ADT prioritetna lista

Strukture podataka i algoritmi 2

Prioritetna lista

- Prilikom sekvencijalne (paketne) obrade podataka često je zgodno uvesti prioritete tako da podaci sa većim prioritetom budu procesirani pre podakata sa manjim prioritetom

- Prioritetna lista (engl. *priority queue*) je kolekcija objekata u kojoj svaki element ima prioritet obrade

- Element sa najvećim prioritetom se procesira (obrađuje) prvi. Pošto je element obrađen biva izbačen iz liste

- **Elementi prioritetne liste ne moraju biti sortirani po prioritetu, bitno nam je da u svakom trenutku znamo koji element ima najveći prioritet obrade**

Prioritetna lista

- Prioriteti elemenata u prioritetnoj listi mogu biti
  - **Eksplicitni** – svakom elementu u prioritetnj listi pridružujemo ceo ili realan broj kojim je određen prioritet obrade
  - **Implicitni** – prioritet elementa određen
    - **njihovim prirodnim uređenjem**
      - objekti klase koja implementira Comparable interfejs
      - dva tipa prioritetnih listi
        - **MAX-PQ:** veći element veći prioritet
        - MIN-PQ: manji element veći prioritet
    - proizvoljnim komparatorom

ADT prioritetna lista

- Tip podataka definiše skup mogućih vrednosti i skup operacija nad varijablama tog tipa
- ADT: način realizacije tipa sakriven od korisnika tipa
  - Korisnik ADT-a ne mora da zna detalje implementacije operacija i kojom strukturom podataka se predstavljaju promenljive tipa kako bi koristio tip
- ADT prioritetna lista
  - Vrednost tipa – kolekcija uporedivih elemenata (može biti struktuirana na više fundamentalno različitih načina)
  - Operacije
    - Dodavanje novog elementa u prioritetnu listu
    - Dobavljanje elementa sa najvećim prioritetom obrade
    - Brisanje elementa sa najvećim prioritetom iz prioritetne liste
    - Provera da li je prioritetna lista prazna
    - Veličina prioritetne liste

Specifikacija ADT-a

- Specifikacija ADT-a je apstraktni deo ADT-a
  - **zaglavlja public** metoda kojima definišemo operacije ADT-a
  - Ono što korisnik ADT-a treba da zna kako bi koristio ADT
  - **Različite realizacije ADT-a dele istu specifikaciju ADT-a**

- U PJ Java ADT-ove možemo specificirati **Java interfejsima**
  - Interfejs se sastoji od definicija konstanti i zaglavlja metoda
  - Java klase implementiraju Java interfejse
  - Ako **ne-apstraktna** klasa *A* implementira interfejs *I* tada *A* mora definisati sva zaglavlja data u *I*
  - Korisniku je dovoljno da zna interfejs da bi koristio klasu
  - Interfejsi će detaljnije biti pokriveni kursom OOP1

Specifikacija ADT prioritetna lista

```
public interface PriorityQueue<T extends Comparable<T>> {

        /** Dodaje novi element u prioritetnu listu */
        void insert(T element);

        /** Vraca element sa najvecim prioritetom */
        T max();

        /** Vraca i brise iz prioritetne liste element sa najvecim prioritetom*/
        T delMax();

        /** Da li je prioritetna lista prazna? */
        boolean isEmpty();

        /** Vraca velicinu prioritetne liste */
        int size();
}
```

PriorityQueue<T extends Comparable<T>>

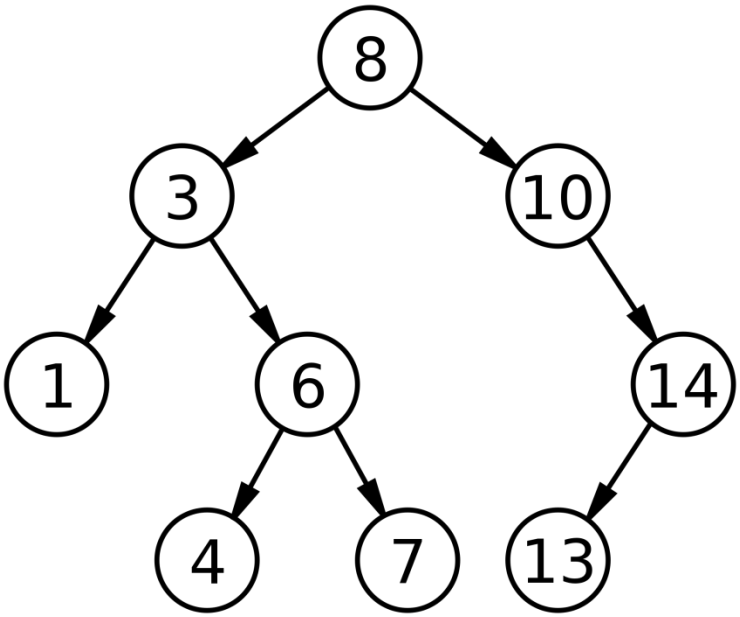
1. Klasa PriorityQueue je parametrizovana tipom T
2. Klasa T mora implementirati interfejs Comparable<T>

