Gomila i prioritetni redovi

Cindrić, Luka

ODJEL ZA INFORMATIKU
SVEUČILIŠTE U RIJECI

Rijeka, 8. veljače 2021.

Zadatak za završni rad

Pristupnik: **Luka Cindrić**

Naziv završnog rada: **Gomila i prioritetni redovi**

Naziv završnog rada na eng. jeziku: **Heap and priority queues**

Sadržaj zadatka:

Zadatak završnog rada je dati teorijski prikaz osnovnih algoritama sortiranja i posebno struktura podataka prioritetni red i gomila te objasniti i ilustrirati izvođenje osnovnih operacija na tim strukturama podataka. Potrebno je predstaviti postupke koji se izvode na gomile i uz ilustraciju postupaka diskutirati njihovu učinkovitost.

Mentor

Prof. dr. sc. Maja Matetić

Voditelj za završne radove

doc. dr. sc. Miran Pobar

Zadatak preuzet: 8. veljača 2021.

(potpis pristupnika)

Sveučilište u Rijeci – Odjel za informatiku

Jednopredmetna informatika

Luka Cindrić

Gomila i prioritetni redovi

Završni rad

Mentor: Prof. dr. sc. Maja Matetić

Rijeka, 15.07.2021.

Sažetak

Cilj istraživanja je proučiti napredne algoritme koji imaju široku primjenu. Izvori kojim sam se koristio nisu zastarjeli te se još uvijek koriste za učenje i proučavanje. Istražujući algoritme prioritetnog reda i gomile može se vidjeti široka primjena i njihova važnost, bila u sortiranju ili sl.

Ključne riječi: prioritetni red, gomila, sortiranje

Sadržaj

1. Uvod

Zašto su prioritetni redovi te njihova implementacija uporabom gomile zanimljivi?

Ukoliko želimo dovoditi na ulaz aplikacije neke podatke tako da svaki od njih ima određenuvažnost odnosno vrijednost, poslužit ćemo se algoritmom prioritetnih redova. Svakom elementu dodjeljujemo neku vrijednost koja označava njegov prioritet.

Nadalje, mi nećemo samo promatrati prioritetni red, nego i njegovu implementaciju zvanu **gomila**.

Gomila nam daje vrlo brz i efikasan algoritam sortitanja, sortiranje uporabom gomile (engl. **heap sort**).

2. Prioritetni red

Prioritetni red je apstraktni tip podataka za spremanje kolekcije prioritetnih elemenata koji podržava proizvoljno umetanje elemenata, tj. element s najvećim prioritetom ima prednost pri izlasku iz reda u bilo kojem vremenu.

2.1. Ključevi

Aplikacije obično zahtijevaju usporedbu i rangiranje prema parametrima ili svojstvima, koji se nazivaju ključevi, a koji se dodijeljuju svakom objektu u zbirci. Formalno, ključ definiramo kao objekt koji je elementu dodijeljen kao specifični atribut za taj element i koji se može koristiti za identificiranje ili rangiranje tog elementa. Ključ je elementu dodijeljen obično od strane korisnika ili aplikacije, stoga ključ može predstavljati svojstvo koje element izvorno nije posjedovao. Ključ dodijeljen elementu ne treba nužno biti jedinstven, ali bilo bi poželjno. Ključ može biti jednostavan broj, kao i neko složenije svojstvo. Npr. prioritet putnika na čekanju je inače realiziran gledajući razne faktore kao što su faktor učestalosti letača, plaćene vozarine i vrijeme dolaska.

Ime „prioritetni red“ dolazi iz činjenice da ključevi određuju „prioritet“ koji se koristi za odabir elemenata koji će biti uklonjeni.

Osnovne operacije prioritetnog reda su:

(i) insert(e): umetanje elementa *e* u prioritetni red
(ii) min(): vraća element prioritetnog reda povezan s ključem najmenje vrijednosti, tj. element čiji ključ je manji ili jednak svim ostalim ključevima prioritetnog reda
(iii) removeMin(): uklanja iz prioritetnog reda element min()

2.2. Komparatori

Vrlo važno pitanje je kako definirati način za usporedbu ključeva povezanih sa svakim elementom. Najopćenitiji pristup, zvan **metoda kompozicije**, temelji se na definiranju svakog ulaza prioritetnog reda da bude par *(e,k)* koji se sastoji od elementa *e* i ključa *k*. Dio koji sadrži element pohranjuje podatke, a dio s ključem pohranjuje informacije koje određuju prioritetni redoslijed. Svaki ključni objekt definira vlastitu funkciju kompozicije.

2.3. Funkcije

Funkcije koje podržava prioritetni red *P* su:

(i) size(): vraća broj elemenata *P*
(ii) empty(): vraća true ako je *P* prazan i false ako nije
(iii) insert(*e*): umeće novi element *e* u *P*
(iv) min(): vraća referencu na element *P* s najmanjim ključem; pojavljuje se greška ako je *P* prazan
(v) removeMin(): uklanja iz *P* element referenciran s min(); pojavljuje se greška ako je *P* prazan

2.4. Sortiranje

Još jedna bitna primjena prioritetnih redova je sortiranje, gdje nam je dostupna kolekcija *L* od *n* elemenata koji mogu biti uspoređeni metodom kompozicije. Algoritam sortiranja *L* prioritetnim redom *Q*, zvan **PriorityQueueSort**, sastoji se od dva koraka:

(i) U prvom koraku, umećemo elemente *L* u prioritetni red *Q* koji je početno bio prazan
(ii) U drugom koraku, izvlačimo elemente iz *Q* u *n* kombinacija *removeMin()* i *min()* operacija, te ih vraćamo natrag u *L*.

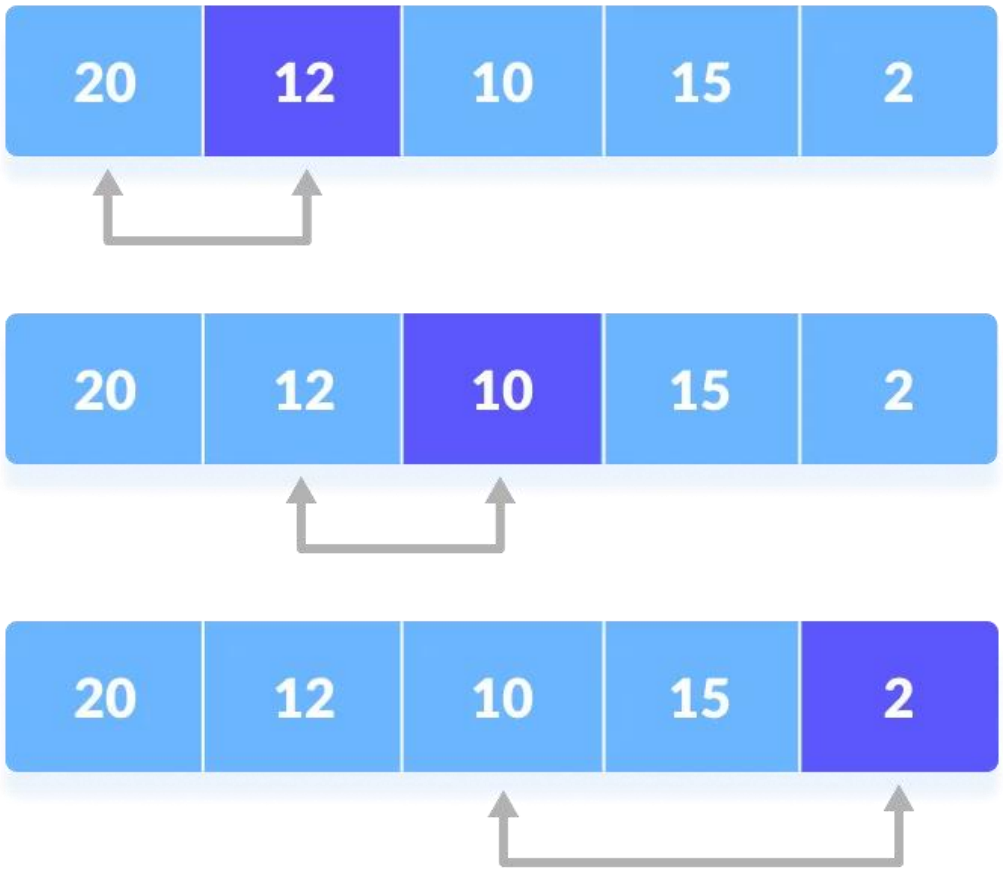
Algoritam radi ispravno za bilo koji dani prioriteni red, bez obzira kako je primijenjen. Međutim, brzina izvođenja određena je brzinama izvođenja operacija insert, min i removeMin. PriorityQueueSort više se smatra kao „shema“ sortiranja nego metoda sortiranja. Razne metode sortitanja baziraju se na ovoj „shemi“ sortiranja.

```
Algorithm PQ-Sort(S, C)
    Input list S, comparator C for the
    elements of S
    Output list S sorted in increasing
    order according to C
    P ← priority queue with
        comparator C
    while ¬S.isEmpty ()
        e ← S.remove(S.first ())
        P.insert (e, ∅)
    while ¬P.isEmpty()
        e ← P.removeMin().getKey()
        S.addLast(e)
```

