Zahtevnejše podatkovne strukture

Luka Fürst

Zahtevnejše podatkovne strukture

- dvojiško iskalno drevo
- predponsko drevo (*trie*)
- segmentno drevo

Zahtevnejše podatkovne strukture

Dvojiško iskalno drevo

Dvojiško iskalno drevo

- implementacija množice z elementi, ki pripadajo tipu, za katerega je mogoče definirati urejenost
- učinkovito dodajanje elementov
- učinkovito brisanje elementov
- učinkovito preverjanje prisotnosti elementov
- učinkovito izvajanje operacij, ki temeljijo na urejenosti
  - iskanje minimuma in maksimuma
  - iskanje k-tega najmanjšega elementa
  - določanje položaja elementa v urejenem zaporedju

Dvojiško iskalno drevo vs. zgoščena tabela

- zgoščena tabela
  - dodajanje, brisanje in iskanje: $O(n \,/\, b)$
    - b: število predalčkov
    - $O(1)$ v primeru $n = O(b)$
  - operacije, ki temeljijo na urejenosti: $O(n)$
- dvojiško iskalno drevo
  - vse operacije v $O(\log n)$ (v povprečju)

Jezikovna podpora

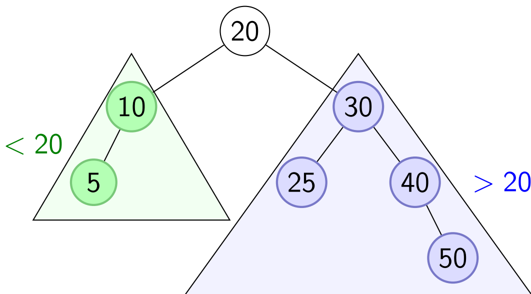
- `set` v C++
- `TreeSet` v javi
- podpora osnovnim operacijam
  - dodajanje, brisanje, preverjanje prisotnosti
  - iskanje minimuma / maksimuma
- če želimo kaj več, implementiramo svoje drevo

Dvojiško iskalno drevo

- možnost 1
  - prazno drevo (drevo brez elementov)
- možnost 2
  - koren + levo poddrevo + desno poddrevo
  - vsi elementi v levem poddrevesu so manjši od elementa v korenu
  - vsi elementi v desnem poddrevesu so večji od elementa v korenu
  - levo in desno poddrevo sta tudi dvojiški iskalni drevesi

Dvojiško iskalno drevo

- primer



- relaciji levo poddrevo $<$ koren in desno poddrevo $>$ koren veljata za vsako vozlišče
  - $25 < 30$, $40 > 30$, $50 > 30$
  - $50 > 40$
  - $5 < 10$

Implementacija v C++

- struktura/razred za predstavitev vozlišča

```
struct Vozlisce {
    int vrednost;    // vrednost (element) v vozlišču
    Vozlisce* levo;  // kaz. na levega otroka (nullptr, če ga ni)
    Vozlisce* desno; // kaz. na desnega otroka (nullptr, če ga ni)

    Vozlisce(int v, Vozlisce* l = nullptr, Vozlisce* d = nullptr) {
        vrednost = v;
        levo = l;
        desno = d;
    }
};
```

- struktura/razred za predstavitev drevesa

```
struct Drevo {
    Vozlisce* koren; // koren drevesa (nullptr, če je drevo prazno)

    Drevo() {
        koren = nullptr;
    }
};
```

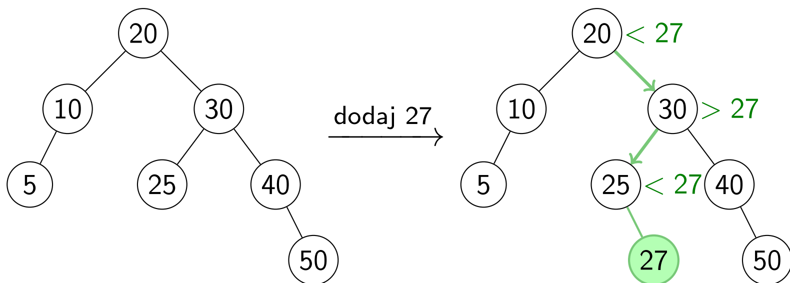
Preverjanje prisotnosti elementa x

- pričnemo v korenu
- vrednost v trenutnem vozlišču (t) primerjamo z x
  - $x = t \implies$ našli smo ga!
  - $x < t \implies$ premakni se v levega otroka
  - $x > t \implies$ premakni se v desnega otroka

```
bool obstaja(int vrednost) {
    Vozlisce* v = koren;
    while (v) {   // v != nullptr
        if (v->vrednost == vrednost) {
            return true;
        }
        if (vrednost > v->vrednost) {
            v = v->desno;
        } else {
            v = v->levo;
        }
    }
    return false;
}
```