Reprezentacija prioritetne liste

- Prioritetnu listu ćemo reprezentovati nizom koji ima hip (engl. *heap*) strukturu/osobinu.
- Niz sa hip osobinom obebezbeđuje efikasne insert/delete operacije vremenske složenosti O(log n).
- Niz sa hip osobinom možemo vizuelno predstaviti u obliku binarnog stabla
- Binarno stablo je struktura koja se sastoji od čvorova, pri čemu svaki čvor ima pokazivač na levo i desno pod-stablo.

Terminologija binarnih stabala

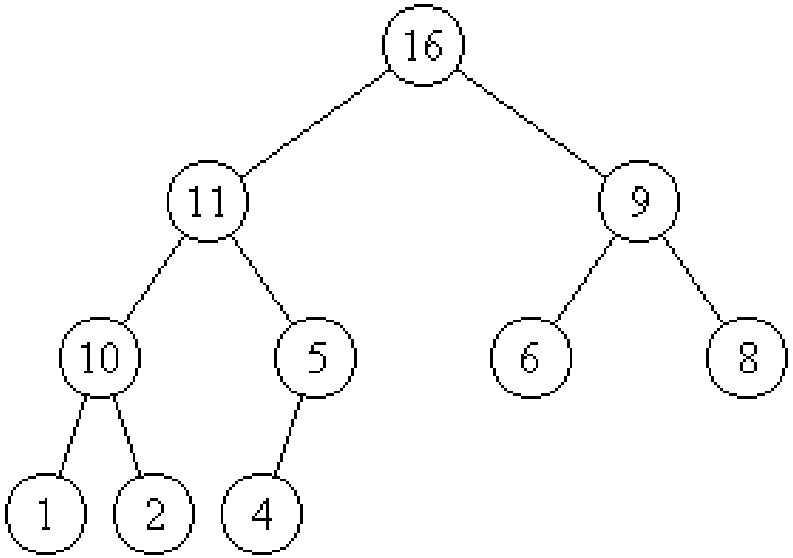
- Svaki čvor može da ima najviše dva sina – levog i desnog.
- Koren (korenski čvor) nije sin ni jednom čvoru.
- Listovi stabla su čvorovi koji nemaju sinove.
- Roditelj (otac) čvora B je čvor A ako je B sin A.



- Koren – 8
- Listovi – 4, 7, 13
- 1 – levi sin od 3
- 6 – desni sin od 3
- 3 – roditelj od 1 i 6
- 14 nema desnog sina, dok 10 nema levog sina

