšem primeru preverimo toliko števil, kolikor je vrstic v drevesu (na zgornji sliki 4). Če števila pametno razporedimo, lahko poskrbimo, da bo višina drevesa približno $\log n$.

Tudi operaciji dodajanja in odvzemanja števil iz take strukture lahko implementiramo

tako, da bosta imeli časovno zahtevnost $O(\log n)$, vendar sta bolj kompliciran, zato se vanju ne bomo podrobno spuščali.

Takšno drevesno podatkovno strukturo v jeziku C++ uporabljajo *množice*. Vsi elementi v množici so različni — tudi če isti element dodamo večkrat, bo shranjena le ena ponovitev. Množice definiramo tako kot vektorje, na voljo pa so nam naslednje metode:

- `size()` vrne število elementov v množici,
- `clear()` izbriše vse elemente iz možice,
- `insert(x)` vstavi `x` v množico,
- `count(x)` vrne 1, če je `x` v množici, in 0 sicer,
- `erase(x)` izbriše `x` iz množice.

Poglejmo si primer uporabe.

```
#include <stdio.h>
#include <set>
using namespace std;

int main() {
    set<int> mnozica;

    mnozica.insert(1);
    mnozica.insert(2);
    mnozica.insert(2);
    mnozica.insert(4);
    mnozica.insert(6);
    mnozica.insert(6);
    mnozica.insert(6);
    mnozica.insert(7);
    mnozica.insert(10);
    mnozica.insert(10);
    mnozica.insert(13);
    mnozica.insert(19);

    printf("V mnozici je %d elementov.\n", mnozica.size());

    if (mnozica.count(10)) {
        printf("10 je v mnozici.\n");
    } else {
        printf("10 ni v mnozici.\n");
    }

```

```
    mnozica.erase(10);

    if (mnozica.count(10)) {
        printf("10 je v mnozici.\n");
    } else {
        printf("10 ni v mnozici.\n");
    }

    return 0;
}
```

Če želimo pregledati vse elemente množice, tega žal ne moremo narediti z zanko kot pri seznamih, saj ne moremo dostopati do posamičnih elementov — vprašanje „Kaj je $i$-ti element?“ sploh nima smisla pri množicah, saj elementi niso postavljeni zaporedno. Še vedno pa lahko pregledamo vse elemente v množici z zanko naslednje oblike:

```
for (auto it : mnozica) {
    printf("%d\n", *it);
}
```

V tem primeru je `it` iterator na elemente množice in ima tip `set<int>::iterator`. Da nebi toliko pisali, lahko uporabimo tudi `auto`.

## 2.1 Slovarji

Včasih želimo podatke obravnavati v parih (ključ, vrednost), in se hitro sklicati na vrednost pri danem ključu. To lahko dosežemo tako, da ustvarimo množico oblike

```
set<pair<tip_kljuca, tip_vrednosti>> mnozica;
```

kjer podamo tip za ključ in vrednosti. Potem do ključa v paru dostopamo s `par.first`, do vrednoti pa s `par.second`.

V C++ pa nam je na voljo podatkovna struktura, ki to naredi namesto nas: *slovar*. Ko ga ustvarimo, moramo podati tip za ključe in tip za vrednosti v slovarju, potem pa lahko do shranjenih vrednosti dostopamo tako, da zapišemo ključ v oglatih oklepajih.

```
#include <stdio.h>
#include <map>
using namespace std;

```

```
5  int main() {
6
7      map<char, int> slovar;
8
9      // Dodamo nekaj vrednosti
10     slovar['a'] = 3;
11     slovar['b'] = 6;
12     slovar['g'] = 7;
13     slovar['8'] = 8;
14
15     // Vse shranjene podatke lahko pregledamo s for-each zanko
16     for (auto p : slovar) {
17         // tip spremenljivke p je v tem primeru pair<char, int>
18         printf("%c: %d\n", p.first, p.second);
19     }
20
21     // Do podatkov lahko dostopamo z oglatimi oklepaji
22     printf("Vrednost pri 'g': %d\n", slovar['g']);
23
24     // Preverimo, če ima nek ključ shranjeno vrednost
25     if (slovar.count('c')) {
26         printf("'c' je v slovarju.\n");
27     } else {
28         printf("'c' ni v slovarju.\n");
29     }
30
31     // Podatke lahko odstranjujemo po ključih
32     slovar.erase('b');
33
34     // Na voljo sta nam tudi metodi lower_bound in upper_bound, ki
35     // poiščeta prvi ključ v slovarju, večji ali enak (oziroma večji)
36     // danemu iskanemu ključu.
37     // Funkciji vrneta iterator, ki kaže na par (ključ, vrednost),
38     // oziroma na slovar.end(), če ne najdeta ničesar.
39     map<char, int>::iterator it = slovar.lower_bound('c');
40     printf(
41         "Najmanjsi kljuc, vecji od 'c': %c. Vrednost = %d\n",
42         (*it).first,
43         (*it).second);
44
45     return 0;
46 }
```