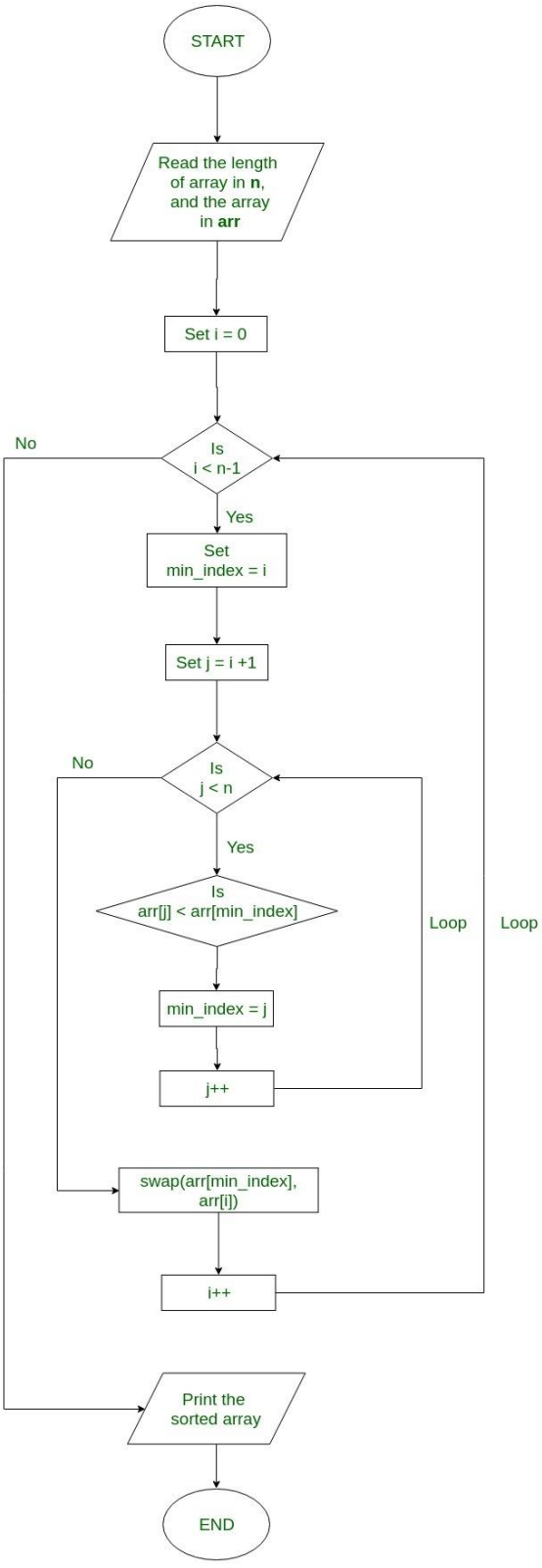
Slika 2.1: priorityQueueSort algoritam

2.4.1. Selektivno sortiranje

Ako implementiramo prioritetni red *P* s neuređenom listom elemenata, tada prva faza metode *PriorityQueueSort* traje *O*(n) vremena, jer svaki element možemo umetnuti u realnom vremenu. U drugoj fazi, vrijeme izvođenja svake od *min* i *removeMin* operacije je proporcionalan broju elemenata trenutno u *P*. Dakle, implementacija je bazirana na ponavljanom „selektiranju“ minimalnog elementa iz nesortirane liste u drugoj fazi. Iz ovog razloga, ovaj algoritam poznat je kao **selektivno sortitanje**. Kao što je gore navedeno, zahtijevnija je druga faza, gdje više puta uklanjamo element s najmanjim ključem. Veličina od *P* počinje s *n* i smanjuje se s svakim removeMin. Stoga, prva removeMin operacija zahtijeva *O*(n) vremena, druga *O*(n-1) itd. Dakle, druga faza zahtijeva *O*(n^2) vremena za izvođenje.



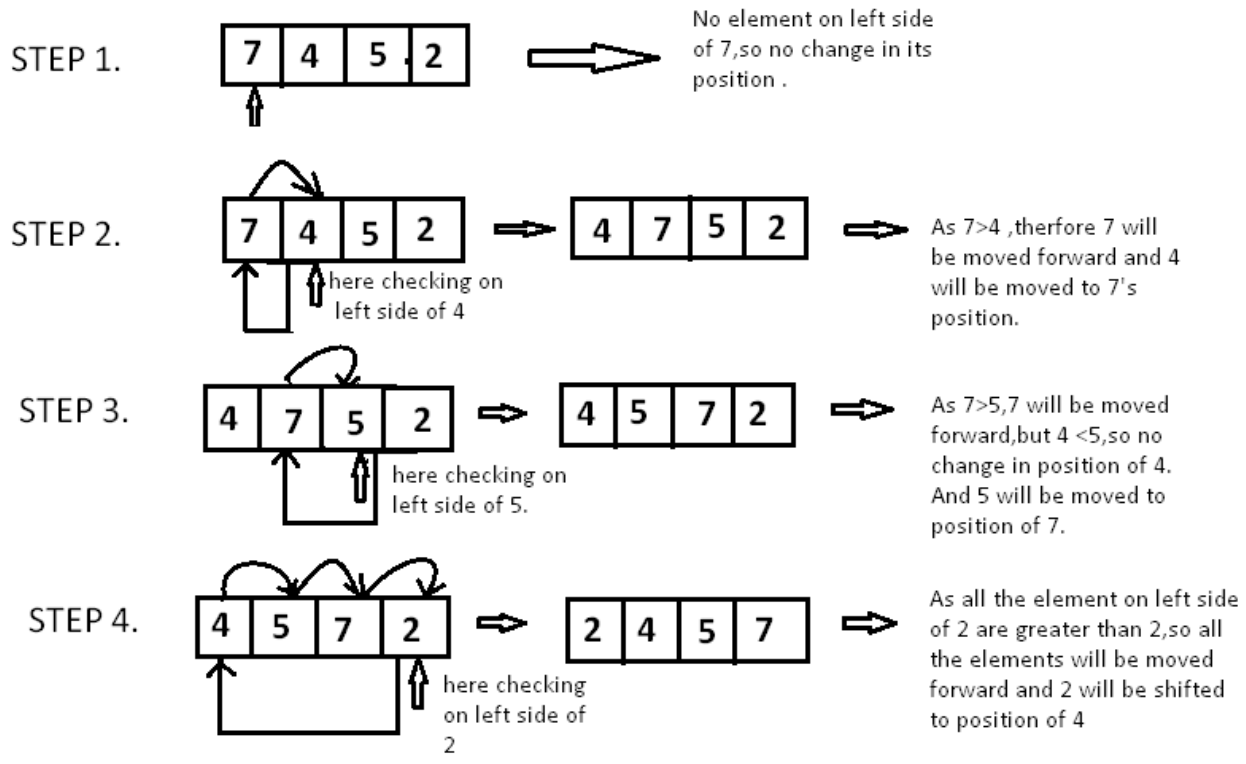
Slika 2.2: primjer selektivnog sortiranja



