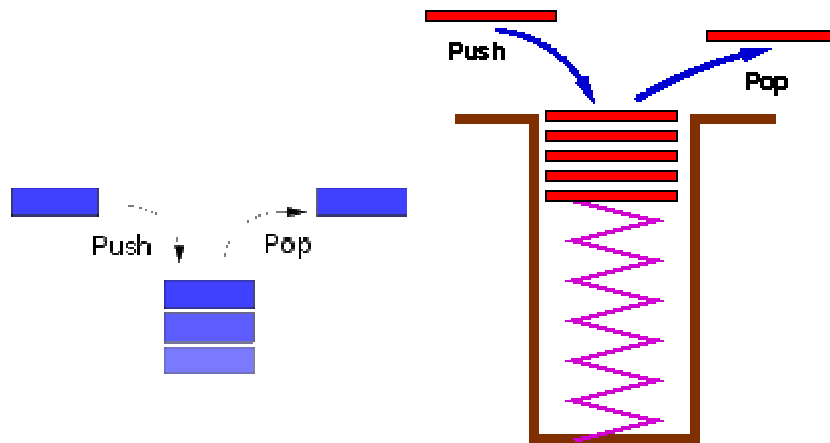
Stek (Stack)



Stek je last in, first out (LIFO) apstraktni tip podataka. Na steku se mogu čuvati bilo koji elementi. Postoje dvije osnovne operacije sa stekom: *push* i *pop*. Operacija push dodaje element na vrh steka (engl. top of the stack). Operacija pop uklanja element sa vrha steka i vraća ga pozivaču.

Elementi se uklanjaju sa steka u redosljedu obrnutom od onoga u kojem su dodavani.

Druge operacije sa stekom

Pored operacija "push" i "pop", postoje i operacije top, remove i isempty..

Apstraktna definicija

Ako N označava element (u ovom slučaju prirodan broj) i U označava uniju:

```
init: -> Stack
push: N x Stack -> Stack
top: Stack -> (N U ERROR)
remove: Stack -> Stack
isempty: Stack -> Boolean
```

Semantika

```
top(init()) = ERROR
top(push(i,s)) = i
remove(init()) = init()
remove(push(i, s)) = s
isempty(init()) = true
isempty(push(i, s)) = false
```

Implementacija

U većini jezika, stek se može implementirati pomoću niza ili olančane liste.

Implementacija - jezik C/C++

Pomoću niza

Kreira se niz u kome je prvi element onaj koji se stavlja na stek i posljednji element koji se uklanja sa steka. Stek se inplementira kao struktura sa dva polja:

```
typedef struct {
    int size;
    int items[STACKSIZE];
} STACK;
```

Funkcija `push()` se koristi da inicijalizuje stek i da sačuva vrijednost. Takođe, provjerava se da li ima prostora u nizu za novi element.

```
void push(STACK *ps, int x)
{
    if (ps->size == STACKSIZE) {
        fputs("Error: stack overflow\n", stderr);
        abort();
    } else
        ps->items[ps->size++] = x;
}
```

Funkcija `pop()` uklanja elementa sa steka i provjerava da li je stek prazan.

```
int pop(STACK *ps)
{
    if (ps->size == 0){
        fputs("Error: stack underflow\n", stderr);
        abort();
    } else
        return ps->items[--ps->size];
}
```

Pomoću olanačane liste

Ova implementacija steka je još prostija od one pomoču niza. Obična olančana lista u kojoj je moguće dodavanja na vrh liste i brisanje sa vrha liste završava posao:

```
typedef struct stack {
    int data;
    struct stack *next;
} STACK;
```

Ovakav čvor je identičan čvoru u tipičnoj olančanoj listi u jeziku C.

Funkcija `push()` ekvivalentna je dodavanju novog čvora na početak liste:

```
void push(STACK **head, int value)
```

```
{
    STACK *node = malloc(sizeof(STACK));  /* create a new node */

    if (node == NULL){
        fputs("Error: no space available for node\n", stderr);
        abort();
    } else {        /* initialize node */
        node->data = value;
         /* insert new head if any */
        node->next = empty(*head)?NULL:*head;
        *head = node;
    }
}
```

Funkcija `pop()` uklanja glavu liste:

```
int pop(STACK **head)
{
    if (empty(*head)) {                        /* stack is empty */
       fputs("Error: stack underflow\n", stderr);
       abort();
    } else {                                   /* pop a node */
        STACK *top = *head;
        int value = top->data;
        *head = top->next;
        free(top);
        return value;
    }
}
```