Slovarje lahko uporabimo tudi kot razpršene tabele, torej kot sezname, kjer so indeksi lahko poljubno veliki ali majhni, in kjer nekateri indeksi manjkajo. Običajnega seznama tu ne moremo uporabiti, saj bi lahko potrebovali zelo veliko tabelo — če so indeksi med $-10^9$ in $10^9$, moramo imeti tabelo z več kot dvema milijardama elementov, ki bi zavzela več gigabajtov spomina. Velikost slovarja pa bo odvisna le od števila elementov, ki jih dejansko shranimo.

```cpp
#include <stdio.h>
#include <map>
using namespace std;

int main() {
    map<int, int> arr;

    // Preberemo podatke
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        int x, y;
        scanf("%d%d", &x, &y);
        arr[x] = y;  // O(log n) namesto O(1) pri seznamu
    }

    // Sedaj imamo podatke shranjene, in lahko z njimi naredimo,
    // kar algoritem zahteva.
    // Za demonstracijo jih samo izpišemo, kjer vidimo tudi nekaj
    // nove sintakse.
    for (auto[x, y] : arr) {
        printf("%d: %d\n", x, y);
    }

    return 0;
}
```

## 2.2 Neurejene množice in slovarji

Množice in slovarji, kot smo jih spoznali zgoraj, si elemente na pameten način shranijo urejene po vrsti, s čimer dosežejo časovno zahtevnost $O(\log n)$ za operacije (O1), (O2) in (O3). Obstajata pa tudi neurejeni različici teh struktur, ki podpirata iste operacije (z izjemo `lower_bound` in `upper_bound`), a imata časovno zahtevnost $O(1)$ za naštete

operacije. V delovanje teh struktur se ne bomo spuščali, lahko pa seveda uporabljamo C++ implementaciji teh struktur.

```
#include <stdio.h>
#include <unordered_set>
#include <unordered_map>
using namespace std;

int main() {
    unordered_set<int> mnozica;
    unordered_map<char, int> slovar;

    for (int i = 0; i < 20; i++) {
        mnozica.insert(2*i+7);
        slovar[i] = 2*i+7;
    }

    slovar.erase(0);
    mnozica.erase(7);

    for (int i = 0; i < 20; i++) {
        if (mnozica.count(i)) {
            printf("%d\n", slovar[i]);
        }
    }

    return 0;
}
```

## 3 Kopice

*Kopica* ali *prioritetna vrsta* deluje podobno kot običajna vrsta, le da namesto prvo dodanega elementa vedno odstranimo največji element iz kopice. Na voljo so nam podobne operacije kot pri vrstah: `push(x)` doda `x` na kopico, `pop()` odstrani največji element iz kopice, `top()` pa vrne največji element v kopici. Prvi dve našteti metodi imata časovno zahtevnost $O(\log n)$, zadnja pa $O(1)$. Da to dosežemo, tudi pri kopici elemente hranimo v drevesni strukturi, ki pa je malo drugačna kot struktura pri množicah.

```
#include <stdio.h>
#include <queue>  // priority_queue se najde v <queue>
using namespace std;
```

```cpp

int main() {
    priority_queue<int> vrsta;

    vrsta.push(3);
    vrsta.push(7);
    vrsta.push(5);

    while (vrsta.size() > 0) {
        // v prioritetni vrsti uporabimo .top() namesto .front()
        printf("%d\n", vrsta.top());
        vrsta.pop();
    }

    return 0;
}
```

Če želimo, da namesto največjega elementa delamo z najmanjšim elementom na kopici, ji moramo ob definiciji dodati še nekaj posebnih argumentov:

```cpp
priority_queue<int, vector<int>, greater<int>> vrsta;
```

## 4 Primerjava

Spoznane strukture lahko primerjamo glede na to, katere operacije podpirajo, in kako hitre so te operacije. V spodnji tabeli je prikazana časovna zahtevnost pogostih operacij na različnih strukturah. Operacija „dodaj“ pomeni, da v strukturo dodamo nek element, „poišči“ pa da preverimo, ali se dani element nahaja v naši strukturi. Ostaneta še operaciji „odstrani“ in „odstrani $x$“. Prva od njih iz strukture odstrani *nek* element — kateri element bo odstranjen je odvisno od podatkovne strukture (npr. pri skladu bo to vrhnji element, pri vrsti pa prvi element). Operacija „odstrani $x$“ pa poišče dan element $x$ v strukturi in ga odstrani, če je prisoten. Če podatkovna struktura ne podpira neke operacije, je to v tabeli označeno z X.

| Operacija | Seznam | Vektor | Sklad | Vrsta | Množica | N. množica | Kopica |
|---|---|---|---|---|---|---|---|
| dodaj | X | $O(1)$ | $O(1)$ | $O(1)$ | $O(\log n)$ | $O(1)$ | $O(\log n)$ |
| odstrani | X | X | $O(1)$ | $O(1)$ | X | X | $O(\log n)$ |
| odstrani $x$ | X | $O(n)$ | X | X | $O(\log n)$ | $O(1)$ | X |
| poišči | $O(n)$ | $O(n)$ | X | X | $O(\log n)$ | $O(1)$ | X |