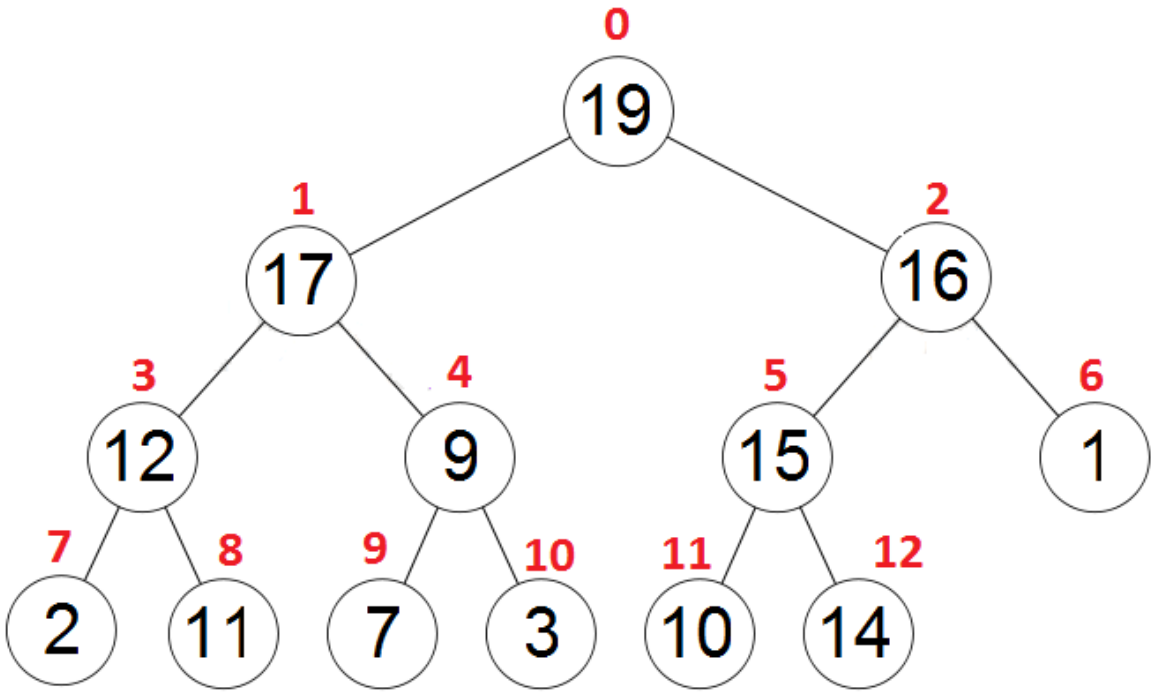
Binarno stablo sa hip osobinom

- Binarno stablo za hip osobinom zadovoljava sledeće uslove
  - (1) Stablo je kompletno
    - Svi nivoi stabla osim eventualno poslednjeg su potpuno popunjeni
    - Poslednji nivo je ako nije potpun je popunjen redom sa leva na desno
  - (2) Za svaki čvor važi da je veći od svojih sinova
    → čvor je veći od svih čvorova u levom i desnom podstablu
    → koren je maksimum



Reprezentacija kompletnih binarnih stabala

- Kompletna binarna stabla se jednostavno predstavljaju nizom
- Sinovi čvora sa indeksom *p* imaju indekse: 2*p* + 1 i 2*p*+ 2
- Roditelj čvora sa indeksom *s* ima indeks (s – 1)/2 (celobrojno deljenje)



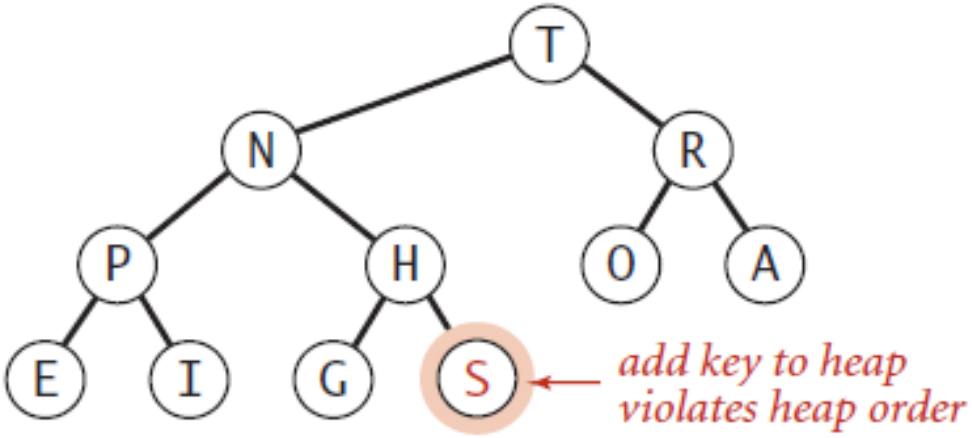
19	17	16	12	9	15	1	2	11	7	3	10	14
0	1	2	3	4	5	6	7	8	9	10	11	12

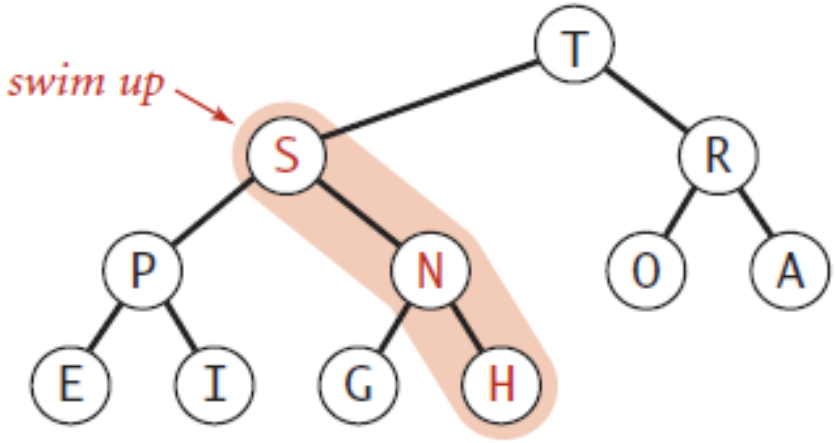
HeapPQ.java

```
import java.util.ArrayList;
import java.util.NoSuchElementException;

public class HeapPQ<T extends Comparable<T>> implements PriorityQueue<T> {
    private static final int DEFAULT_INITIAL_CAPACITY = 100;

    private ArrayList<T> queue;

    public HeapPQ(int initialCapacity) {
        if (initialCapacity <= 0)
            throw new IllegalArgumentException("initial capacity <= 0 ?!");

         queue = new ArrayList<>(initialCapacity);
    }

    public HeapPQ() {
        this(DEFAULT_INITIAL_CAPACITY);
    }

    //... to be continued
}
```