Dodajanje elementa

- podobno iskanju
- novo vozlišče vedno dodamo kot list



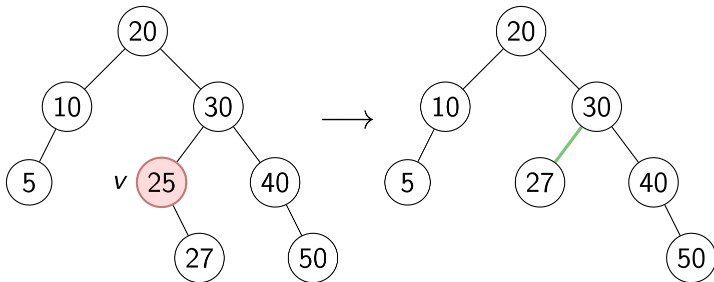
Dodajanje elementa

```
// Doda <stevilo> v drevo s korenom <v> in vrne novi koren drevesa.
Vozlisce* dodaj(Vozlisce* v, int stevilo) {
    if (!v) {
        // vstavljanje v prazno drevo
        v = new Vozlisce(stevilo);
        return v;
    }

    // vstavljanje v desno oz. levo poddrevo
    if (stevilo > v->vrednost) {
        v->desno = dodaj(v->desno, stevilo);
    } else {
        v->levo = dodaj(v->levo, stevilo);
    }
    return v;
}

void dodaj(int stevilo) {
    koren = dodaj(koren, stevilo);
}
```

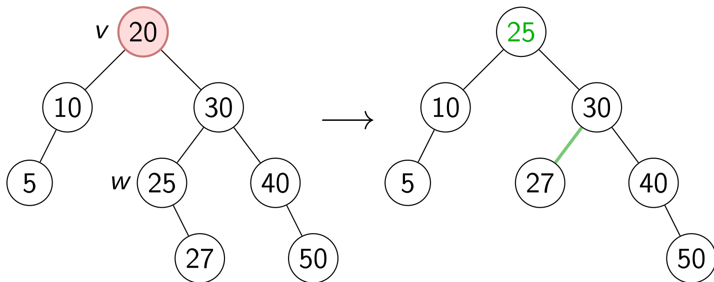
Brisanje elementa

- v: vozlišče, ki vsebuje iskani element
- v je list
  - v preprosto izbriši
- v ima enega otroka
  - v je koren $\implies$ otrok v postane novi koren, izbriši v
  - sicer pripni otroka v na starša v in izbriši v
- v ima dva otroka
  - w: skrajno desno vozlišče v levem poddrevesu (predhodnik v) ali skrajno levo vozlišče v desnem poddrevesu (naslednik v)
  - `v->vrednost = w->vrednost;`
  - izbriši w (w ima kvečjemu enega otroka)

Brisanje elementa: v ima enega otroka



Brisanje elementa: v ima dva otroka



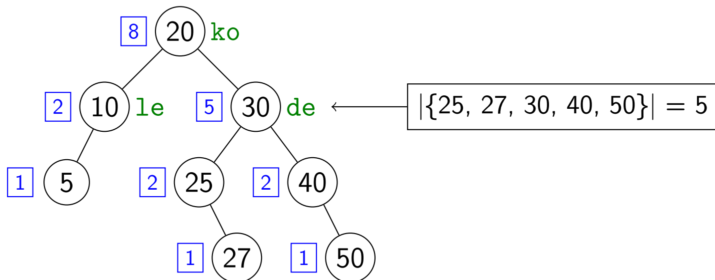
Minimum, maksimum, urejeni izpis

- minimum: skrajno levi element
- maksimum: skrajno desni element
- izpis v urejenem vrstnem redu

```
void izpisi(Vozlisce* v) {
    if (v) {
        izpisi(v->levo);
        cout << v->vrednost << endl;
        izpisi(v->desno);
    }
}
```

k-ti najmanjši element, položaj elementa

- najmanjši element: mesto 0; drugi najmanjši: mesto 1; ...
- vsako vozlišče opremimo z velikostjo njegovega poddrevesa



