Структури от данни. Стек, опашка, списък, дърво. Основни операции върху тях. Реализация.

Структурата от данни е „място", където се съхраняват обекти. Сама по себе си структурата е съставен обект, който съдържа други обекти – прости или съставни. СД могат да бъдат *статични* или *динамични*. Статични са тези СД, които не променят параметрите на вътрешното си представяне. Т.е. имат фиксирана големина и заемат фиксирано количество памет във всеки един момент от времето. Динамични са СД, които променят вътрешната си архитектура. Също така СД могат да бъдат *линейни*, когато когато елементите ѝ образуват последователност или линеен списък. *Обобщени* СД са такива, които могат да съдържат всякакъв тип данни, т.е. всички елементи са от един тип, но той може да е произволен. Докато *хетерогенни* са такива СД, чиито елементи могат да бъдат разнообразни. Структурите от данни също така могат да бъдат рекурсивни – ако са частично съставени от по-малки или по-прости инстанции на същата структура от данни. Например едно дърво се състои от по-малки дървета (поддървета) и корен.
Една СД трябва да съдържа определени класове операции:

- Конструктори – създават конкретна СД. Обикновено името на конструктора започва с *create* :
  List createEmptyList();
- Деструктори
  Clear();
  removeAll();
- Инспектори – служат за инспектиране на определени свойства на структурата
  Boolean isEmpty(); //Този и следващия пример са за инспектори
  Int size(); //към цялата БД

  Boolean contains(Object value); //Това са примери за инспектори
  Int indexOf(Object value); //към отделен елемент в БД
- Модификатори – модифицират архитектурата на самата структура
  Add, remove, insert
- Итератори – извършват операции върху цяла СД. Т.е. те обхождат всеки един елемент от структурата, като порграмистът може да модифицира и да получава всеки един елемент от нея. Итераторите се използват за постигане на абстракция, тъй като те са универсални – не се интересуват какъв тип са елементите в последователността, върху която итераторите работят.
  printAll(); toString();

Стекът е линейна статична СД. Той е крайна редица от елементи от един и същ тип. Операциите включване и изключвне на елементи от стека са разрешени само за единия му край, който се нарича *връх на стека*. Възможен е достъп само до елемента, намиращ се на върха на стека, като достъпът е пряк. При тази организация на логическите операции последния включен елемент се изключва пръв. Т.е. казваме, че логическата организация на достъп на стека е *FILO – First In Last Out*.

Реализация на стек с масив:

```
Interface Store{ // интерфейс за основните функции на всяка СД
Void clear();
Boolean isEmpty();
Int size();
```

```
}
Interface Stack extends Store { // спецефичен интерфейс за класа стек
Void push(Object item);
Void pop();
Object top();
}

Class StackArray implements Stack { // реализация на стек с масив
Private int[] stack;
Private int sp;

        StackArray() {
        Stack = new int[10];
        Sp = -1;
}

StackArray(int size) {
        Stack = new int[size];
        Sp = -1;
}

        Boolean isEmpty() {
        Return sp==-1;
}

        Void clear() {
        Sp = -1;
}

        Int size() {
        Return sp+1;
}

        Void push(int item) {
        If(sp = (stack.length-1)) {
                Throw new StackOverFlow ();
        }
        Else {
                Sp++;
                Stack[sp] = item;
        }
}

Int top() {
        If (stack.isEmpty()) {
                Throw new StackUnderFlow();
        }
        Else return stack[sp];
}

Void pop() {
        If (stack.isEmpty()) {
                Throw new StackUnderFlow();
```

```
        }
        Else sp = sp - 1;
}
```

Опашката е линейна статична СД. Тя е крайна редица от елементи от един и същ тип. Операцията включване е допустима за елементите от единия край на редицата, който се нарича *край на опашката*, а операцията изключване на елемент – от другия край на редицата, който се нарича *начало на опашката*. Възможен е достъп само до елемента, намиращ се в началото на опашката, като достъпът е пряк. При тази организация на логическите операции първия включен елемент се изключва пръв. Т.е. казваме, че логическата организация на достъп на опашката е *FIFO – First In First Out*.