Implementacija - jezik Pascal

Pomoću niza

```
Program StekNiz;

const
       EmptyTOS = 0;
       MinStackSize =  5;
       Duz = 100;
Type
       ElementType = integer;

       niz = array[1..DUZ] of ElementType;

       Stack = record
             Capacity:integer;
             TopOfStack:integer;
             Memory: niz ;
       end;

var
       st1:stack;
       el:ElementType;
        i:integer;
```

```
Function IsEmpty( S:Stack ):boolean;
begin
        IsEmpty := S.TopOfStack = EmptyTOS;
end;

Function IsFull( S:Stack ):boolean;
begin
        IsFull := S.TopOfStack = S.Capacity ;
end;

Procedure MakeEmpty( var S:Stack );
begin
        S.TopOfStack := EmptyTOS;
end;

Procedure CreateStack( MaxElements:integer; var S:Stack );
begin
        if( MaxElements < MinStackSize ) or (MaxElements > Duz) then
        writeln( 'Stek nije odgovarajuce velicine. Mora imati vise od ',
MinStackSize, ' i manje od ', Duz, ' elemenata.' )
    else
        begin
                S.Capacity := MaxElements;
                S.TopOfStack := 1;
        end;
        MakeEmpty( S );
end;

Procedure Push( X:ElementType; var S:Stack );
begin
        if( IsFull( S ) ) then
                writeln( 'Stek je pun!' )
        else
                begin
                        S.TopOfStack := S.TopOfStack + 1;
                        S.Memory[ S.TopOfStack ] := X;
                end;
end;

Function Pop( var S:Stack ):ElementType;
begin
        if( not IsEmpty( S ) ) then
                begin
                        Pop := S.Memory[ S.TopOfStack ];
                        S.TopOfStack := S.TopOfStack - 1;
                end
        else
                begin
                writeln( 'Stek je prazan!' );
                pop := 0;
        end;
end;

procedure DisposeStack(var S:Stack );
begin
     while (not IsEmpty(S)) do
      writeln(Pop(S));
end;
```

Pomoću olančane liste

```
program stek(input,output);
 type sledeci = ^Lista;
       Lista  = record
                  el:integer;
                  next:sledeci;
       end;
 var p,q :sledeci;
     x:integer;

procedure push(var lst:sledeci);
  var  p:sledeci;
Begin
       new(p);
       read(p^.el);
       p^.next := lst;
       lst:= p;
end; {Procedure push}

procedure pop (var lst:sledeci);
   var p:sledeci;
begin
        if lst = nil then
           writeln('Prazan')
        else
        begin
             p:=lst;
             lst:= lst^.next;
             writeln(p^.el);
             dispose(p);
        end;
end; {pop}
```

Implementacija - jezik C++ (STL)

```
#include <iostream>
#include <stack>

using namespace std;

int main ()
{
    stack <string> cards; /* stek stringova */
    cards.push("King of Hearts"); /* dodajemo kartu */
    cards.push("King of Clubs");  /* dodajemo kartu */
    cards.push("King of Diamonds");
    cards.push("King of Spades");
    cout << "U spilu imamo " << cards.size () << " karata " << endl;
    cout << "Karta na vrhu je " << cards.top() << endl;
    /* Stampace King of Spades, jer je on posljednji dodat */
    cards.pop();
    cout << "Karta na vrhu je " << cards.top() << endl;
    cout << cards.size();
    cin.get ();
    return EXIT_SUCCESS;
}
```