- `ko` je na mestu `ko->levo->stPotomcev`
- `le` je na mestu `le->levo->stPotomcev`
- `de` je na mestu
  `de->levo->stPotomcev + ko->levo->stPotomcev + 1`

k-ti najmanjši element

```
// Vrne vozlišče s k-tim najmanjšim elementom v drevesu s korenom <v>.
Vozlisce* poisciKtega(Vozlisce* v, int k) {
    if (k < 0 || !v) {
        return nullptr;
    }
    int stLevihPotomcev = v->levo ? v->levo->stPotomcev : 0;

    if (stLevihPotomcev == k) {
        // natanko k vozlišč ima manjšo vrednost kot vozlišče <v>,
        // zato je iskano vozlišče kar <v>
        return v;
    }
    if (k < stLevihPotomcev) {
        // poišči k-ti element v levem poddrevesu
        return poisciKtega(v->levo, k);
    }

    // poišči ustrezni element v desnem poddrevesu;
    // upoštevamo, da je (stLevihPotomcev + 1) elementov manjših od
    // vrednosti v korenu desnega poddrevesa
    return poisciKtega(v->desno, k - stLevihPotomcev - 1);
}
```

Položaj elementa

```
// Vrne mesto podanega elementa v drevesu s korenom <v>.
// Ob prvem klicu naj bo pristevek = 0.
int mesto(Vozlisce* v, int element, int pristevek) {
    if (!v) {
        return -1;
    }
    int stLevihPotomcev = v->levo ? v->levo->stPotomcev : 0;

    if (v->vrednost == element) {
        // element smo našli
        return stLevihPotomcev + pristevek;
    }

    if (element < v->vrednost) {
        // poišči element v levem poddrevesu
        return mesto(v->levo, element, pristevek);
    }

    // poišči element v desnem poddrevesu; upoštevaj, da je
    // (stLevihPotomcev + 1) elementov manjših od vrednosti v korenu
    return mesto(v->desno, element, pristevek + stLevihPotomcev + 1);
}
```

Časovna zahtevnost

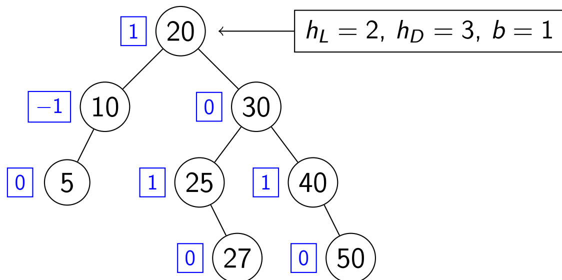
- $O(h)$, kjer je h višina drevesa (za vse operacije)
- v idealnem primeru
  - drevo višine h ima $2^{h+1} - 1$ vozlišč
  - drevo z n vozlišči ima višino $\lceil \log_2(n+1) \rceil - 1$
  - $O(h) = O(\log n)$
- v povprečju
  - prav tako $O(\log n)$
- v najslabšem primeru
  - drevo se izrodi v seznam $\implies$ $O(n)$

Dvojiška iskalna drevesa z $O(\log n)$ v najslabšem primeru

- AVL-drevo
- rdeče-črno drevo
- B-drevo
- 2-3-drevo
- ...

AVL-drevo

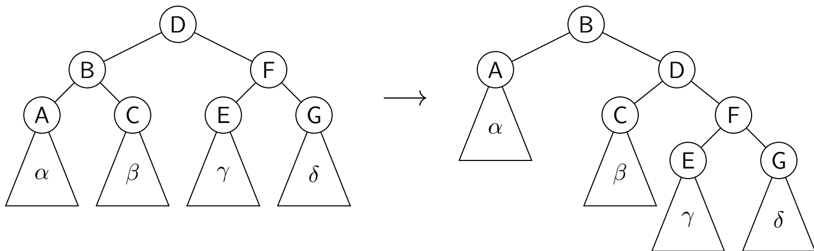
- dvojiško iskalno drevo, v katerem za vsako vozlišče v velja $-1 \leq b(v) \leq 1$, pri čemer
  - $h_L(v)$ = višina levega poddrevesa vozlišča v
  - $h_D(v)$ = višina desnega poddrevesa vozlišča v
  - $b(v) = h_D(v) - h_L(v)$ (uravnoteženost vozlišča)



Rotacija

- operacija, ki preoblikuje drevo, a ohrani lastnost »iskalnosti«
- desna rotacija drevesa s korenom `ko`

```
Vozlisce* le = ko->levo;
Vozlisce* lede = le->desno;
le->desno = ko;
ko->levo = lede;
return le;    // to je novi koren drevesa
```