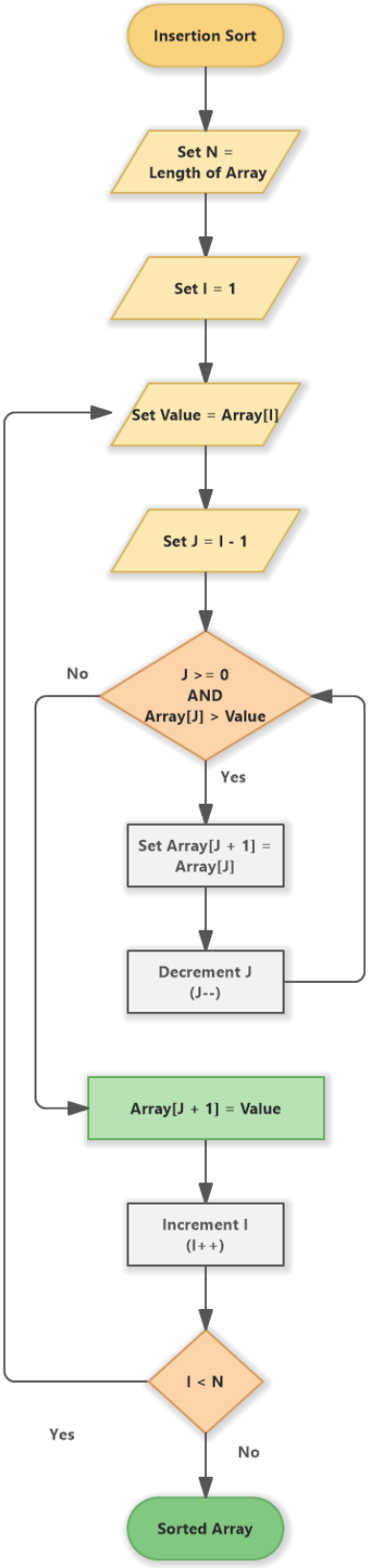
Slika 2.3: dijagram toka selektivnog sortiranja

2.4.2. Sortiranje metodom umetanja

Ako implementiramo prioriteni red *P* koristeći sortiranu listu, onda možemo poboljšati vrijeme izvođenja druge faze na *O*(n), jer svaka operacija *min* i *removeMin* na *P* sada zahtijeva *O*(1) vremena. Nažalost, prva faza sada postaje vremenski zahtijevnija, jer u najgorem slučaju svaka *insert* operacija zahtijeva vrijeme izvođenja proporcinalno veličini prioritetnog reda *P*. Zbog toga, ova metoda poznata je i kao **sortiranje metodom umetanja**. Prva faza izvodi se *O*(n) vremena, stoga i cijeli algoritam. Alternativno, mogli bismo promijeniti definiciju sortiranja umetanjem tako da umetnemo elemente koji počinju od kraja prioritetnog reda u prvoj fazi. U tom slučaju bi se sortiranje ovom metodom izvršilo na već sortiranoj listi u *O*(n) vremenu. Zapravo, vrijeme izvođenja metodom umetanja je u ovom slučaju *O*(n+I), gdje je *I* broj inverzija na ulaznoj listi, odnosno broj parova elemenata koji započinju u ulaznoj listi pogriješnim redoslijedom.



Slika 2.4: primjer sortiranja metodom umetanja



Slika 2.5: dijagram toka sortiranja metodom umetanja

2.5. Implementacija prioritetnog reda u C++

Na slici 2.6. prikazana je pomoćna funkcija koja služi za prikaz elemenata prioritetnog reda. Proslijeđuje joj se prioriteni red koji je sastoji od elemenata cjelobrojnog tipa. Dok ima elemenata u proslijeđenom prioritetnom redu ispisuje se svaki element, te briše nakon ispisa.

```
void showpq(priority_queue<int> gq)
{
    priority_queue<int> g = gq;
    while (!g.empty()) {
        cout << '\t' << g.top();
        g.pop();
    }
    cout << '\n';
}
```

Slika 2.6: pomoćna funkcija showpq

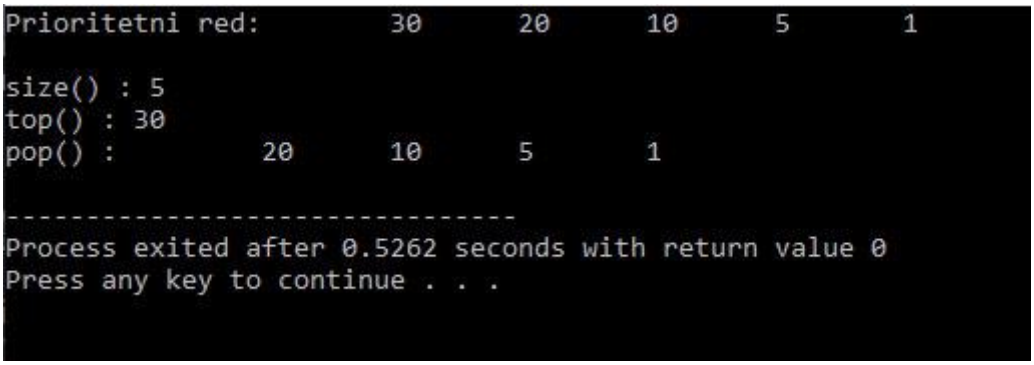
```
int main()
{
    priority_queue<int> prioritetni_red;
    prioritetni_red.push(10);
    prioritetni_red.push(30);
    prioritetni_red.push(20);
    prioritetni_red.push(5);
    prioritetni_red.push(1);

    cout << "Prioritetni red: ";
    showpq(prioritetni_red);

    cout << "\nsize() : " << prioritetni_red.size();
    cout << "\ntop() : " << prioritetni_red.top();

    cout << "\npop() : ";
    prioritetni_red.pop();
    showpq(prioritetni_red);

    return 0;
}
```

Slika 2.7: interakcija s prioritetnim redom

Na slici 2.7. prikazana je interakcija s prioritetnim redom unutar main funkcije. Prvo stvaramo prioritetni red *prioritetni_red* i dodajemo mu 5 proizvoljnih, nesortiranih cijelih brojeva. Nakon toga, proslijeđujemo stvoreni *prioritetni_red* funkciji *showpq* sa slike 2.7. koja ispisuje elemente proslijeđenoga prioritetnog reda. Nakon toga, funkcija *size* ispisuje veličinu, odnosno broj elemenata prioritetnog reda. Funkcija *top* ispisuje element na vrhu prioritetnog reda.