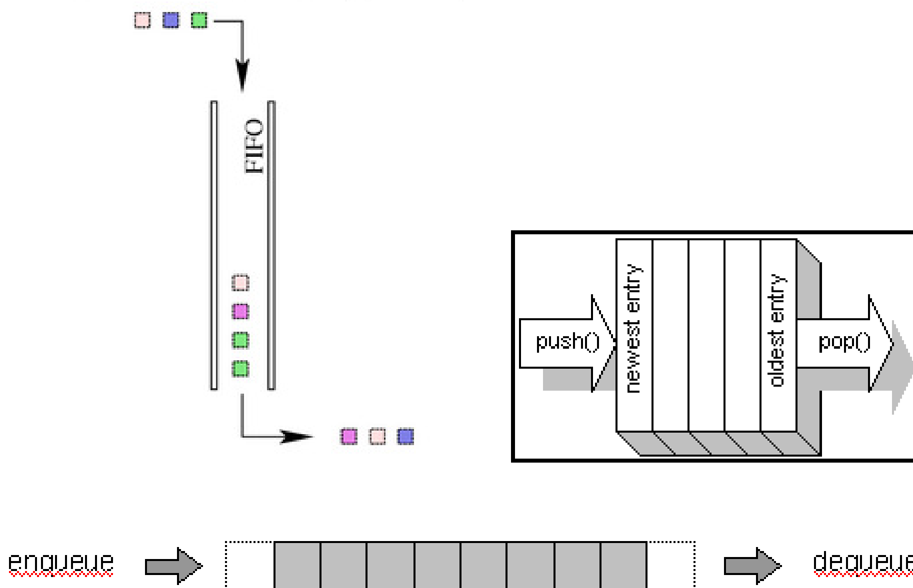
Implementacija - jezik Java

Koristimo ugrađenu klasu `Stack` iz paketa `java.util.*`. Objekti ove klase (kao npr. `st`) ne mogu da primaju primitivne tipove (kao što su `int`, `double`, `boolean`, `char`) već samo objekte. Zbog toga se, u našem primjeru, cio broj `x` pretvara u objekat klase `Integer` pri upisivanju u stek (kod operacije `push`) i prilikom izvlačenja iz steka (operacija `pop`).

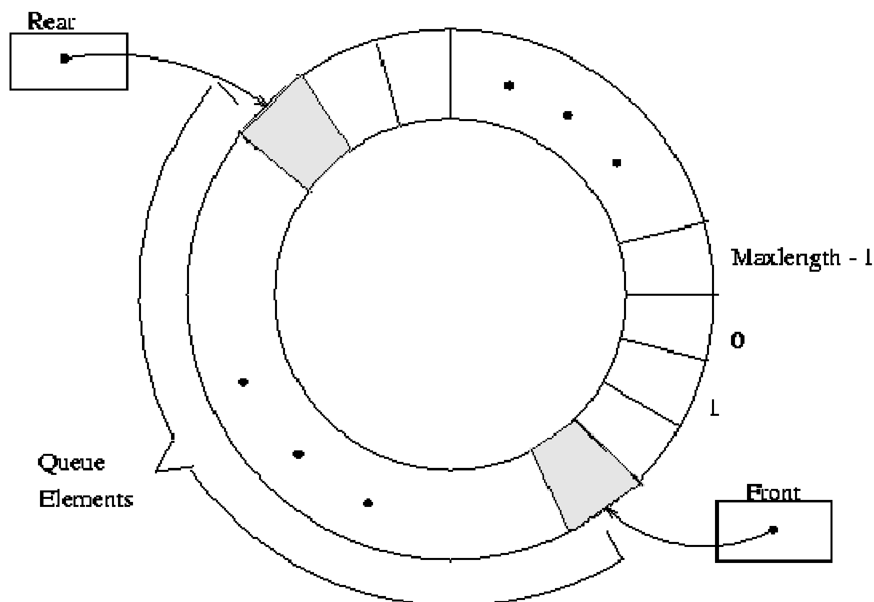
```
import java.util.*;
...
Stack st = new Stack();
Random r = new Random();

for (int i = 0; i<10; i++) // popunjavamo stek sa 10 slucajnih brojeva
{
	int x = r.nextInt(100);
	st.push(new Integer(x));
}

Integer a;
System.out.printf("\nSadrzaj steka\n");
while (!st.empty())
{
	a = (Integer)st.pop();
	System.out.printf("%4d", (int)a.intValue());
}
```

Red (Queue)

Red je first in, first out (FIFO) apstraktni tip podataka. U redu se mogu čuvati bilo koji elementi. Postoje dvije osnovne operacije sa redom: `push (ili enqueue ili addQ)` i `pop (ili dequeue ili removeQ)`. Operacija `push` (`enqueue`) dodaje element na kraj reda (engl. `rear` ili `last` ili `back`). Operacija `pop` (`dequeue`) uklanja element sa vrha reda (engl. `front` ili `first`) i vraća ga pozivaču. Elementi se uklanjaju iz reda u istom redosljedu u kojem su dodavani.



Implementacija preko cirkularnog niza



Koristimo cjelobrojne front i rear.

- front je jednu poziciju suprotno kazaljki sata od prvog elementa
- rear daje poziciju posljednjeg elementa

Operacija dodavanja elementa:

- Pomjeri se rear jedno mjesto u smjeru kazaljke na satu.
- Zatim se element stavi u queue[rear].

Operacija uklanjanja elementa:

- Pomjeri se front jednu poziciju u smjeru kazaljke sata.
- Zatim se vrati queue[front].

Pomjeranje u smjeru kretanja kazaljke sata (i za front i za rear):

```
    rear++;
    if (rear = = capacity) rear = 0;
ili rear = (rear + 1) % capacity;
```

Situacija front==rear se pojavljuje u dvije situacije:

- Red je prazan
- Red je pun (svi elementi niza su popunjeni).