insert

1. Dodaj novi element na kraj (proširivog) niza
2. Popravi niz tako da ponovo ima hip strukturu
   I. Ako je novi element veći od oca razmenih ih
   II. Ponavljaj prethodni korak dokle god je zadovoljen uslov razmene ili dok ne stigneš do korenskog čvora

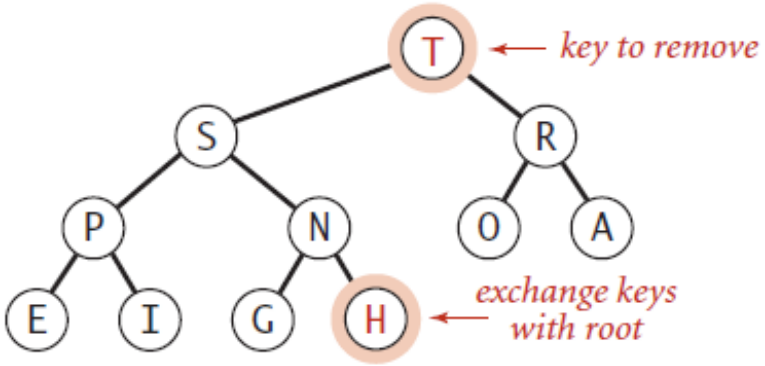


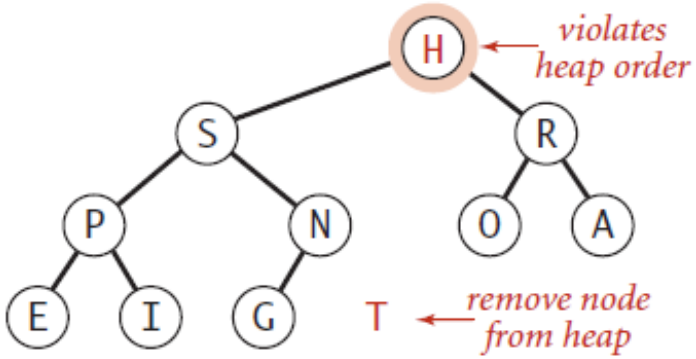


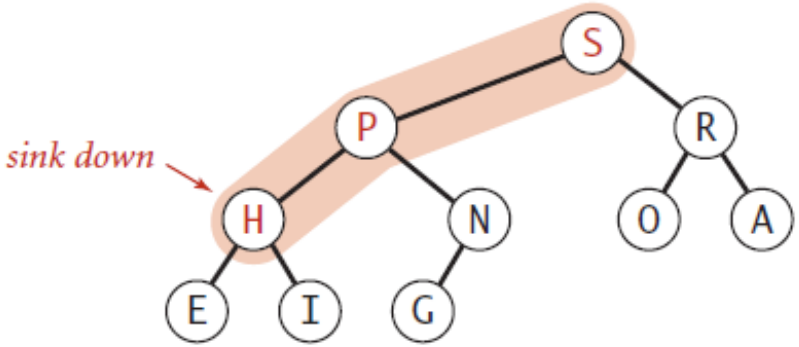
```java
public void insert(T element) {
	queue.add(element);
	restoreHeapProperty(queue.size() - 1);
}

private void restoreHeapProperty(int sonIndex) {
	boolean heapRestored = false;
	int parentIndex = (sonIndex - 1) / 2;
	while (!heapRestored && sonIndex > 0) {
		T parent = queue.get(parentIndex);
		T son = queue.get(sonIndex);

		// da li je otac veci od sina?
		if (parent.compareTo(son) > 0) {
			heapRestored = true;
		} else {
			// razmeni oca sa sinom
			swap(sonIndex, parentIndex);

			// nastavi dalje ka korenu: sin je sada na
			// poziciji oca i ima novog oca
			sonIndex = parentIndex;
			parentIndex = (sonIndex - 1) / 2;
		}
	}
}
```

delMax

1. Poslednji element niza stavi na početak niza
2. Povrati hip strukturu, ovoga puta od korena ka kraju niza
   I. Ako je otac manji od većeg sina razmeni ih
   II. Ponavljaj prethodni korak dokle god otac ima bar jednog sina i dokle god je uslov razmene zadovoljen







delMax

```
public T delMax() {
   if (queue.size() == 0)
      throw new NoSuchElementException("empty queue");

   T res = queue.get(0);
   swap(0, queue.size() - 1);
   queue.remove(queue.size() - 1);

   restoreHeapProperty();

   return res;
}
```