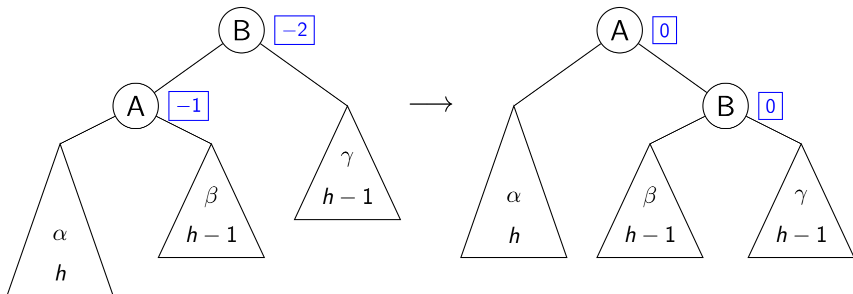
- leva rotacija poteka analogno

AVL-drevo

- vse operacije izvajamo tako kot pri navadnem dvojiškem iskalnem drevesu
- če pri vozlišču v uravnoteženost postane > 1 ali < -1, potem
    - na poddrevesu s korenom v izvedemo eno ali dve rotaciji
    - po potrebi (rekurzivno) ponovimo postopek na staršu vozlišča v

Rotacije pri AVL-drevesu

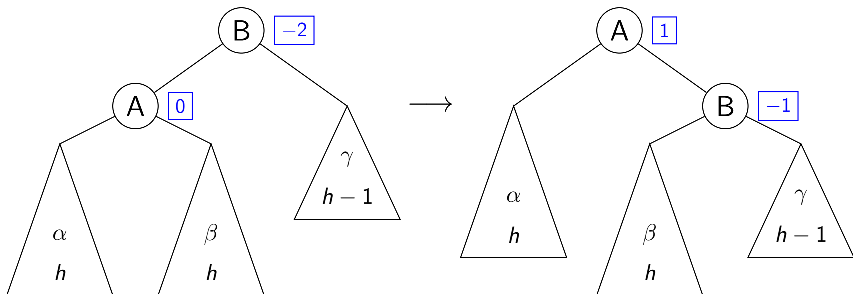
- primer 1 (pri dodajanju ali brisanju):
  - $b(v) = -2$, $b(v\texttt{->levo}) = -1$
  - izvedemo desno rotacijo drevesa s korenom v



- analogen primer: $b(v) = 2$, $b(v\texttt{->desno}) = 1$

Rotacije pri AVL-drevesu

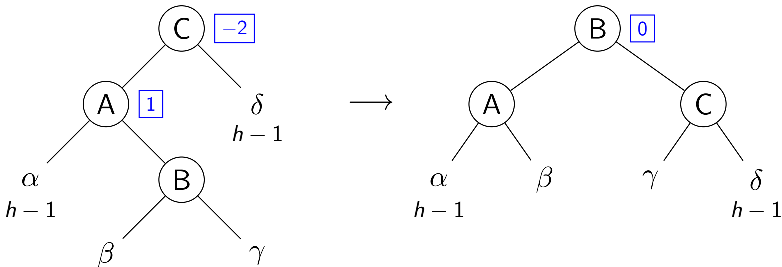
- primer 2 (samo pri brisanju):
  - $b(v) = -2$, $b(v\texttt{->levo}) = 0$
  - izvedemo desno rotacijo drevesa s korenom v



- analogen primer: $b(v) = 2$, $b(v\texttt{->desno}) = 0$

Rotacije pri AVL-drevesu

- primer 3 (pri dodajanju ali brisanju):
  - $b(v) = -2$, $b(v\texttt{->levo}) = 1$
  - najprej izvedemo levo rotacijo drevesa s korenom v`->levo`, nato pa desno rotacijo drevesa z novim korenom v`->levo`



- eden od β in γ je visok $h - 1$, drugi pa $h - 1$ ali $h - 2$
- analogen primer: $b(v) = 2$, $b(v\texttt{->desno}) = -1$

Predponsko drevo

Predponsko drevo

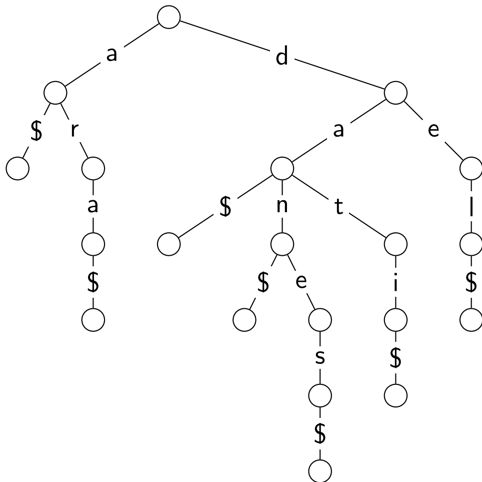
- *trie* (izgovori se enako kot *try*)
- dvojiško iskalno drevo za hrambo nizov
- učinkovito dodajanje, brisanje in iskanje nizov
- učinkovito izvajanje operacij, ki temeljijo na urejenosti
  - k-ti po abecedi
  - položaj shranjenega niza v abecednem vrstnem redu
- učinkovito iskanje nizov po predponah
  - koliko nizov se začne na določeno predpono?