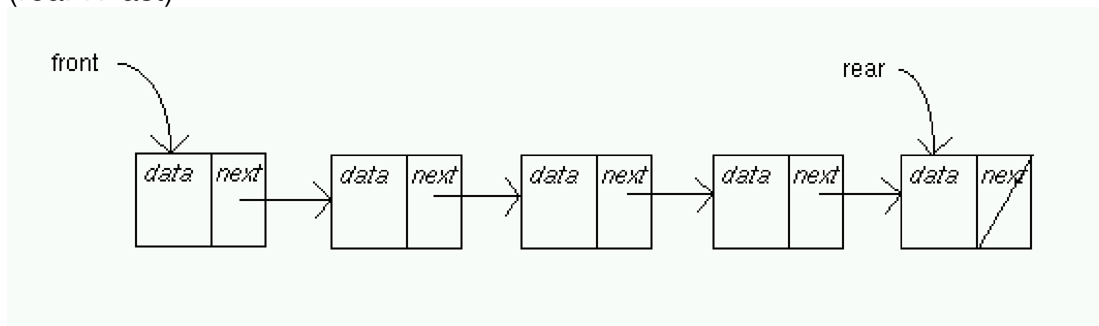
Da bi otklonili ovaj problem, možemo uraditi sljedeće:

- Ne dozvoliti da se red napuni.
  - Ako dodavanje elementa uzrokuje prepunjavanje, povećati veličinu niza (moguće u C/C++ i Java-i, nije moguće u Pascal-u).
- Definisati Bulovsku promjenljivu lastOperationIsAddQ.
  - Svako dodavanje u red postavlja ovu promjenljivu na true.
  - Svako brisanje iz reda postavlja ovu promjenljivu false.
  - Red je prazan ako je (front == rear) && !lastOperationIsAddQ
  - Red je pun ako je (front == rear) && lastOperationIsAddQ
- Definisati cjelobrojnu promjenljivu size.
  - Svako AddQ odrađuje i size++.
  - Svako DeleteQ odrađuje i size--.
  - Red je prazan ako je (size == 0)
  - Red je pun ako je (size == arrayLength)

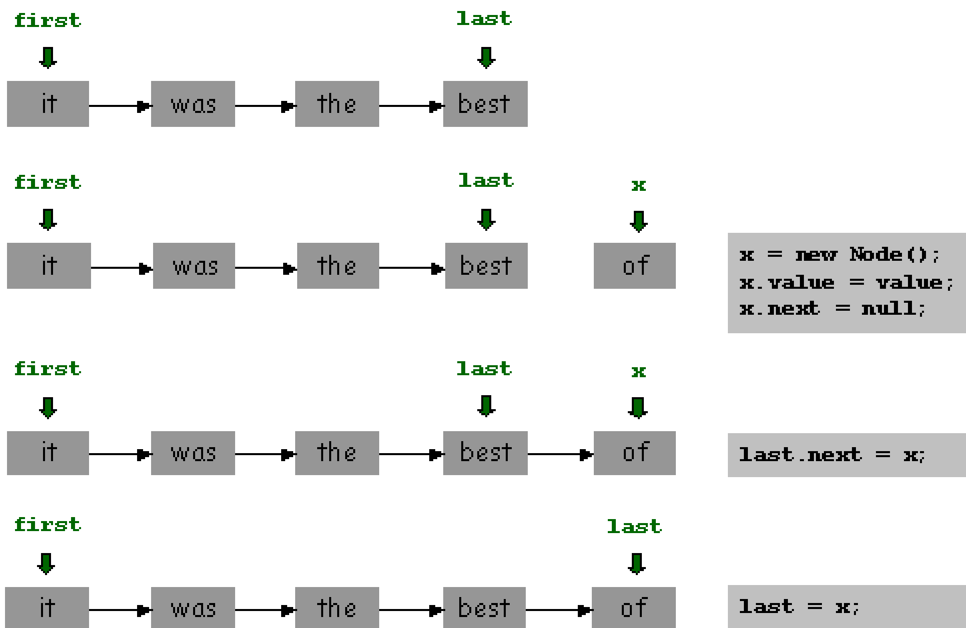
Za detalje kako rade operacije dodavanja i uklanjanja elementa u red pogledati prezentaciju Queue.ppt.

Implementacija preko olančane liste

Red se opisuje jednostruko olančanom listom ili dvostruko olančanom listom. U oba slučaja imamo pokazivač na prvi element liste (first ili front) i pokazivač na posljednji element liste (rear ili last).



