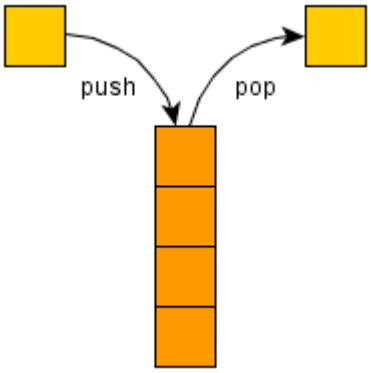
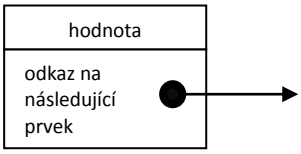
Zásobník (Stack), Fronta (Queue), Kruhový buffer, Spojový seznam, Množina

Zásobník = Stack



- je jednou ze základních datových struktur, která se využívá především pro dočasné ukládání dat v průběhu výpočtu.
- data ukládá způsobem LIFO - last in, first out - čili poslední vložený prvek jde na výstup jako první, předposlední jako druhý a tak dále.
- pokud je implementace polem, pak je vrchol VPRAVO.
- každý prvek má dvě části – hodnotu a odkaz na následující prvek



Základní operace - abstraktní datový typ zásobník specifikuje tyto operace:

- **push** - vloží prvek na vrch zásobníku
- **pop** - odstraní vrchol zásobníku
- **top** - dotaz na vrchol zásobníku
- **isEmpty** - dotaz na prázdnost zásobníku (size - dotaz na velikost zásobníku)

Využití
Zásobník se v informatice používá zejména pro ukládání stavu algoritmů a programů. Je použit v Tarjanově algoritmu, v prohledávání do hloubky a implicitně ve všech rekurzivních algoritmech. Na zásobníkové architektuře jsou postaveny virtuální stroje pro jazyky Java a Lisp.

Implementace spojovym seznamem

```
public class Stack {
    private Node first;
    private int size;

    public Stack() {
        this.size = 0;
    }

Ulozi prvek na vrch zasobniku
     Slozitost - O(1)
     @param i prvek k ulozeni
public void push(int i) {
    Node n = new Node(i);

    Node currFirst = first;
    first = n;
    n.next = currFirst;

    size++;
}

Odstrani vrchni prvek ze zasobniku
      Slozitost - O(1)
      @return hodnota vrchniho prvku
public int pop() {
    if (size == 0) {
        throw new IllegalStateException ("Zasobnik je prazdny, nelze odebrat
        prvek");
    }
    int value = first.value;
    first = first.next;
    size--;
    return value;
}
```

```
Vrati vrchol zasobniku
      Slozitost - O(1)
      @return hodnota vrchniho prvku
public int top() {
    if (size == 0) {
        throw new IllegalStateException("Zasobnik je prazdny, nelze
        vratit");
    }
    return first.value;
}

Dotaz na prazdnost
      @return true, pokud je fronta prazdna
public boolean isEmpty() {
    return this.size == 0;
}

Vrati velikost zasobniku
      @return velikost zasobniku
public int getSize() {
    return this.size;
}

Klasicka toString metoda, vraci textovou reprezentaci objektu
      @return textova reprezentace objektu
@Override
public String toString() {
    StringBuilder builder = new StringBuilder();
    Node curr = first;
    for (int i = 0; i < this.size; i++) {
        builder.append(curr.value).append(" ");
        curr = curr.next;
    }
    return builder.toString();
}

Vnitrni trida reprezentujici uzel spojoveho seznamu
private class Node {
    private int value;
    private Node next;

    private Node(int value) {
    this.value = value;
    }
}
```

Implementace polem

Rozhraní zásobníku můžeme definovat např. takto:

```
class IntZasobnik {
  IntZasobnik()        // Vytvoreni prazdneho zasobniku
  boolean jePrazdny()  // Test jestli je zasobnik prazdny
  void push(int)       // Vlozeni prvku
  int pop()            // Vybrani prvku
}
```

Třída **IntZasobnik** bude vypadat např. takto:

```
class IntZasobnik {

  private int[] z;
  private int vrchol;
  final int MAX=10;

  IntZasobnik() {
    z = new int[MAX];
    vrchol = 0;
  }

  boolean jePrazdny() {
    return (vrchol == 0);
  }

  void push(int klic) {
    z[vrchol++] = klic;
  }

  int pop() {
    return z[--vrchol];
  }
}
```