Nakon ispisa elementa, funkcijom *pop* briše se taj element.

Na slici 2.8. prikazan je cjelokupni ispis programa sa prethodne dvije slike. Kao što je vidljivo ispisuju se elementi počevši od najvećeg prema najmanjem. Razlog tome je što kod kreiranja prioritetnog reda C++ automatski stvara *max-heap* (detaljnije o gomilama u slijedećem poglavlju) što znači da je na vrhu najveći element.



Slika 2.8: ispis programa sa slike 2.6. i 2.7.

2.6. Prilagodljivi prioritetni redovi

Pretpostavimo da je *insert(e)* operacija prioritetnog reda uvećana tako da, nakon umetanja elementa *e* vraća referencu na novostvoreni unos, nazvan ***pozicija***. Ova pozicija je trajno povezana s unosom, tako da, čak iako se lokacija ulaza promijeni unutar prioritetnog reda, pozicija ostaje fiksirana s ulazom. Dakle, pozicije nam omogućavaju jedinstveno određivanje unosa na koji se svaka operacija primjenjuje.

Formalno definiramo prilagodljivi prioritetni red *P* takav da uz standardne operacije prioritetnog reda, podržava i dodatna poboljšanja:

(i) insert(e): Umeće element *e* u *P* i vraća poziciju koja se odnosi na nj
(ii) remove(p): Ukljanja ulaz referenciran s *P*
(iii) replace(p,e): Zamjenjuje ulaz refernciran s *p* s elementom *e* i vraća poziciju zamijenjenog elementa

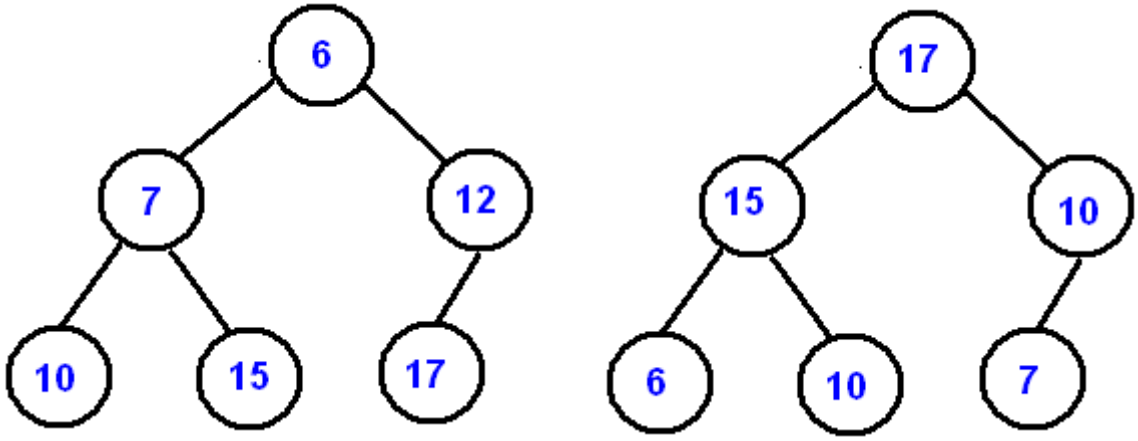
3. Gomila

Gomila je implementacija prioritetnog reda.

Gomila je skoro potpuno binarno stablo koje pohranjuje zbirku elemenata sa njihovim pridruženim ključevima.

Svojstvo uređenosti gomile: Za svaki čvor v, osim korijena, ključ pridružen v je manji ili jednak ključu roditelja v.

Svojstvo iznad navedeno je za max_heap gomilu gdje je ključ čvora veći ili jednak od ključeva svoje djece. Min_heap gomila ima obrnuto svojstvo gdje je ključ čvora veći ili jednak od ključeva svoje djece.



Slika 3.1: Min_heap(lijevo) i Max_heap(desno)

Radi učinkovitosti, želimo da gomila s kojom radimo ima što manju visinu. Ovime inzistiramo da gomila zadovoljava svojstvo potpunosti stabla.

Potpuno binarno stablo: stablo čijim se čvorovima mogu dati imena 0,1,…..N tako da za svaki i:

(i) lijevo dijete čvora i ima ime $2i + 1$ (osim ako je $2i + 1 > N$, onda ne postoji to dijete)

(ii) desno dijete čvora i ima ime $2i + 2$ (osim ako je $2i + 2 > N$, onda ne postoji to dijete)

3.1. Visina gomile

Neka *h* označava visinu stabla *T*. Drugi način definiranja posljednjeg čvora *T* je taj da je to čvor na razini *h* takav da su svi ostali čvorovi na toj razini lijevo od tog čvora.

Propozicija 3.1: Gomila *T* koja pohranjuje *n* unosa ima visinu $h = \lfloor \log n \rfloor$

Propozicija 3.1 ima jednu bitnu posljedicu. Podrazumijeva da ako možemo izvršiti operacije ažuriranja na gomilu u vremenu proporcionalnom njenoj visini, te operacije pokretat će se u logaritamskom vremenu.

3.2. Potpuno binarno stablo i njena reprezentacija

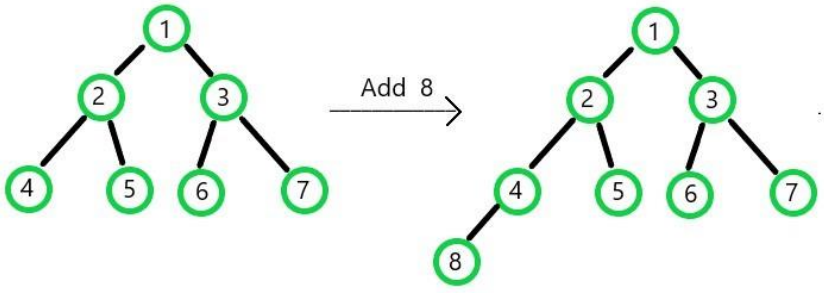
Kao apstraktni tip podataka (ADT) potpuno binarno stablo *T* podržava sve funkcije kao i binarno stablo ADT, plus dvije dodatne funkcije:

(i) add(*e*): dodaje element u *T* i vraća novi vanjski čvor *v* koji sadrži element *e*, tako da rezultirajuće stablo ostane potpuno binarno stablo sa zadnjim čvorom *v*
(i) remove(): uklanja zadnji čvor od *T* i vraća njegov element