Primjer dodavanja elementa prikazan je na sljedećoj slici:



Implementacija – jezik C++ (STL)

```
/* STL queue primjer */

#include <iostream>
#include <queue>

using namespace std;

/* simple queue example */
void functionA()
{
  queue <int> q;                //q je red cijelih brojeva

  q.push(2);                    //dodajemo 2, 5, 3, 2 u red
  q.push(5);
  q.push(3);
  q.push(1);
  cout<<"q ima " << q.size() << " elemenata.\n";

  while (!q.empty()) {
    cout << q.front() << endl;    //stampa prvi element u redu
    q.pop();                      //brise prvi element iz reda
  }
}

int main()
{
  cout << "calling functionA...\n";
  functionA();

  return 0;
}
```

Implementacija – jezik Java

Postoji interfejs Queue, ali kao i za klasu Stack, u redu moramo čuvati objekte a ne primitivne tipove. Osnovne operacije sa redom su prikazane u tabeli:

Queue Interface Structure		
Operacija	**Throws exception**	**Returns special value**
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Primjer:

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {

    public static void main(String[] args) {

        Queue<String> qe=new LinkedList<String>(); // red stringova
        // dodavanje 5 stringova
        qe.add("b");
        qe.add("a");
        qe.add("c");
        qe.add("e");
        qe.add("d");

        Iterator it=qe.iterator();

        System.out.println("Initial Size of Queue :"+qe.size());

        while(it.hasNext())
        {
            String iteratorValue=(String)it.next();
            System.out.println("Queue Next Value :"+iteratorValue);
        }

        // get value and does not remove element from queue
        System.out.println("Queue peek :"+qe.peek());

        // get first value and remove that object from queue
        System.out.println("Queue poll :"+qe.poll());

        System.out.println("Final Size of Queue :"+qe.size());
    }
}
```

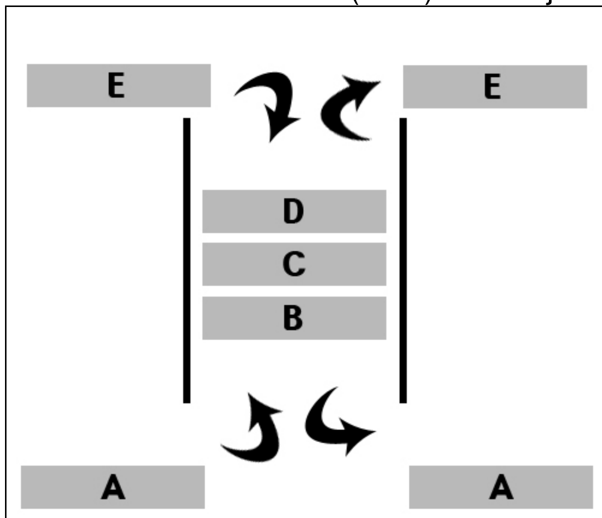
Izlaz:

Initial Size of Queue :5
Queue Next Value :b
Queue Next Value :a

Queue Next Value :c
Queue Next Value :e
Queue Next Value :d
Queue peek :b
Queue poll :b
Final Size of Queue :4

Dvostrani red (dequeue, deque, deck)

Samo ime strukture nam govori o čemu se radi: red kod koga je moguće dodavati i brisati elemente i sa vrha reda (front) i sa kraja reda (rear ili back).



Načini implementacije su isti kao i kod reda. Stek i red možemo posmatrati kao specijalne slučajeve ove strukture.
U jezicima C++ i Java postoje implementacija ove strukture. Spisak operacija dat je u tabeli:

Operacija	C++	Java
dodavanje elementa na kraj	`push_back`	`offerLast`
dodavanje elementa na početak	`push_front`	`offerFirst`
brisanje posljednjeg elementa	`pop_back`	`pollLast`
brisanje prvog elementa	`pop_front`	`pollFirst`
vrijednost posljednjeg elementa	`back`	`peekLast`
vrijednost prvog elementa	`front`	`peekFirst`

Implementacija – jezik C++ (STL)

U jeziku C++, STL ima podršku za dvostrani red – potrebno je odraditi `#include <deque>`.

```
#include <iostream>
#include <stdexcept>   // zbog excepions
#include <deque>

using namespace std;

int main() {
        //deque sa 5 elemenata, svi imaju vrijednost 8.1
        deque<double> dq(5, 8.1);
```

```
        for (int i=0; i<=5; ++i) {
                cout << "Element "<< i << " sa [] : "<< dq[i]<< endl;
        }

        try {
            cout << "Element " << i << " sa at(i) :" << dq.at(i) << endl;
        } catch (out_of_range&) {
            cout << "**out of range za element " << i << " with at() **" << endl;
        }

        return 0;
}
```

Implementacija – jezik Java

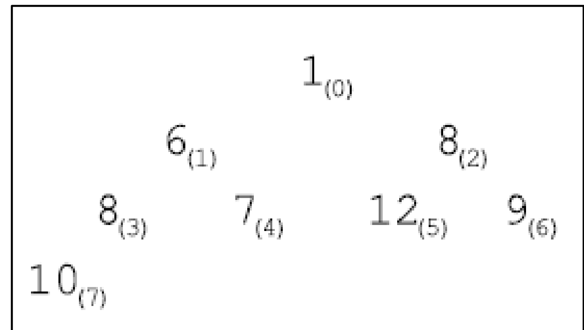
U programskom jeziku Java, `Deque` (obratite pažnju na skraćeni naziv) je intefejs koji se može implementirati preko niza (`ArrayDeque`) ili preko olančane liste (`LinkedList`).

Primjer Java koda koji koristi ovu strukturu:

```
import java.util.Deque
...
Deque dequeA = new LinkedList();
Deque dequeB = new ArrayDeque();
...
```

Hrpa (heap)

Hrpa ili gomila (engl. heap) je struktura podataka koja vraća minimalni (ili maksimalni) element za O(1), a dodaje novi element i uklanja rekući minimum (masksimum) za O(logn). Druge operacije za hrpu obično nisu definisane. Ova se struktura često koristi za implementaciju prioritetnog reda.



Pogledajmo kako se hrpa može realizovati pomoću niza. Elementi sa indeksima 2*i+1 i 2*i+2 su potomci elementa sa indeksom i, indeksi počinju od 0. Potomci dva različita elementa se ne sijeku i svaki element je nečiji potomak (osim elementa sa indeksom 0). Osnovno svojstvo hrpe je da je svaki element manji ili jednak od svojih potomaka. Na primjer, niz 1,6,8,7,12,8,10 je hrpa.

Kreirajmo strukturu za hrpu:

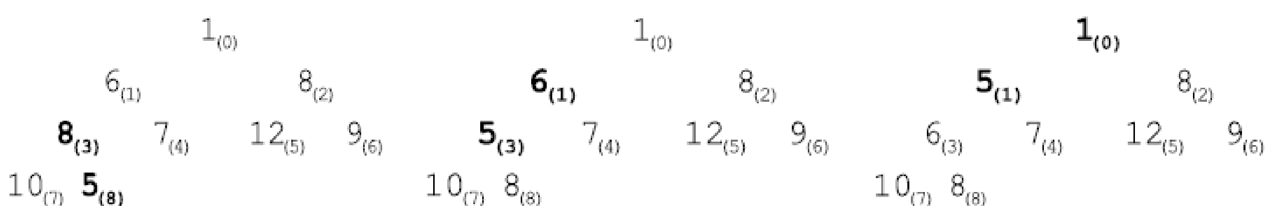
```
typedef struct
{
      int hs;
      int val[MAXN];
} heap;
```

Opišimo tri funkcije za rad sa hrpom. Prvo je funkcija koja vraća najmanji elemenet. Ona je jednostavna jer se najmanji element nalazi na pozicji 0.

```
int get_min(heap *h)
{
return h->val[0];
}
```

Prije poziva ove funkcije, obavezno provjeriti da hrpa nije prazna.

Druga funckija je za dodavanje elementa u hrpu. Dodamo ga na kraj niza, a zatim ga zamjenjujemo sa njegovim precima sve dok ne postane manji od pretka ili dok indeks ne bude 0. Svojstvo hrpe se ne narušava, a složenost operacije je O(logn). Primjer dodavanja u hrpu prikazan je na slici:



```
void add_heap(heap *h, int x)
{
      int y, pos=h->hs, npos;
```

```
	h->val[h->hs++] = x;
	npos=(pos-1)/2;
	while (pos && h->val[pos] < h->val[npos]) {
		y=h->val[pos];
		h->val[pos]=h->val[npos];
		h->val[npos] = y;
		pos=npos;
		npos=(pos-1)/2;
	}
}
```

Treća funkcija je za udaljavanje minimalnog elementa iz hrpe. Na mjesto elementa sa indeksom 0 zapišemo posljednji element niza, smanjimo veličinu niza za 1 i zatim poguramo element iz korijena na njegovo pravo mjesto. Ako je element veći od manjeg od svojih potomaka, onda oni zamijene mjesta; produžavamo postupak dok nije zadovoljen uslov ili dok ne izađemo van granica hrpe.

```
void del_heap(heap *h)
{
	int minp, pos=0, y;
	h->val[0]=h->val[--h->hs];
	while (pos*2+1 < h->hs) {
		y=pos*2+1;
		minp=h->val[y]<h->val[y+1]?y:y+1;
		if (h->val[pos]>h->val[minp]) {
			y=h->val[pos];
			h->val[pos]=h->val[minp];
			h->val[minp]=y;
			pos=minp;
		}
		else break;
	}
}
```

Na prvi pogled, jedino mjesto gdje je teorijski moguća greška jeste kada imamo samo jednog potomka, tj. pos*2+2 == h->hs. Takva varijanta je moguća ako je broj elemenata u hrpi paran. U tom slučaju, mi određujemo manji od prvog potomka i prvo elementa van hrpe, što izgleda kao greška. Međutim, znamo da su element koji guramo i prvi element van granice niza jednaki, pa neće biti greške.

Implementacija – jezik Java

Implementacija hrpe pomoću niza, koja u potpunosti odgovara gore opisanim funkcijama.