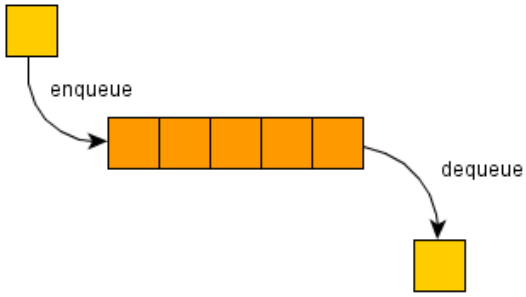
V konstruktoru **IntZasobnik()** se nastaví prázdný zásobník (vytvoří se pole **z** o délce **MAX** a do proměnné **vrchol** se přiřadí hodnota nula - vrchol ukazuje na první položku zásobníku (na položku s indexem 0 v poli **z**.

Jestliže je zásobník prázdný, pak proměnná **vrchol** je nulová a metoda **jePrazdny()** vrátí hodnotu *true*. V ostatních případech *false*.

Metoda push() nejprve do pole s indexem **vrchol** uloží klíč, který je jejím parametrem, a následně zvýší proměnnou **vrchol** o jedničku. Proměnná **vrchol** tedy ukazuje na následující položku a při dalším vkládání tak nedochází k přepisování původních dat.

Metoda **pop()** nejprve sníží hodnotu proměnné **vrchol** o jedničku a následně vrátí hodnotu položky, na kterou ukazuje proměnná **vrchol**.

Fronta = Queue, Buffer



- je jedním ze základních datových typů a slouží k ukládání a výběru dat
- prvek, který byl uložen jako první, je také jako první vybrán – FIFO princip = first in, fist out.
- speciálním případem fronty je tzv. prioritní fronta, ve které mohou prvky s vyšší prioritou předbíhat na výstupu ty s nižší prioritou.
- ve spojovém seznamu má navíc odkaz na začátek nebo na konec, jinak není konstantí složitost.
- v poli – přidání prvku znamená posunutí všech ostatní prvků.
- v poli přidání na konec znamená vložení na první index, tj. i = 0 (= lastIndex)
- lze řešit tzv. kruhovým polem, kde je začátek i konec na i = 0.
- v kruhovém poli je problém zjistit zda je fronta prázdná nebo plná, potřebuji k tomu ještě vědět, kolik jsem uložila prvků.

Typické operace

- **addLast** – Vloží prvek do fronty.
- **deleteFirst** – Získá a odstraní první prvek (hlavu) fronty.
- **getFirst** – Získá první prvek fronty.
- **isEmpty** – Dotaz na prázdnost fronty.
- **size** – Vrátí počet obsažených prvků.

Využití - fronta má v informatice široké využití, toto jsou některé příklady:

- Synchronizační primitivum
- Operátor roura (pipe) - komunikace mezi procesy v operačních systémech
- Kruhový buffer - vyrovnávací paměť pro datové toky
- Řazení prioritní frotou (haldou) - heapsort

Implementace spojovym seznamem

```
public class Queue {
    private Node first;
    private Node last;
    private int size;

    public Queue() {
        this.size = 0;
    }

Prida na konec fronty prvek
      @asymptoticka slozitost O(1)
      @param i prvek k vlozeni
public void addLast(int i) {
    Node n = new Node(i);
    if(getSize() == 0) {
        this.first = n;
        this.last = n;
    } else {
        this.last.next = n;
        this.last = n;
    }
    size++;
}

Odebere z fronty prvni prvek
      @asymptoticka slozitost O(1)
      @return prvni prvek
public int deteteFirst() {
    if(getSize() == 0) throw new IllegalStateException("Fronta je prazdna");
    int value = first.value;
    first = first.next;
    size--;
    return value;
}
```

```
Vrati prvni prvek fronty
      @return prvni prvek
public int getFirst() {
    if(getSize() == 0) throw new IllegalStateException("Fronta je prazdna");
    return first.value;
}

Dotaz na prazdnost
      @return true, pokud je fronta prazdna
public boolean isEmpty() {
    return this.size == 0;
}

Getter na delku
      @return delka fronty
public int getSize() {
    return size;
}

Klasicka toString metoda, vraci textovou reprezentaci objektu
      @return textova reprezentace objektu
@Override
    public String toString() {
    StringBuilder builder = new StringBuilder();
    Node curr = first;
    for(int i = 0;  i < this.size; i++) {
        builder.append(curr.value).append(" ");
        curr = curr.next;
    }
    return builder.toString();
}

Vnitrni reprezentace prvku
private class Node {
    private int value;
    private Node next;

    private Node(int value) {
        this.value = value;
    }
}
```