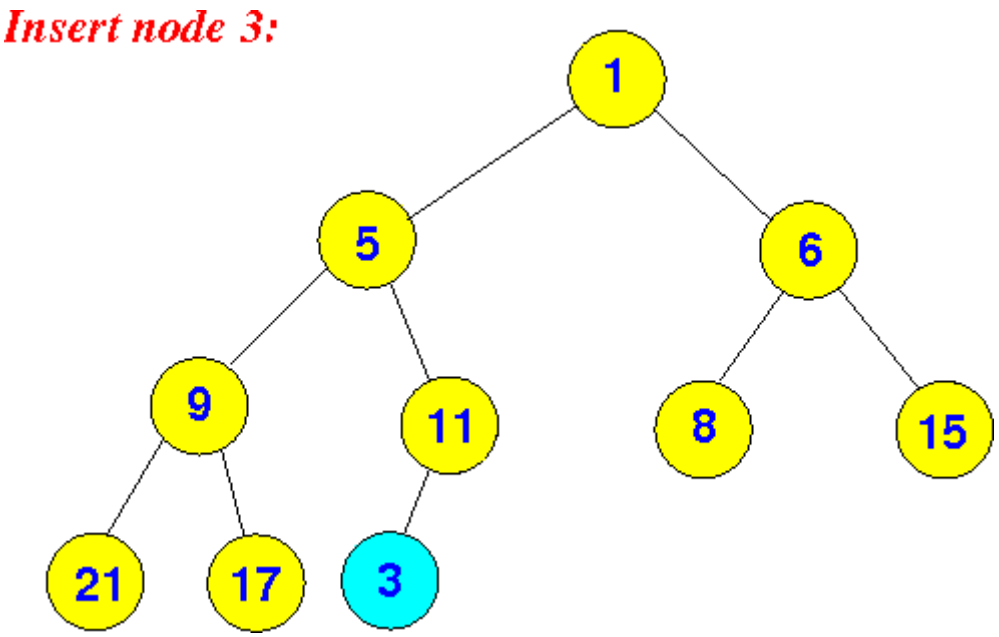
Koristeći se ovim operacijama ažuriranja, rezultirajuće stablo sa sigurnošću će biti potpuno binarno.

Imamo dva slučaja kod dodavanja čvora:

(i) ako je najniži level stabla *T* pun, onda *add()* umeće novi čvor kao lijevo dijete od najdaljnjeg lijevog čvora najnižeg levela, tada se visina stabla *T* povećava za 1
(ii) ako najniži level stabla *T* nije pun, onda *add()* umeće novi čvor desno od zadnjeg čvora u stablu *T*



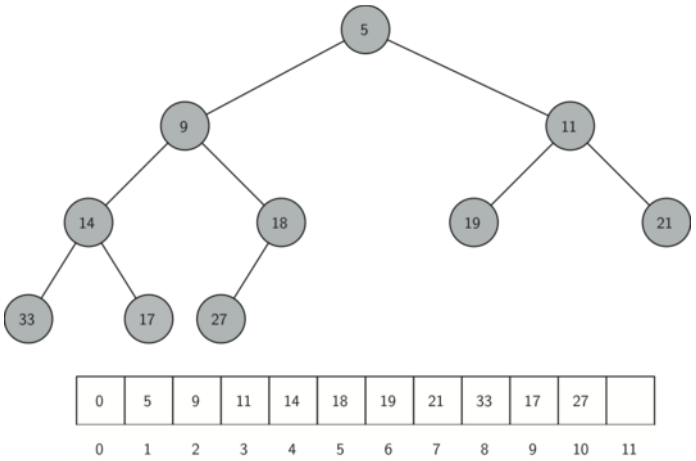
Slika 3.2: Dodavanje čvora na novu razinu



Slika 3.3: Dodavanje čvora na zadnju razinu

3.3. Vektorska reprezentacija potpuno binarnog stabla

Ovakvom implementacijom, čvorovi stabla *T* imaju indekse u rasponu [1,*n*] i zadnji čvor od *T* je uvijek na poziciji *n*, gdje je *n* broj čvorova od *T*.



Slika 3.4: Vektorska reprezentacija stabla

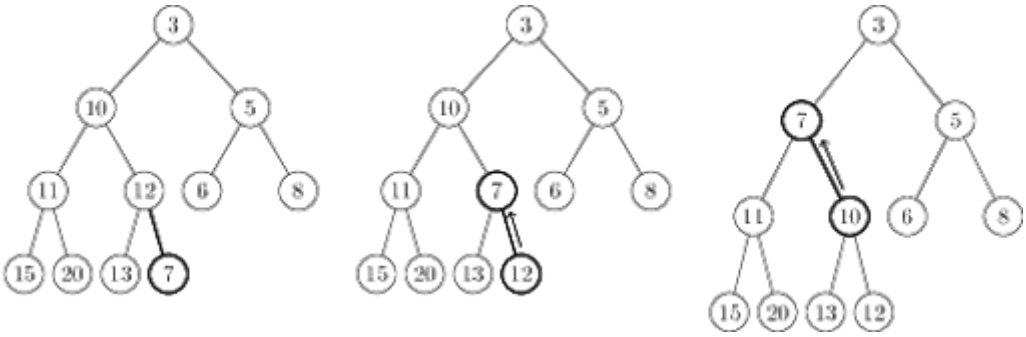
Ova implementacija korisna nam je kod funkcija dodavanja i brisanja. One mogu biti izvršene u *O*(1) vremenu jer je samo potrebno izbaciti ili dodadati element na kraj vektora.

3.4. Implementacija priritetnog reda u gomilu

Osim što se operacije dodavanja i brisanja izvršavaju u realnom vremenu, tako se i operacija izvlačenja najmanjeg elementa kod min-gomile ili najvećeg elementa kod max-gomile izvodi u $O(1)$ vremenu jer je samo potrebno izvući element s vrha gomile.

3.4.1. Umetanje

Prvo dodajemo novi čvor *z* u na kraj stabla *T* tako da taj čvor postaje zadnji čvor tog stabla. Nakon toga element *e* pohranjujemo u *z*. Nakon *add()* operacije moramo provjeriti je li narušeno svojstvo gomile tj. moramo provjeriti je li umetnuti element veći od svog roditelja (ako se radi o gomili kod koje je najveći element na vrhu, a ne najmanji). Ako je veći, onda moramo izvesti operaciju zamjene elemenata. Nakon zamjene, svojstvo opet može biti narušeno pa operaciju izmjene izvodimo sve dok se ne zadovolji svojstvo. Ovo kretanje prema gore u praksi se naziva ***up-heap bubbling***. U najgorem slučaju, novi ulaz kretati će se skroz do korijena stabla. U tom slučaju, broj zamjena čvorova bit će jednak visini stabla odnosno ⌊log n⌋.



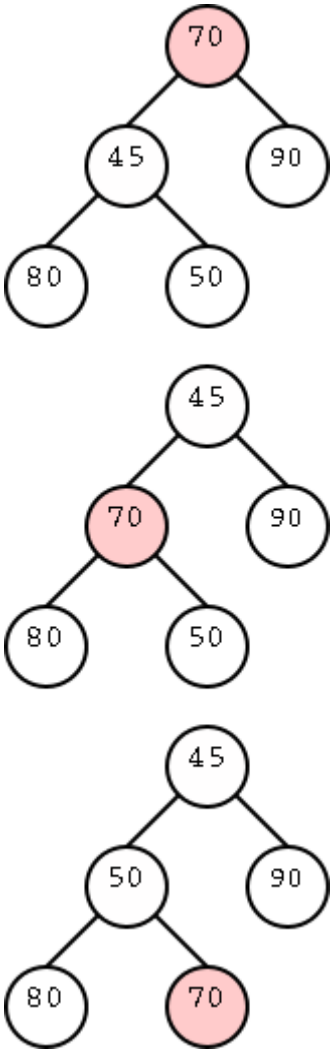
Slika 3.5: up-heap bubbling

3.4.2. Brisanje

Znamo da se najveći element nalazi u korijenu stabla, ali ne možemo samo tako obrisati korijen stabla. Umjesto toga zamijenimo zadnji i prvi čvor u stablu te ispišemo novi zadnji element u stablu odnosno, sada najveći element.