Predponsko drevo

- abeceda z m znaki + znak \$, ki ne pripada abecedi
- v vsakem vozlišču hranimo tabelo $m + 1$ kazalcev
- i-ti kazalec v vsakem vozlišču predstavlja i-ti znak abecede
- j-ti nivo drevesa ustreza j-temu mestu v shranjenih nizih
- niz $a_1 a_2 \dots a_n$ je predstavljen s potjo od korena do lista, sestavljeno iz povezav $\langle a_1, a_2, \dots, a_n, \$ \rangle$

Primer

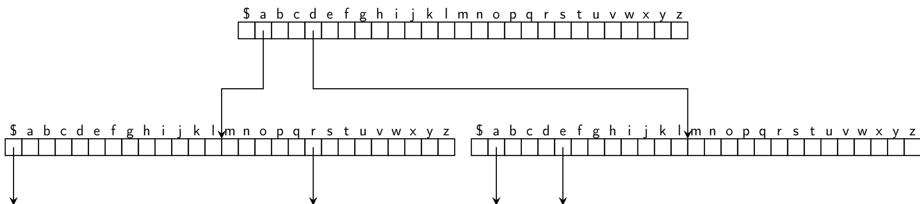
- abeceda = {a, b, . . . , z}
- drevo z besedami a, ara, da, dan, danes, dati, del



Implementacija v C++

```
struct Vozlisce {
    vector<Vozlisce*> otroci;   // otroci vozlišča

    Vozlisce() {
        otroci = vector<Vozlisce*>(27);
    }
};

struct Trie {
    Vozlisce* koren;  // koren drevesa
};

// Vrne referenco na otroka, ki pripada podanemu znaku.
// INDEKSI['$'] == 0, INDEKSI['a'] == 1, INDEKSI['b'] == 2 itd.
Vozlisce*& otrok(char znak) {
    return otroci[INDEKSI[znak]];
}
```

Vrhnji del prejšnjega primera (prikaz vektorjev kazalcev)



Iskanje, dodajanje, brisanje

- vse tri operacije temeljijo na preprostem sprehodu od korena proti listom
- $O(d)$, kjer je d dolžina niza, ki ga iščemo / dodajamo / brišemo

Iskanje

```
// Preveri, ali v drevesu obstaja beseda (če se konča z $) ali
// predpona (če se ne konča z $).

bool obstaja(const string& beseda) {
    // pričnemo v korenu
    Vozlisce* v = koren;

    // za vsak znak besede se premaknemo v pripadajočega otroka trenutnega
    // vozlišča; če vozlišče ne obstaja, vemo, da besede ni v drevesu
    for (char znak: beseda) {
        v = v->otrok(znak);
        if (!v) {
            return false;
        }
    }

    // zdaj smo v vozlišču, ki pripada besedi oz. predponi
    return true;
}
```

Dodajanje

```
// Doda besedo v drevo.  Beseda se mora končati z znakom $.
void dodaj(const string& beseda) {
    // pričnemo v korenu
    Vozlisce* v = koren;

    // za vsak znak besede se premaknemo v pripadajočega otroka
    // trenutnega vozlišča; če vozlišče ne obstaja, ga ustvarimo
    for (char znak: beseda) {
        if (!v->otrok(znak)) {
            v->otrok(znak) = new Vozlisce();
        }
        v = v->otrok(znak);
    }
}
```

Štetje pojavitev

- predponsko drevo zlahka prilagodimo za štetje pojavitev besed ali njihovih predpon

```
struct Vozlisce {
    vector<Vozlisce*> otroci;
    int kolikokrat;

    Vozlisce() {
        otroci = vector<Vozlisce*>(27);
    }

    Vozlisce*& otrok(char znak) {
        return otroci[INDEKSI[znak]];
    }
};
```