Implementace polem

Rozhraní fronty můžeme definovat asi takto:

```
class Fronta {
  IntFronta()          // Vytvoreni prazdne fronty
  boolean jePrazdna()  // Test je-li fronta prazdna
  void vloz(int)       // Vlozeni prvku
  int vyber()          // Vybrani prvku
}
```

Prvky fronty jsou uchovávány v poli **f**, konstanta **MAX** určuje maximální délku pole.

Pro efektivnost se prvky fronty ukládané v poli neposouvají (jako v běžném životě), ale udžují se dvě hodnoty indexu **zacatek** a **konec**.
zacatek - index označující místo, odkud vybereme prvek
konec - index označující místo, kam vložíme prvek

V případě, že fronta je prázdná, jsou si oba indexy rovny.

Kvůli jednoduchosti vytvoříme pole pro implementaci fronty o 1 větší, než je maximální počet prvků fronty, a rovnost indexů bude znamenat prázdnou frontu.

Třída **IntFronta** bude vypadat např.

```
class IntFronta {

    private int[] f;
    private int zacatek, konec;
    final int MAX=5, n;

    IntFronta() {
      n = MAX + 1;
      f = new int[n];
      zacatek = 0;
      konec = 0;
    }

    boolean jePrazdna() {
      return (zacatek == konec);
    }

    void vloz(int klic) {
      f[konec++] = klic;
      konec = konec % n;
    }

    int vyber() {
      int v = f[zacatek++];
      zacatek = zacatek % n;
      return v;
    }
  }
```

Při vkládání prvků se prvek nejprve vloží na místo, kam ukazuje **konec**, proměnná **konec** se následně inkrementuje (ukazuje na další pozici - ta je prázdná, na ní se pak ukládá případný další prvek).

Proměnná **zacatek** zůstává nezměněna, tedy ukazuje na prvek, který jsme vložili jako první.

Při operaci **vyber()** se do proměnné **v** uloží hodnota prvku, na který ukazuje proměnná **zacatek**, která se následně inkrementuje, a ukazuje tak na prvek, který jsme vložili jako druhý atd.

V okamžiku, kdy jsou si indexy **zacatek** a **konec** rovny, je fronta prázdná.

Fronta se chová jako kruhová. Pokud do fronty vložíme dva prvky a následně je vybereme, oba ukazatelé se setkají na pozici s indexem 2.

Při dalším vkládání se vkládá na pozici s indexem 2, proměnná **konec** se inkrementuje, **zacatek** zůstává stejný.

V okamžiku, kdy se dosáhne konce pole, má **konec** hodnotu rovnu proměné **n**, následně se provede dělení modulo n. Výsledek tohoto dělení je nula - další prvky se vkládají od začátku pole.

Stejným způsobem se mění **zacatek** při vybírání prvků. Musíme ale dávat pozor na to, abychom nepřekročili rozsah fronty - pokud bychom např. do fronty pro 5 prvků vložili šestý prvek, pak by ukazatel **konec** dosáhl úrovně ukazatele **zacatek** a fronta by se chovala jako prázdná a při dalším vkládání by se přepisovala původně vložená data.

Kruhový buffer

Kruhový buffer (Circular buffer) je implementací fronty (FIFO) nad polem, která se používá především jako vyrovnávací paměť v datových tocích.

Princip

Kruhový buffer se skládá z pole fixní délky a dvou ukazatelů. První z ukazatelů míří na první obsazený prvek, druhý na první volné místo v poli. V okamžiku, kdy je do bufferu přidán nový prvek, tak se druhý ukazatel zinkrementuje, v případě odstranění prvního prvku fronty dojde k inkrementaci prvního ukazatele (a k přemazání příslušného paměťového segmentu). Jelikož jsou obě inkrementace prováděny modulárně (pokud je přidáván prvek a konec pole je již obsazen, tak je prvek přidán na uvolněný začátek pole), tak struktura bufferu tvoří kruh.