```
public class Heap {

	int hs; // velicina hrpe
	int[] val; // niz za cuvanje elemenata

	public Heap()
	{
		hs = 0;
		val = new int[10];
	}
```

```
	public Heap(int n)
	{
		if (n<0) n = 10;
		hs = 0;
		val = new int[n];
	}

	public int get_min()
	{
		return val[0];
	}

	public void add_heap( int x)
	{
		int y, pos= hs, npos;
		val[hs++] = x;
		npos=(pos-1)/2;
		while (pos>0 && val[pos] < val[npos]) {
			y= val[pos];
			val[pos]=val[npos];
			val[npos] = y;
			pos=npos;
			npos=(pos-1)/2;
		}
	}

	public void del_heap()
	{
		int minp, pos=0, y;
		val[0]= val[--hs];
		while (pos*2+1 < hs) {
			y=pos*2+1;
			minp= val[y]<val[y+1]?y:y+1;
			if (val[pos]>val[minp]) {
				y=val[pos];
				val[pos]=val[minp];
				val[minp]=y;
				pos=minp;
			}
			else break;
		}
	}

	public void print()
	{
		for(int i=0; i<hs; i++)
		{
			System.out.printf("%4d", val[i]);
		}
		System.out.printf("\n");
	}

} // end class Heap


public class TestHeap {

	public static void main(String[] args) {
		Heap h = new Heap(30);
```

```
        h.add_heap(1);
        h.add_heap(12);
        h.add_heap(8);
        h.add_heap(6);
        h.add_heap(7);
        h.add_heap(8);
        h.print();
        System.out.printf("Minimum je %4d\n", h.get_min());
        h.del_heap();
        h.print();
    }

}
```

Prioritetni red (priority queue)

Prioritetni red je apstraktni tip podataka gdje je svakom elementu pridružen prioritet. Ovaj tip podataka mora implementirati bar sljedeće dvije operacije:

- insert_with_priority: dodajemo element sa pridruženim prioritetom u red
- pull_highest_priority_element: brišemo elemnt sa anjvećim prioritetom iz reda i vraćamo ga pozivaču (takođe se može zvati "pop_element(Off)", "get_maximum_element" ili "get_front(most)_element"). Neke implementacije smatraju da elementi čiji je prioritet manji broj imaju veći prioritet, pa se ova operacija često naziva i "get-min".

Prioritetni red može se implemenirati na više načina: pomoću niza, liste, hrpe (heap) itd.

Implementacija – jezik C++ (STL)

STL ima podršku za prioritetni red. Dovoljno je napočetku koda dodati #include <queue>. Važno je da se elementi koje dodajemo u prioritetni red mogu upoređivati, tj. da je za njih definisan operator manje (<). To se posebno odnosi ako u red dodajemo objekte neke klase – tada se za tu klasu mora definisati komparator ili preopteretiti operator <. Ako u red dodajemo elemente tipa za koji je definisan operator <, onda nije potrebno ništa raditi, ako je to poredak koji nam odgovara (kao u prvom primjeru niže). U drugom primjeru, naredba priority_queue <int, vector<int>, greater<int> > pq nam kaže da je pq prioritetni red cijelih brojeva koji se implementira pomoću strukture vector, a kao operator poređenja se ne koristi standardni operator manje (<) već operator veće (>, zbog greater<int>).

```
/* STL priority queue primjeri */

#include <iostream>
#include <queue>

using namespace std;

/* primjer upotrebe prioritetnog reda */
void functionB()
{
  priority_queue <int> pq;      //pq je prioritetni red cijelih brojeva

  pq.push(2);                   //dodajmo 2, 5, 3, 1 u red
  pq.push(5);
  pq.push(3);
  pq.push(1);
  cout<<"pq sadrzi " << pq.size() << " elemenata.\n";

  while (!pq.empty()) {
    cout << pq.top() << endl;   // stampamo element sa najvecim prioritetom
    pq.pop();                   // brisemo element sa najvecim prioritetom
  }
}

/* primjer prioritetnog reda gdje veci prioritet ima manji broj */
void functionC()
```

```
{
  priority_queue <int, vector<int>, greater<int> > pq;