Štetje pojavitev

```
void dodaj(const string& beseda) {
    Vozlisce* v = koren;
    for (char znak: beseda) {
        if (!v->otrok(znak)) {
            v->otrok(znak) = new Vozlisce();
        }
        v = v->otrok(znak);
        v->kolikokrat++;
    }
}

int steviloPojavitev(const string& beseda) {
    Vozlisce* v = koren;
    for (char znak: beseda) {
        v = v->otrok(znak);
        if (!v) {
            return 0;
        }
    }
    return v->kolikokrat;
}
```

Izboljšave

- problem: potencialno velika poraba prostora
- v praksi sicer ni nujno tako hudo
  - Fran Saleški Finžgar, Pod svobodnim soncem (`sonce.txt`)
  - brez ločil, šumniki $\rightarrow$ sičniki, brez besed z znaki izven abecede a..z
  - skupaj 139076 besed, maks. dolžina = 17, povprečna dolžina = 4.56
  - predponsko drevo zasede okrog 16 MB prostora
- izboljšave
  - v vozlišču hranimo povezani seznam namesto vektorja
  - stiskanje nerazvejanih vej
  - drevo Patricia

Sorodne podatkovne strukture

- priponsko drevo (*suffix tree*)
  - (stisnjeno) preponsko drevo, ki hrani vse pripone vseh nizov
  - omogoča hitro iskanje po besedilu in številne druge operacije
- priponska tabela (*suffix array*)
  - enostavnejša in prostorsko varčnejša podatkovna struktura s podobnimi zmogljivostmi

Segmentno drevo

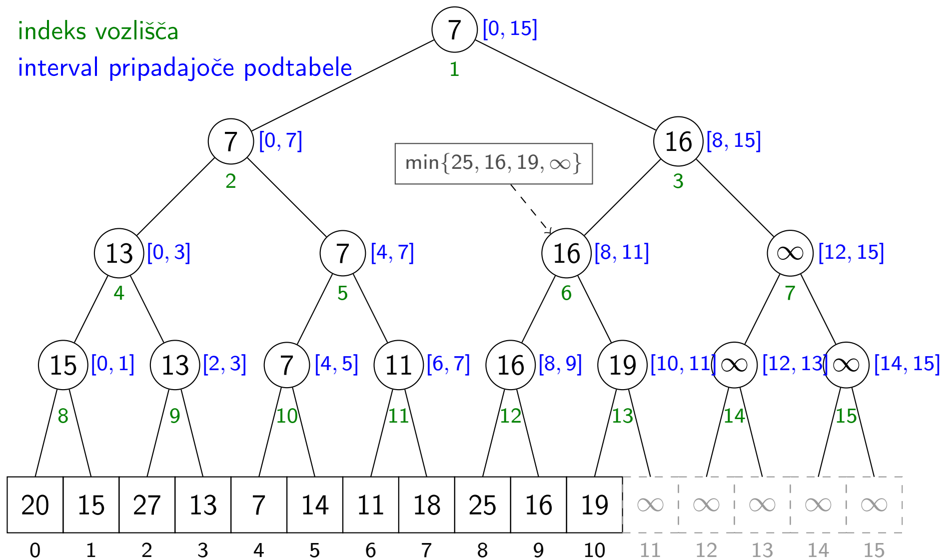
Segmentno drevo

- drevo, ki omogoča dinamične intervalne poizvedbe na podani tabeli
  - vrednost ali indeks največjega ali najmanjšega elementa v podani podtabeli (*range maximum/minimum query*, RMQ)
  - vsota elementov podane podtabele (*range sum query*, RSQ)
- smiselno, kadar se elementi tabele spreminjajo in kadar nimamo poizvedb po vsotah podtabel
  - statični podatki + RSQ $\Longrightarrow$ kumulativna tabela
  - statični podatki + RMQ $\Longrightarrow$ redka tabela (*sparse table*)
  - dinamični podatki + RSQ $\Longrightarrow$ Fenwickovo drevo

Segmentno drevo

- izdelamo ga nad tabelo t dolžine n
  - zaradi enostavnosti naj bo $n = 2^m$ (to sicer ni nujno)
- polno uravnoteženo drevo z $(m+1)$ nivoji (višina $= m$)
  - vsi nivoji so v celoti zapolnjeni
  - zadnji nivo je kar originalna tabela
- koren hrani rezultat operacije (npr. minimum) na celotni tabeli
- vozlišče i ($i = 0$: skrajno levo) na nivoju j ($j = 0$: koren) hrani rezultat operacije na podtabeli $[2^{m-j}i, 2^{m-j}(i+1) - 1]$