Složitost operací
Asymptotická složitost výběru a čtení prvku na prvním indexu je O(1), složitost operace přidání prvku na konec fronty je O(1).

```
Kruhovy buffer - fronta nad polem
public class CircularBuffer<ENTITY> {
    private int size;
    private Object[] array;
    private int pointer;              //prvni volny index

Konstruktor
      @param length velikost bufferu
public CircularBuffer(int length) {
    this.array = new Object[length];
    this.size = 0;
    pointer = 0;
}

Pridej prvek na konec bufferu
      @param i prvek
public void addLast(ENTITY i) {
    if (this.size == array.length) {
        throw new IllegalStateException("Buffer is full");
    }
    array[pointer] = i;
    pointer = modulo((pointer + 1), array.length);
    size++;
}

Vrat a smaz prvni prvek
      @return prvni prvek
public ENTITY getFirst() {
    if (this.size == 0) {
        throw new IllegalStateException("Buffer is empty");
    }
    ENTITY value = (ENTITY) array[modulo((pointer - size), array.length)];
    array[modulo((pointer - size), array.length)] = null;
    size--;
    return value;
}

Precti prvni prvek
      @return prvni prvek
public ENTITY readFirst() {
    if (this.size == 0) {
        throw new IllegalStateException("Buffer is empty");
    }
    ENTITY value = (ENTITY) array[modulo((pointer - size), array.length)];
    return value;
}
```

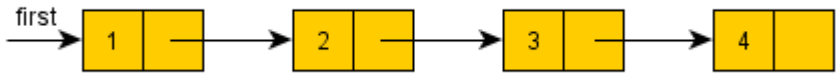
```
x ≡ number mod(modulo)
      @param number number
      @param modulo modulo
      @return nejmensi nezaporne residuum
private int modulo(int number, int modulo) {
    if (number >= 0) {
        return number % modulo;
    }
    int result = number % modulo;
    return result == 0 ? 0 : result + modulo;
}

Pocet obsazenych prvku
      @return pocet prvku v bufferu
public int getSize() {
    return this.size;
}

toString
@Override
public String toString() {
    StringBuilder builder = new StringBuilder();
    builder.append("Content: ");
    for (int i = 0; i < size; i++) {
        builder.append(array[modulo((pointer - size + i), array.length)
        ]).append(" ");
    }
    builder.append("\nfirst index: ").append(modulo((pointer - size),
    array.length)).append(", last index:").append(pointer - 1).append(",
    size: ").append(size);

    return builder.toString();
}

Velikost bufferu
      @return maximalni pocet prvku v bufferu
public int getLength() {
    return array.length;
}
```

Spojový seznam = List (Lineární seznam, Linked list)

- je kontejner určený k ukládání dat předem neznámé délky.
- základní stavební jednotkou spojového seznamu je uzel, který vždy obsahuje ukládanou hodnotu a ukazatel na následující prvek.
- může být jednoduše nebo dvojitě zřetězený.
- může být setříděný (uspořádaný) nebo nesetříděný (neuspořádaný)
- může být zacyklený nebo ne
- může být s hlavou (zarážkou) nebo bez

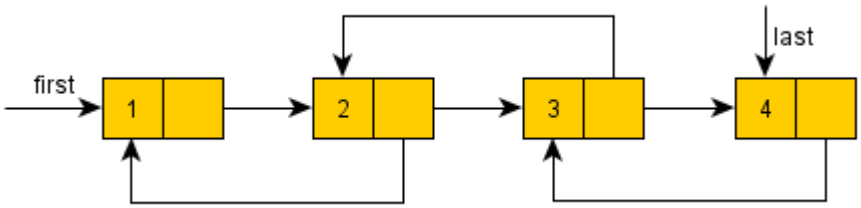
Varianty spojového seznamu

Jednosměrně zřetězený seznam
je základní variantou této datové struktury, ve které jednotlivé uzly obsahují kromě dat pouze ukazatel na další uzel (poslední uzel ukazuje na null), čímž umožňují pouze traverzování jedním směrem. Samotná struktura seznamu pak obsahuje pouze ukazatel na první prvek a data jsou přidávána vždy na začátek seznamu. Dílčím vylepšením je přidání ukazatele také na poslední prvek a umisťování nových prvků na konec seznamu.