3.4.3. Down-Heap Bubbling poslije brisanja najvećeg/najmanjeg elementa

Nakon što se izvršila operacija brisanja korijenskog čvora odnosno zamjene prvog i zadnjeg čvora, treba provjeriti zadovoljava li stablo svojstva gomile. Ako stablo ima samo jedan čvor onda je svojstvo trivijalno zadovoljeno. Ako čvor kojeg smo smjestili na vrh ima djecu i ako je lijevo ili desno dijete veće od tog čvora, mora se izvršiti zamjena čvorova. Ova zamjena izvršava se sve dok se ne zadovolji svojstvo gomile. Ova operacija u praksi se naziva ***down-heap bubbling***. Kao i kod dodavanja čvora, ova operacija u najgorem slučaju izvesti će se ⌊log n⌋ puta.



Slika 3.6: down-heap bubbling

3.4.4. Usporedba brzina izvođenja

Operacija	Vrijeme
size, empty	*O* (1)
min	*O* (1)
insert	*O* (log n)
removeMin	*O* (log n)

Svaka od funkcija prioritetnog reda može se izvesti u *O* (1) ili *O* (log n) vremenu, gdje je *n* broj elemenata u trenutku kada je funkcija pokrenuta. Slijedi:

(i) Gomila *T* ima *n* čvorova, svaki sadržavajući element
(ii) Operacije *add()* i *remove()* na *T* zahtijevaju *O* (1) u najboljem i *O* (log n) u najgorem slučaju
(iii) U najgorem slučaju, *up-heap bubbling* i *down-heap bubbling* zahtijevaju broj operacija koji je jednak visini stabla.
(iv) Visina gomile *T* je *O* (log n), jer je *T* potpuno stablo

3.5. Implementacija u C++

```
class MaxHeap{
        private:

            int size;
            vector<int> vect;

            int p(int i){return i>>1;}; // i/2
            int l(int i) {return i<<1; }; // i*2
            int r(int i) {return (i<<1) + 1;}; // i*2 + 1

        public:
            bool isEmpty() const {return size == 0;};
            int getMax() const {return vect[1];};
            void insertItem(int val);
            void shiftUp(int i);
            int extractMax();
            void shiftDown(int i);

    };
```

Slika 3.7: Definicija klase

Na slici 3.7. prikazana je definicija klase. Atributi klase su čvor *p*,*l* i *r* koji predstavljaju čvorove roditelja te lijevog i desnog dijeteta. Atribut *size* predstavlja veličinu vektora *vect* koji se sastoji od elemenata tipa *int*. Osim vektorske reprezentacije prioritetnog reda kojeg

koristimo za izvedbu gomile, također se možemo koristiti i listama. U javnom dijelu klase navedene su funkcije za rad s gomilom. U nastavku su prikazane i objašnjene.

```
void MaxHeap::shiftUp(int i){
    if(i>size) return;
    if(i == 1) return;
    if(vect[i] > vect[p(i)]){
        swap(vect[p(i)] , vect[i]);
    }
    shiftUp(p(i));
}
```

Slika 3.8: shiftUp funkcija

Na slici 3.8. prikazana je *shiftUp()* funkcija koja vraća svojstvo gomile ukoliko je ono narušeno tj. u ovom slučaju ako je element čvora koji je dijete veći od elementa čvora koji je roditelj tog dijeteta. Prvo se provjerava je li čvor kojeg premještamo prema gore veći od veličine gomile, te ukoliko je to istina radi se o pogrešci te se izlazi iz funkcije. Ako je broj čvorova 1 tada nema potrebe izvoditi ovu funkciju te se također zaustavlja izvođenje. Ako je element dijeteta, u ovom slučaju element vektora na poziciji *vect[i]* veći od elementa roditelja, u ovom slučaju element vektora na poziciji *vect[p(i)]*, tada je potrebno izvesti zamjenu elemenata kako bi se osiguralo svojstvo gomile. To ne osigurava potpuno zadovoljavanje svojstva gomile kroz cijelo stablo pa rekurzivno provjeravamo elemente prema gore.

Na slici 3.9. prikazana je *shiftDown()* funkcija. Kod brisanja najvećeg elementa gomile prvo se moraju zamijeniti prvi (korijen) čvor i zadnji čvor te nakon toga izbrisati zadnji čvor (sada element s najvećom vrijednošću). Nakon što se izvela operacija zamjene može biti narušeno svojstvo gomile. Unutar funkcije *shiftDown()* prvo se provjerava je li čvor kojeg premještamo prema dolje na lokaciji koja je veća od veličine gomile tj. je li gomila prazna. Ako je ovaj slučaj istinit prekida se izvođenje funkcije. Također, ako se radi o gomili s jednim čvorom, prekida se izvođenje funkcije. Nakon toga provjeravamo je li lijevo dijete na poziciji koja je manja ili jednaka veličini gomile i ujedno, je li čvor kojeg promatramo manji od lijevog dijeteta. Ako je ovaj slučaj istinit, izvodi se zamjena čvorova.

Isti postupak ponavljamo i za desno dijete.

```
void MaxHeap::shiftDown(int i){
    if(i> size) return;

    int swapId = i;

    if(l(i) <= size && vect[i] < vect[l(i)]){
        swapId = l(i);
    }

    if(r(i) <= size && vect[swapId] < vect[r(i)]){
        swapId = r(i);
    }

    if (swapId != i ){
        swap(vect[i] , vect[swapId]);
        shiftDown(swapId);
    }
    return;

}
```

Slika 3.9: shiftDown funkcija

Na slici 3.10. prikazana je *extractMax()* funkcija koja vraća najveći ili najmanji element, ovisno o kojoj vrsti gomile se radi (u ovom slučaju najveći element). Unutar funkcije koristi se prethodno navedena funkcija *shiftDown(),* koja provjerava je li narušeno svojstvo gomile.

```
int MaxHeap::extractMax(){
    int maxNum = vect[1];
    swap(vect[1] , vect[size--]);
    shiftDown(1);
    return maxNum;
}
```

Slika 3.10: extractMax funkcija

Na slici 3.11. prikazana je *insertItem()* funkcija koja umeće element na kraj gomile, te nakon toga koristi *shiftUp()* funkciju kako bi se provjerilo je li narušeno svojstvo gomile.

```
void MaxHeap::insertItem(int val){
    if(size + 1> vect.size()) {
        vect.push_back(0);
    }
    vect[++size] = val;
    shiftUp(size);
    return;
}
```