Segmentno drevo za iskanje minimuma



Implementacija segmentnega drevesa

- celotno segmentno drevo (skupaj originalno tabelo) hranimo v tabeli dolžine $2n = 2^{m+1}$
- indeks 0 zanemarimo
- indeksi $[2^j, 2^{j+1} - 1]$: j-ti nivo segmentnega drevesa
  - nivo m = originalna tabela
- levi otrok vozlišča i = vozlišče $2i$
- desni otrok vozlišča i = vozlišče $2i + 1$
- starš vozlišča i = vozlišče i / 2

Implementacija

```
class Drevo {
    int velOrig;  // vel. orig. tabele, zaokrožena navzgor na potenco 2
    vector<int> elementi;   // elementi drevesa

    public:

    Drevo(const vector<int>& t) {
        // 8 * sizeof(int) - __builtin_clz(n) = št. bitov števila n
        velOrig = 1 << (8 * sizeof(int) - __builtin_clz(t.size() - 1));
        elementi = vector<int>(2 * velOrig, -1);
        zgradi(t, 1, 0, velOrig - 1);
    }
    int L(int i) {  // indeks levega otroka vozlišča i
        return 2 * i;
    }
    int D(int i) {  // indeks desnega otroka vozlišča i
        return 2 * i + 1;
    }
    int manjsi(int a, int b) {  // manjši izmed a in b (-1 predstavlja ∞)
        return (a < 0) ? (b) : (b < 0 ? a : min(a, b));
    }
}
```

Gradnja

- gradnja drevesa = polnjenje tabele
- drevo s korenom z indeksom i zgradimo tako:
  - če je vozlišče i list, je njegova vrednost kar pripadajoča vrednost originalne tabele
  - sicer rekurzivno zgradimo levo in desno poddrevo vozlišča i, vrednost vozlišča i pa določimo kot minimum njegovih dveh otrok
- $O(n)$

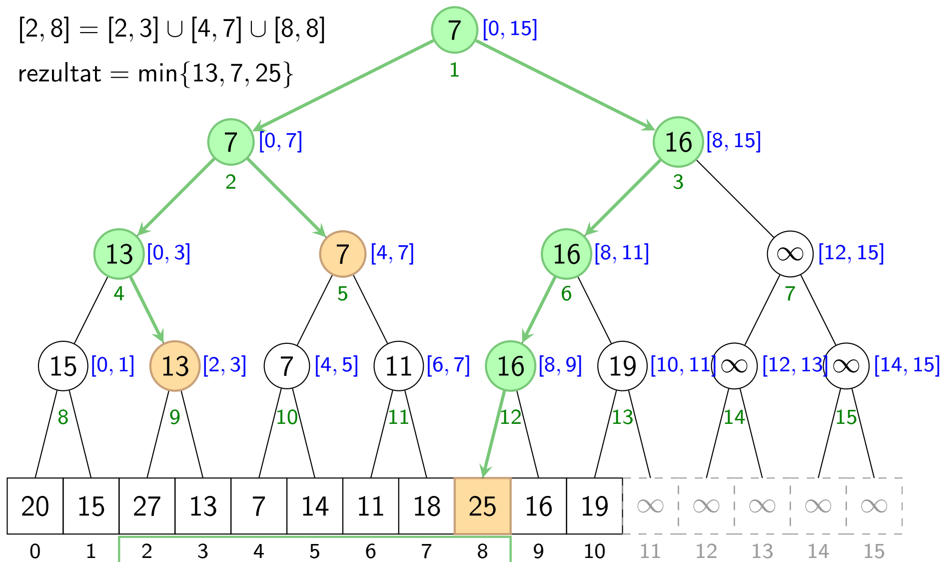
Gradnja

```
// ind: indeks trenutnega vozlišča
// lv, dv: meji podtabele orig. tabele, ki pripada trenutnemu vozlišču
void zgradi(const vector<int>& t, int ind, int lv, int dv) {
    if (lv == dv) {
        // smo na zadnjem nivoju (kopija originalne tabele)
        if (lv < t.size()) {
            elementi[ind] = t[lv];
        }
    } else {
        // zgradimo levo in desno poddrevo trenutnega vozlišča,
        // vsebina trenutnega vozlišča pa je minimum korenov obeh
        // poddreves
        int sredina = (lv + dv) / 2;
        zgradi(t, L(ind), lv, sredina);
        zgradi(t, D(ind), sredina + 1, dv);
        elementi[ind] = manjsi(elementi[L(ind)], elementi[D(ind)]);
    }
}
```