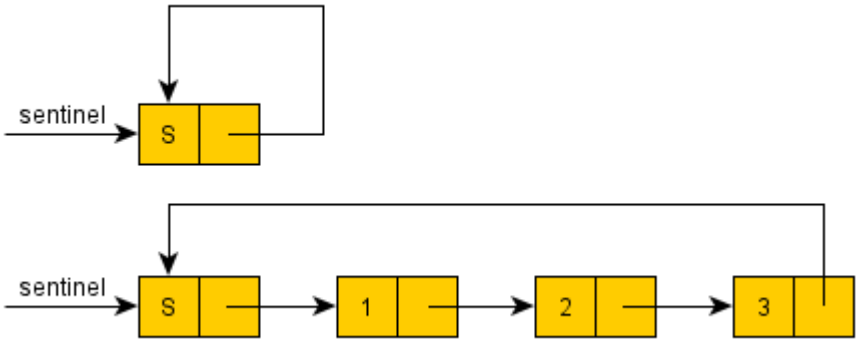
Obousměrně zřetězený spojový seznam

V obousměrně zřetězeném spojovém seznamu prvky obsahují nejen ukazatel na další prvek, ale také ukazatel na předchozí prvek. Tímto se sice poněkud zkomplikuje implementace struktury, ale toto je vyváženo zvýšenou flexibilitou, protože lze traverzovat v obou směrech.



Kruhový spojový seznam

Velmi oblíbeným trikem zjednodušujícím implementaci spojového seznamu je kruhový spojový seznam s hlídkou (sentinel). V této implementaci struktura seznamu obsahuje vždy ukazatel na objekt hlídky – speciálního uzlu, který slouží zároveň jako první a jako poslední prvek (v případě prázdného seznamu ukazuje sám na sebe). Tímto trikem dojde ke ztotožnění operací vložení na začátek, konec a mezi prvky spojového seznamu. Nevýhodou tohoto přístupu jsou dodatečné paměťové nároky na objekt hlídky.



Zjednodušení vyhledávání

Vyhledávání ve spojovém seznamu můžeme zjednodušit použitím zarážky – zarážka je uzel zvláštního typu umístěný na konec seznamu (v případě použití hlídky bude hlídka zarážkou). Do zarážky vždy umístíme data, která vyhledáváme, čímž si zajistíme, že je také najdeme. Tímto trikem eliminujeme nutnost neustálé kontroly toho, jestli již nejsme na konci seznamu.

Složitost základních operací

Operace vkládání na začátek seznamu proběhne s asymptotickou složitostí O(1), protože se pouze zalokuje nový prvek, který se vloží na počátek seznamu. Operace mazání a čtení prvku na indexu i proběhnou v čase O(i), protože spojový seznam neumožňuje náhodný přístup - k prvku je zapotřebí doiterovat.

```
Jednosmerne zretezeny spojovy seznam (bez zarazky)
public class LinkedList {
    private Node first;
    private Node last;
    private int size;

    public LinkedList() {
    this.size = 0;
}

Vlozi prvek na konec seznamu
      @param i prvek k vlozeni
public void insert(int i) {
    Node n = new Node(i);
    if (size == 0) {
        this.first = n;
        this.last = n;
        } else {
    this.last.next = n;
    this.last = n;
    }
    size++;
}

Vrati prvek na indexu i
      @return prvek na indexu i
public int read(int i) {
    if (i >= size) {
        throw new IndexOutOfBoundsException("Mimo meze index:" + i + ",
        size:" + this.size);
    }
    if (i < 0) {
        throw new IllegalArgumentException("Index mensi nez 0");
    }
    Node curr = first;
    for (int j = 0; j < i; j++) {
        curr = curr.next;
    }
    return curr.value;
}

Delka seznamu
      @return delka seznamu
public int size() {
      return this.size;
}




Smaze prvek na indexu i
      @param i index mazaneho prvku
public void delete(int i) {
    if (i >= size) {
        throw new IndexOutOfBoundsException("Mimo meze index:" + i + ",
        size:" + this.size);
    }
    if (i < 0) {
        throw new IllegalArgumentException("Index mensi nez 0");
```

```
    }
    if (i == 0) {
        first = first.next;
    } else {
        Node curr = first;
        for (int j = 0; j < i - 1; j++) {  //najdeme predchozi
            curr = curr.next;
        }
        curr.next = curr.next.next;        //a mazany prvek vynechame
        if (i == size - 1) {               //pokud mazeme posledni
            last = curr;
        }
    }
    size--;
}
```