Povratak hip osobine

```
private void restoreHeapProperty() {
	boolean heapRestored = false;
	int parentIndex = 0;

	while (!heapRestored) {
		int maxSonIndex = getMaxSon(parentIndex);
		// ne postoji ni jedan od sinova
		if (maxSonIndex == -1) {
			heapRestored = true;
		} else {
			T parent = queue.get(parentIndex);
			T maxSon = queue.get(maxSonIndex);
			// da li je otac veci od veceg sina?
			if (parent.compareTo(maxSon) > 0)
				heapRestored = true;
			else {
				// razmeni oca sa sinom
				swap(parentIndex, maxSonIndex);

				// otac je sada na poziciji sina (i ima nove sinove)
				parentIndex = maxSonIndex;
			}
		}
	}
}
```

```java
/** Odredjuje indeks veceg sina za datog roditelja
 *  Vraca -1 ukoliko ne postoji ni jedan od sinova
 */
private int getMaxSon(int parentIndex) {
	int son1Index = 2 * parentIndex + 1;
	int son2Index = 2 * parentIndex + 2;
	int maxSonIndex = -1;

	// postoji sin1?
	if (son1Index < queue.size()) {
		maxSonIndex = son1Index;
	}

	// postoji sin2?
	if (son2Index < queue.size()) {
		// da li drugi sin ima veci prioritet
		T s1 = queue.get(son1Index);
		T s2 = queue.get(son2Index);
		if (s2.compareTo(s1) > 0)
			maxSonIndex = son2Index;
	}

	return maxSonIndex;
}
```

Ostatak metoda je trivijalan...

```
public T max() {
	if (queue.size() == 0)
		throw new NoSuchElementException("empty queue");

	return queue.get(0);
}

public boolean isEmpty() {
	return queue.size() == 0;
}

public int size() {
	return queue.size();
}

private void swap(int indexa, int indexb) {
	T a = queue.get(indexa);
	queue.set(indexa, queue.get(indexb));
	queue.set(indexb, a);
}
```

Primer...

```
class Kandidat implements Comparable<Kandidat> {
	private String ime;
	private int brBodovaSkola;
	private int brBodovaPrijemni;

	public Kandidat(String ime, int brBodovaPrijemni, int brBodovaSkola) {
		this.brBodovaSkola = brBodovaSkola;
		this.brBodovaPrijemni = brBodovaPrijemni;
		this.ime = ime;
	}

	public String toString() {
		return ime + ", " + brBodovaPrijemni + ", " + brBodovaSkola;
	}

	public int compareTo(Kandidat drugi) {
		int uk1 = brBodovaSkola + brBodovaPrijemni;
		int uk2 = drugi.brBodovaSkola + drugi.brBodovaPrijemni;

		return uk1 - uk2;
	}
}
```

Primer...

```
public class ExPriorityQueue {
    public static void main(String[] args) {
        PriorityQueue<Kandidat> pq = new HeapPQ<Kandidat>(3);
        pq.insert(new Kandidat("Pera", 10, 20));
        pq.insert(new Kandidat("Zika", 15, 18));
        pq.insert(new Kandidat("Mika", 12, 14));
        pq.insert(new Kandidat("Tika", 22, 22));
        pq.insert(new Kandidat("Cale", 20, 20));
        pq.insert(new Kandidat("Sale", 11, 25));

        while (!pq.isEmpty()) {
            Kandidat k = pq.delMax();
            System.out.println("Upisujem kandidata " + k);
        }
    }
}
```

```
Upisujem kandidata Tika, 22, 22
Upisujem kandidata Cale, 20, 20
Upisujem kandidata Sale, 11, 25
Upisujem kandidata Zika, 15, 18
Upisujem kandidata Pera, 10, 20
Upisujem kandidata Mika, 12, 14
```

Primer 2

```
public static <T extends Comparable<T>>
void sort(T[] arr) {
     PriorityQueue<T> pq = new HeapPQ<T>(arr.length);
     for (T el : arr)
          pq.insert(el);

     for (int i = arr.length - 1; i >= 0; i--)
          arr[i] = pq.delMax();
}
```