Poizvedbe

- zanima nas rezultat operacije (npr. minimum) na intervalu $[a, b]$
- naj bo $[l_v, d_v]$ interval (levi in desni indeks v okviru originalne tabele), ki ga pokriva vozlišče v
- naj bo $s_v = \lfloor (l_v + d_v) \,/\, 2 \rfloor$
- pričnemo v korenu drevesa (to je začetni v)
- če je interval vozlišča v celoti vsebovan v intervalu poizvedbe ($[l_v, d_v] \subseteq [a, b]$), vrnemo kar vrednost v vozlišču v
- če levi otrok pokriva vsaj del intervala poizvedbe ($[l_v, s_v] \cap [a, b] \neq \emptyset$), se usmerimo v levega otroka
- če desni otrok pokriva vsaj del intervala poizvedbe ($[s_v + 1, d_v] \cap [a, b] \neq \emptyset$), se usmerimo (tudi) v desnega otroka
- na vsakem nivoju obiščemo največ 4 vozlišča $\implies$ $O(\log n)$

Primer poizvedbe: minimum na intervalu [2, 8]



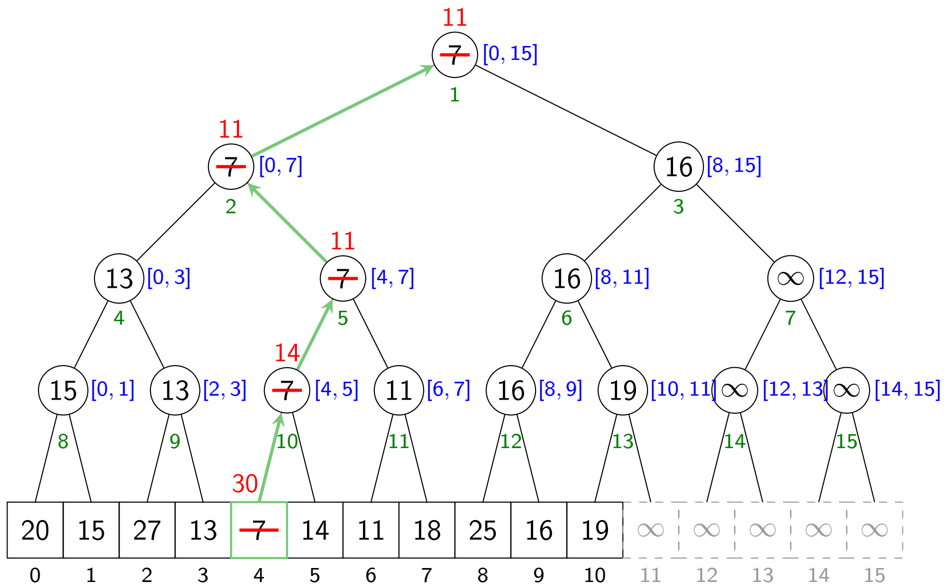
Poizvedbe

```
// a, b: interval poizvedbe
// lv, dv: interval, ki ga pokriva trenutno vozlišče
// ind: indeks trenutnega vozlišča
int minimum(int a, int b, int ind, int lv, int dv) {
    if (a <= lv && b >= dv) {   // [l_v, d_v] ⊆ [a, b]
        return elementi[ind];
    }
    int sv = (lv + dv) / 2;
    int rezultat = -1;
    if (a <= sv) {  // [l_v, s_v] ∩ [a, b] ≠ ∅
        rezultat = manjsi(rezultat, minimum(a, b, L(ind), lv, sv));
    }
    if (b > sv) {  // [s_v + 1, d_v] ∩ [a, b] ≠ ∅
        rezultat = manjsi(rezultat, minimum(a, b, D(ind), sv + 1, dv));
    }
    return rezultat;
}

int minimum(int a, int b) {
    return minimum(a, b, 1, 0, velOrig - 1);
}
```

Posodabljanje

- posodobitev elementa originalne tabele na podanem indeksu
- sprehodimo se od lista (podanega elementa originalne tabele) do korena in v vsakem vozlišču ponovno izračunamo minimum njegovih dveh otrok
- $O(\log n)$

Primer posodobitve: $t[4] := 30$



Posodabljanje

```
// Element originalne tabele na podanem indeksu nastavi na podano
// vrednost in posodobi segmentno drevo.
void posodobi(int indeks, int vrednost) {
    int ix = velOrig + indeks;
    elementi[ix] = vrednost;
    ix /= 2;

    // posodabljamo od lista proti korenu
    while (ix > 0) {
        elementi[ix] = manjsi(elementi[L(ix)], elementi[D(ix)]);
        ix /= 2;
    }
}
```