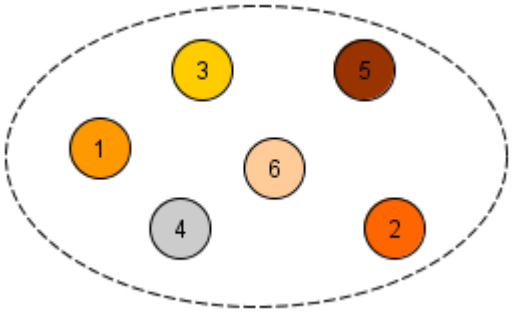
Klasicka toString metoda, vraci textovou reprezentaci objektu
@return textova reprezentace objektu

```
@Override
public String toString() {
    StringBuilder builder = new StringBuilder();
    Node curr = first;
    for (int i = 0; i < this.size; i++) {
        builder.append(curr.value).append(" ");
        curr = curr.next;
    }
    return builder.toString();
}
```

Vnitrni trida reprezentujici uzel spojoveho seznamu

```
private class Node {
    private int value;
    private Node next;

    private Node(int value) {
        this.value = value;
    }
}
```

Množina



- obsahuje hodnoty, aniž by garantovala jejich pořadí
- každý prvek obsahuje vždy maximálně jednou, to je nutno ohlídat zejména při vkládání
- pro vyhledávání je nejlepší vyhledávací strom (hashovací tabulka)

Implementace

- polem
- seznamem – nejjednodušší implementace
- bitovou mapou (polem) – musí být předem známá velikost univerza

Pole

uspořádané pole – složitost O(m log (n)) – logaritmicky vyhledáváme ve větší množině

M1 = 1, 2, 4, 6
M2 = 0, 2, 5, 6

lze udělat i v čase O(m + n)

- pomocí „ukazovátka“ procházíme střídavě obě množiny a porovnáváme prvky
- prvek z M1 vezmu a vyhodnotím a posunu ukazovátko
- pak stejně s prvkem M2
- pokud jsou prvky stejné, přesuneme do průniku a posuneme ukazovátko v obou množinách
- takto je zaručeno, že každý prvek hodnotíme pouze jednou

Spojový seznam

M1 = 6, 1, 4, 2
M2 = 0, 6, 5, 2

průnik – porovnáme prvek po prvku a ty, co jsou v obou množinách vložíme do průniku
složitost O(m * n)
uspořádaný seznam – stejná složitost

Bitová mapa

0	**1**	**2**	**3**	**4**	**5**	**6**
0	**1**	**1**	**0**	**1**	**0**	**1**

reprezentuje bitově množinu 1, 2, 4, 5, 6

sloučení množin	- součet	bitový OR
	- průnik	bitové AND
	- rozdíl	jako standardní rozdíl, jen 0 – 1 = 0

test na podmnožinu – nesmí nastat situace, že množina = 0 a podmnožina = 1

Složitost

- při vyhledání prvku nutné v krajním případě projít celou množinu, tj. složitost O(n) s ohledem na délku iniverza

Implementace množiny

```
* Mnozina implementovana jako list (seznam)
* @param <ENTITY> typovy parametr obsahu
public class Set<ENTITY> {
     private List<ENTITY> list;

* Konstruktor
* @param initialCapacity kapacita, se kterou je inicializovan podrizeny
     seznam
public Set(int initialCapacity) {
     list = new ArrayList<ENTITY>(initialCapacity);
}
```

```
* Vlozi entitu do mnoziny
* @param e entita
* @return true - pokud probehne vlozeni uspesne, false pokud jiz mnozina
     danou entitu obsahuje (dle equals())
public boolean put(ENTITY e) {
     if (list.contains(e)) return false;
     list.add(e);
     return true;
}

* Vrati nejaky prvek z mnoziny (a z mnoziny jej take odstrani)
* @return prvek, null pokud je mnozina prazdna
public ENTITY pick() {
     if(list.size() == 0) return null;
     return list.remove(list.size() - 1);
}

* Odstrani z mnoziny danou entitu
* @param e entita
* @return true pokud mnozina obsahovala danou entitu, false pokud nikoliv
public boolean remove(ENTITY e) {
     return list.remove(e);
}
* Dotaz na pritomnost dane entity
* @param e entita
* @return true - pokud jej mnozina obsahuje, false - pokud nikoliv
public boolean contains(ENTITY e) {
     return list.contains(e);}

* Dotaz na velikost mnoziny
* @return pocet prvku
public int size() {
     return list.size();}
```