/* pq je proritetni red cijelih brojeva koji se implementira preko vektora korsiti
operator > za odredjivanje prioriteta (tj. ako je a>b, tada a ima manji prioritet
od b)
*/

  pq.push(2);                      //dodamo 2, 5, 3, 2 u red
  pq.push(5);
  pq.push(3);
  pq.push(1);
  cout<<"pq sadrzi " << pq.size() << " elemenata.\n";

  while (!pq.empty()) {
    cout << pq.top() << endl;      // stampamo element sa najvecim prioritetom
    pq.pop();                      // brisemo element sa najvecim prioritetom
  }
}

/* definicija klase Height */
class Height
{
  public:
    Height() {};                                       //default konstruktor
    Height(int x, int y) { feet = x; inches = y; }    //konstructor
    bool operator<(const Height&) const;               //overloaded < operator

    int get_feet() const { return feet; }              //accessor methods
    int get_inches() const { return inches; }

  private:
    int feet, inches;                                  //data fields
};

/* overload operator < da bi znali kako da uporedimo 2 objekta klase Height */
bool Height::operator<(const Height& right) const
{
      return feet*12 + inches < right.feet*12 + right.inches;
}

/* primjer prioritetnog reda koji koristi klasu Height */
void functionD()
{
  priority_queue <Height> pq;      //pq is a priority queue of Height objects

  Height x;

  x = Height(10,20);      //dodajemo put 10'20", 11'0", 8'25" i 9'4" u red
  pq.push(x);
  x = Height(11,0);
  pq.push(x);
  x = Height(8,25);
  pq.push(x);
  x = Height(9,4);
  pq.push(x);

  cout<<"pq contains " << pq.size() << " elements.\n";
```

```
  while (!pq.empty()) {
    cout << pq.top().get_feet()
         << "' " << pq.top().get_inches() << "\"" << endl;
    pq.pop();
  }
}

int main()
{
  cout << "calling functionB...\n";
  functionB();
  cout << "calling functionC...\n";
  functionC();
  cout << "calling functionD...\n";
  functionD();

  return 0;
}
```

Implementacija – jezik Java

Klasa `PriorityQueue` je implementacija prioritetnog reda. Kao i klase `Stack` i `Queue`, i ova klasa dopušta da dodajemo samo objekte a ne primitivne tipove. Na objektima klase T koje dodajemo u red mora biti definisana operacija poređenja, pomoću funkcije `compareTo` (pogledati klasu `Test` u primjeru 2) tj. klasa T mora implementirati interfejs (interface) `Comparable<T>`. Funkcija `compareTo` treba da vrati negativan broj, nulu ili pozitivan broj (obično su to -1, 0, 1) u zavisnosti da li je prvi objekat manji, jednak ili veći od drugog. U prvom primjeru, u red dodajemo stringove, a na klasi `String` je već definisan poredak.

Primjer 1:

```
import java.util.*;

public class PriorityQueueDemo {

      PriorityQueue<String> stringQueue;


      public static void main(String[] args){

      stringQueue = new PriorityQueue<String>();

      stringQueue.add("ab");
      stringQueue.add("abcd");
      stringQueue.add("abc");
      stringQueue.add("a");

      while(stringQueue.size() > 0)
             System.out.println(stringQueue.remove());

      }

}
```

Primjer 2:

```java
class Test implements Comparable<Test> {
   private int priority; // prioriret
   public int x,y; // vrijednosti

   // konstruktori
   public Test(int priority) {
      this.priority = priority;
      x = 0;
      y = 0;
   }

   public Test(int priority, int xx, int yy) {
            this.priority = priority;
            x = xx;
            y = yy;
   }

   public int compareTo(Test o) {
      if (this.priority < o.priority)
         return -1;
      else if (this.priority > o.priority)
         return 1;
      return 0;
   }

   public int getPriority() {
      return this.priority;
   }
}

import java.util.*;
public class PriorityQueueExample {

   public static void main(String args[]) {

      Queue<Test> queue = new PriorityQueue<Test>(); // kreiramo novi red
      Test t;
      // Dodamo tri objekta klase Test u red
      queue.offer(new Test(3));
      queue.offer(new Test(1));
      queue.offer(new Test(2));

      // dodamo 10 slucajnih objekata klase Test
      Random r = new Random();
      for(int i=0; i<10; i++)
      {
        queue.offer(new Test(1+r.nextInt(20),1+r.nextInt(20),-10+r.nextInt(30)));
      }

      // ispraznimo red, elementi ce biti poredjani po neopadajucim prioritetima
      while (queue.size()!= 0) {
            t = queue.poll();
            System.out.printf("%d %d %d\n", t.getPriority(), t.x, t.y);
      }
   }
}
```