Slika 3.11: insertItem funkcija

3.6. Heap sort

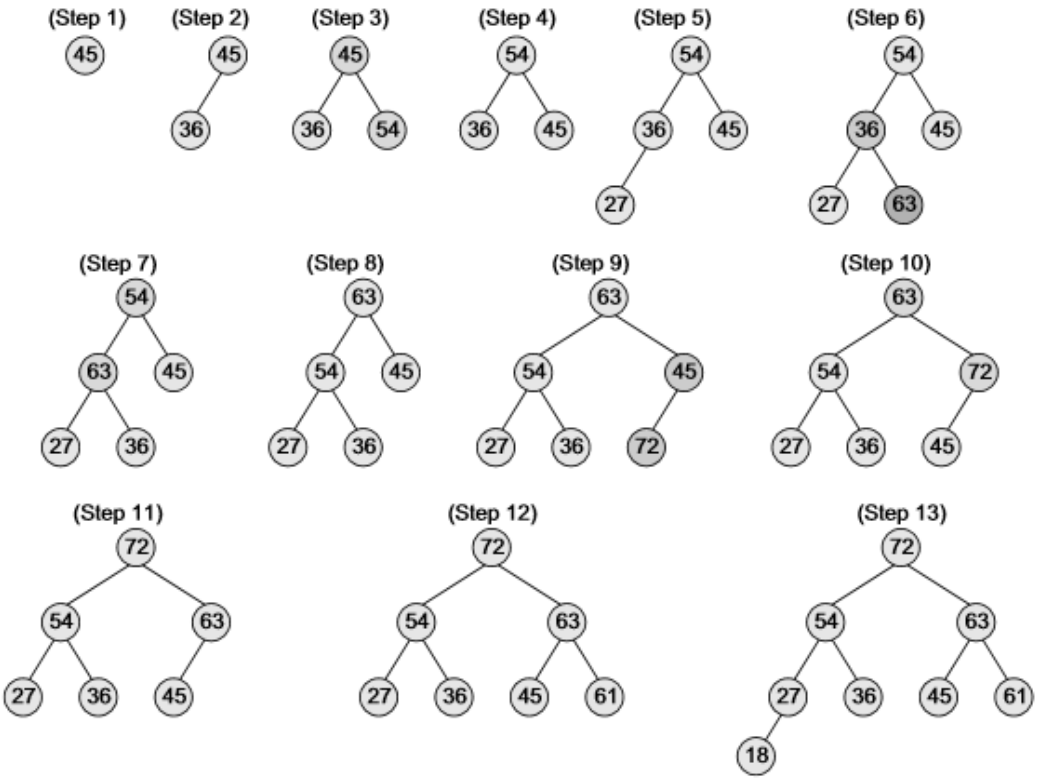
Kao što je navedeno u prethodnim poglavljima, realizacija prioritetnog reda s gomilom dovodi do prednosti izvođenja svih funkcija u logaritamskom vremenu ili bolje. Upravo zbog toga, sortitanje gomilom vrlo je efikasan algoritam.

Propozicija 3.1: Algoritam sortiranja gomilom sortira listu *L* od *n* elemenata u *O* (n logn) vremenu, pretpostavljajući da se dva elementa od *L* mogu usporediti u *O* (1) vremenu.

Koraci koji se izvode tokom sortiranja gomilom:

(i) Izrada gomile iz nesortiranog niza
(ii) Pronalazak najvećeg elementa
(iii) Zamjena korijenskog čvora sa zadnjim čvorom – sada je čvor s najvećim elementom na kraju niza
(iv) Izbacivanje čvora *n* iz niza i smanjivanje veličine gomile za 1
(v) Provjera zadovoljavanja svojstva gomile
(vi) Ponavljanje (iii) – (iv) dok je veličina gomile veća od 1

Zašto predstavljanje gomile nizom? Budući da je gomila potpuno binarno stablo, može se lako predstaviti kao niz što je prostorno efikasno. Ako je roditeljski čvor pohranjen u indeksu I, lijevo dijete može se izračunati s 2*I+1 a desno dijete s 2*I+2 (pod pretpostavkom da indeksiranje počinje s 0)



Slika 3.12: heap sort

3.6.1. In-place heap sort

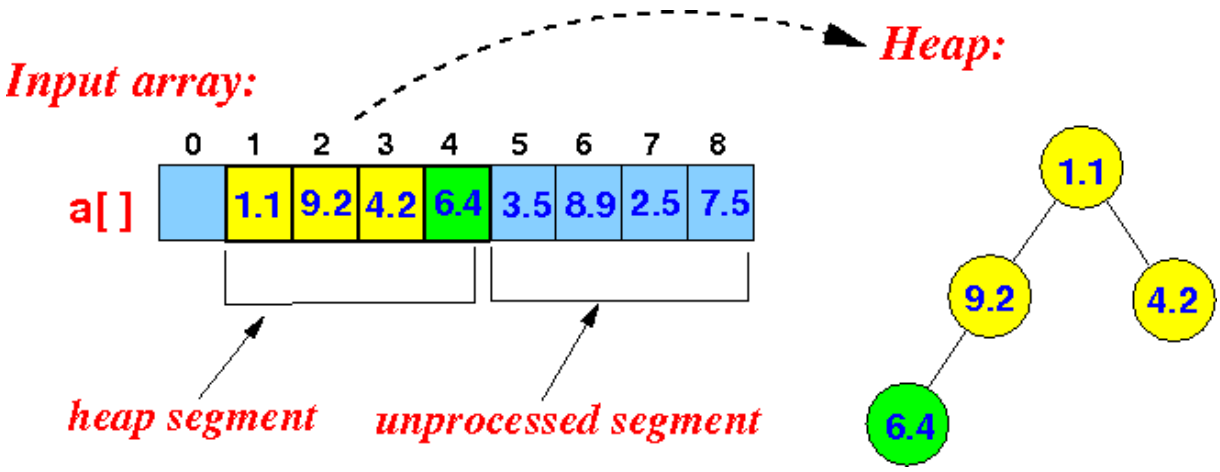
Umjesto izgradnje posebne strukture podataka, možemo preraspodijeliti elemente već postojećeg vektora. Općenito, kažemo da je **in-place** sort vrsta sortitanja uporabom gomile ako koristi istu količinu memorije uz memoriju potrebnu za elemente sortirajući se samostalno.

Prednosti in-place heap sort algoritma su:

(i) Minimalno iskorištavanje memorijskog prostora
(ii) Prilika za **optikizaciju finim podešavanjem**
(iii) Poboljšano keširanje

Primjeri keširanja:

(i) Memorijska hijerarhija
(ii) Brzina memorije
(iii) Caching near by



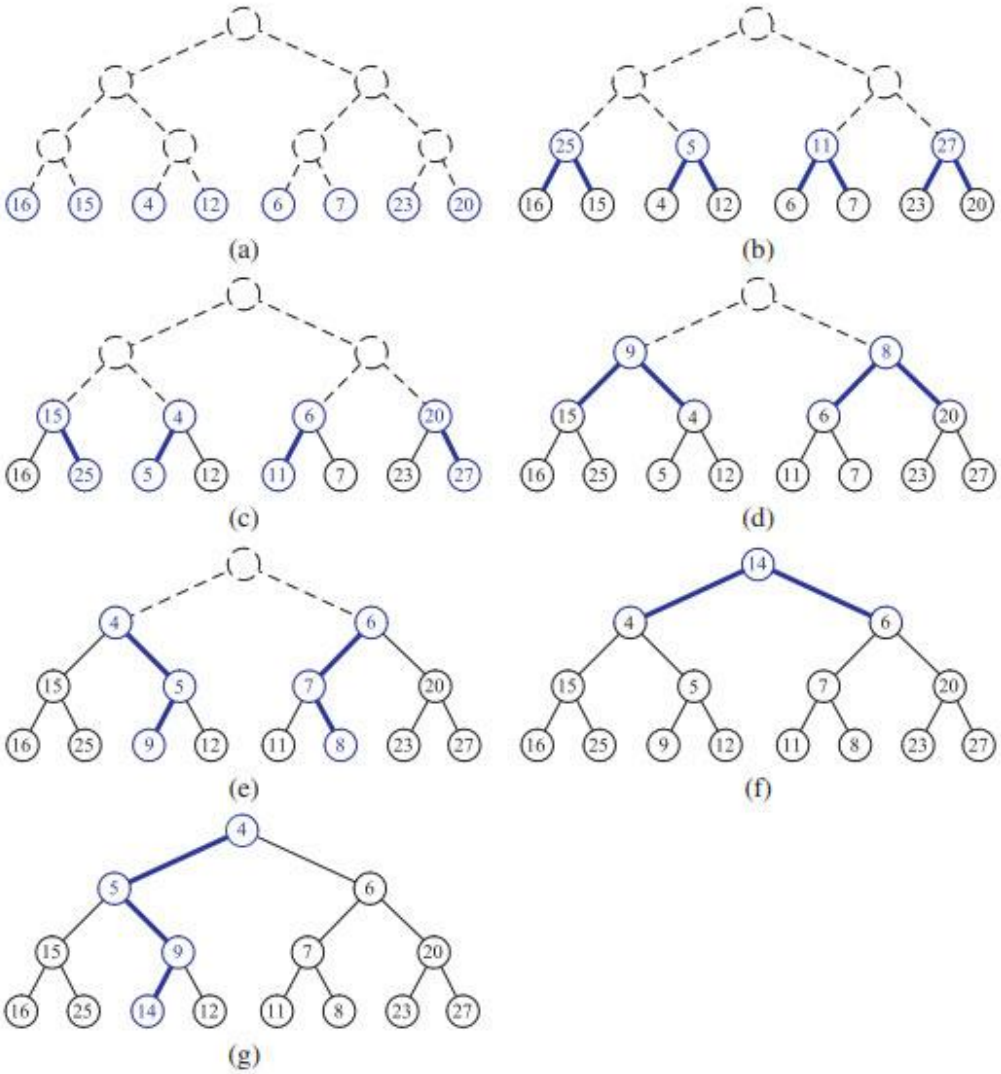
Slika 3.13: in-place heap sort

3.7. Bottom-up konstrukcija gomile

Mogu se napraviti neka dodatna poboljšanja. Možemo poboljšati *O*(n logn) vrijeme izvođenja za izgradnju gomile, ako unaprijed znamo veličinu. Pretpostavimo da je veličina n = 2h – 1 gdje je n neparan broj. Visina gomile je h = lg(n+1).

Osnovna ideja je izgradnja od dna prema vrhu. Iako je postupak prirodno rekurzivan, prvo je objašnjen iterativno:

1. Napravimo (n+1)/2 stabala s jednim elementom ((n-1)/2 stabala u rezervi).
2. U drugom koraku konstruiramo (n+1)/4 gomile, svaka s tri čvora, spajanjem parova gomila s jednim elementom i dodavajući jedan dodatni element. Novi ulaz smješten je u korijenskom čvoru i ovisno o načelu očuvanja svojstva gomile, potrebno ga je zamijeniti s dijetetom.
3. U trećem koraku konstruiramo (n+1)/8 gomila svaka pohranjujući 7 elemenata, spajajući gomile s 3 elemenata i dodavajući jedan dodatni element koji će biti smješten u korijenskom čvoru. Ovisno o načelu očuvanja svojstva gomile, potrebno ga je zamijeniti s dijetetom, odnosno izvesti *down-heap bubbling*
4. Ponavljati ovu operaciju lg(n+1) puta, koliko je i visina stabla



Slika 3.14: bottom-up kontrukcija gomile

3.7.1. Rekurzivni algoritam izgradnje gomile bottom-up metodom

O(n) vrijeme izvođenja nije odmah vidljivo na algoritmu s slike 3.15. Najgori slučaj za svaki rekurzivni poziv algoritma je visina trenutnog stabla, *T*. Ukupni trošak utvrdit ćemo razgraničenjem troškova svih *bubble-down* operacija. Put do nasljednog čvora (prvo dolje desno pa dolje lijevo) povezujemo s korijenskim čvorom. Ukupno vrijeme, svih rekurzivnih poziva, ukupan je broju pridruženih staza, koji iznosi *O*(n).

Recursive Algorithm for **BottomUpHeap(S)**

input: Sequence S with $n = 2^h - 1$ keys
Output: Heap T storing the keys of S.

```
if S empty then return empty heap
Remove the first key, k, from S.
// Note k will become root of new Tree
Split S into S1 and S2 each of size (n-1)/2
 // recursive calls
T1 = BottomUpHeap(S1)
T2 = BottomUpHeap(S2)
Create binary tree T with
        root storing k,
        T1 left subtree and T2 right subtree
Perform Bubble down from root of T
return T
```

Slika 3.15: rekurzivni algoritam za bottom-up metodu izgradnje gomile

4. Zaključak

Korištenje prioritetnih redova i gomile u informatici i programiranju je neizbježno i pojednostavljuje implementaciju nekih dijelova aplikacija.

Ne treba miješati red s prioritetnim redom, jer nam prioritetni redovi omogućuju dodijeljivanje vrijednosti svakom elementu što otvara mnoge mogućnosti.

Gomile se koriste kod Dijkstrinog algoritma za pronalaženje najkraćeg puta, kao i kod drugih mnogih korisnih algoritama. Istražujući gomile i prioritetne redove, može se zaključiti da se ovi algoritmi u biti koriste kada želimo učinkovito pristupiti najmanjem ili najvećem elementu strukture (ovisno radi li se o max-heap ili min-heap gomili).

Popis slika

5. Literatura

5.1. Knjige

1. Goodrich, M.T., Tamassia, R. and Mount, D.M., 2011. Data structures and algorithms in C++. John Wiley & Sons

5.2. Web stranice

1. https://web.math.pmf.unizg.hr/~stojic/SPAbinarytree.pdf
2. https://www.geeksforgeeks.org/heap-sort/
3. http://www.csl.mtu.edu/cs2321/www/newLectures/09_Inplace_Heap_Sort.html

5.3. Youtube

1. https://www.youtube.com/watch?v=B7hVxCmfPtM
2. https://www.youtube.com/watch?v=6jwj2WIRvTE
3. https://www.youtube.com/watch?v=JSqznrzWGvc&t=271s

5.4. Slike

1. https://www.andrew.cmu.edu/course/15121/lectures/Binary%20Heaps/pix/heap.bmp
2. https://media.geeksforgeeks.org/wp-content/uploads/20200520135909/Adding-a-node-to-a-heap2.jpg
3. http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/9-BinTree/FIGS/Heap03a.gif
4. https://runestone.academy/runestone/books/published/cppds/_images/heapOrder.png
5. https://lh3.googleusercontent.com/proxy/eLBkwp-Du6eZue0fav_j-67WS8DJsqwceYjgjXCBFIxtZxwIXIUUY1F9VHJX53NskoI7RtdYFWbbhc0Ft-l5Vp03fKOyB_Y
6. https://cs.nyu.edu/~gottlieb/courses/2000s/2002-03-fall/alg/lectures/figs/down-heap.png
7. https://i.pinimg.com/originals/76/fa/7f/76fa7f2b3dcf4a05933c345056a30279.png
8. http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/9-BinTree/FIGS/HeapSort06.gif

9. https://slideplayer.com/slide/12539827/75/images/2/Priority+Queue+Sorting.jpg
10. https://cdn.programiz.com/cdn/farfuture/9jjqXX0fGtJE2ul2Mga20fvf_GkNlFAFsDMwrrwFzbQ/mtime:1582112622/sites/tutorial2program/files/Selection-sort-0-comparision.png
11. https://media.geeksforgeeks.org/wp-content/cdn-uploads/Selection-sort-flowchart.jpg
12. https://he-s3.s3.amazonaws.com/media/uploads/46bfac9.png
13. https://www.softwareideas.net/i/DirectImage/1765/Insertion-Sort--Flowchart-
14. https://www.geeksforgeeks.org/priority-queue-in-cpp-stl/