Реализация на опашка с масив:

```
Public interface Linear extends Store {
Public void add(Object value);
Public Object get();
Public Object remove();
}

Public interface Queue extends Linear {
Public void enqueue(Object item);
Public Object dequeue();
Public Object peek();
}

Public abstract class AbstractQueue implements Queue {
Public void enqueue(Object item) { add(item) }
Public Object dequeue() { return remove() }
Public Object peek() {return get() }
}

Public class QueueArray extends AbstractQueue implements Queue {
Protected Object[] data;
Protected int head;
Protected int count;

QueueArray(int size) {
        Data = new Object[size];
        Head = 0;
        Count = 0;
}

Public void add(Object value) {
        Int tail = (head + count)%data.length();
        Data[tail] = value;
        Count ++;
}

Public Object remove() {
        Object value = data[head];
        Count --;
        Return value;
```

```
}

Public Object get() {
        Return data[head];
}

Public int size() {
        Return count;
}

Boolean isFull() {
        Return count == data.length();
}

Boolean isEmpty() {
        Return count == 0;
}
```

При стековете и опашките единственият начин за извлчане на данни от структурата се осъществява чрез остраняване на елементи. Често се налага да се използват всички данни от редица, без да се отстраняват. За целта се използва структурата **свързан списък**. Свързаният списък е крайна редица от елементи от един и същ тип. Операциите включване и изключване са допустими на произволо място на редицата. Възможен е пряк достъп до елемена в единия край на редицата, наречен начало на списъка, и последователен достъп до всеки от останалите елементи. Има два основни начина за представяне на свързания списък в паметта на компютъра: с една или две връзки. Т.е. елементите имат достъп или само до съседа си от едната страна, или до двамата си съседи. В Java реализацията на масив е динамична. Т.е. струкрурата автоматично увеличава размера си при добавяне на нов елемент.

Реализация на свързан списък:

```
package list;

import java.util.Enumeration;

interface List extends Linear {
   public void addFirst(Object value);
   public addLast(Object value);
   public Object getFirst();
   public Object getLast()
   public Object removeFirst();
   public Object removeLast();
   public boolean contains(Object value);
   public int indexOf(Object value);
   public int lastIndexOf(Object value);
   public Object get(int index);
   public Object set(int index, Object value);
   public Object add(int index, Object value);
   public Object remove(int index);
}
```

```
public abstract class AbstractList implements List {
   public boolean isEmpty() {
      return size()==0;
   }

   public void addFirst(Object value) {
      add(0, value);
   }

   public void addLast(Object value) {
      add(size(), value);
   }

   public Object getFirst() {
      return get(0);
   }

   public Object getLast() {
      return get(size() - 1);
   }

   public Object removeFirst() {
      return remove(0);
   }

   public Object removeLast() {
      return remove(size() - 1);
   }

   public void add(Object value) {
      return addFirst(value);
   }

   public Object remove() {
      return removeLast();
   }

   public Object get() {
      return getLast();
   }

   public boolean contains(Object value) {
      return (-1 != indexOf(value));
   }
}



public class SinglyLinkedListElement {
   protected Object data;
   protected SinglyLinkedListElement nextElement;
```

```
    public SinglyLinkedListElement(Object v, SinglyLinkedListElement next) {
        data = v;
        nextElement = next;
    }

    public SinglyLinkedListElement(Object v) {
        this(v, null);
    }

    public void setNext(SinglyLinkedListElement next) {
        nextElement = next;
    }

    public SinglyLinkedListElement next() {
        return nextElement;
    }

    public void setValue(Object value) {
        data = value;
    }

    public Object value() {
        return data;
    }

    String toString() {
        return ("[SinglyLinkedListElement: " + value() + "]");
    }
}

// Реализация на едносвързан списък

public class SinglyLinkedList extends AbstractList {
    protected int count;
    protected SinglyLinkedListElement head;

    public SinglyLinkedList() {
        count = 0;
        head = null;
    }

    public void add(Object value) {
        addLast(value);
    }

    public void addFirst(Object value) {
        head = new SinglyLinkedListElement(value, head);
        count++;
    }

    public Object removeFirst() {
        SinglyLinkedListElement tmp = head;
        head = head.next();
```

```
    count--;
    return tmp.value();
}

public void addLast(Object value) {
    SinglyLinkedListElement tmp = new SinglyLinkedListElement(value);
    if(head!=null) {
        SinglyLinkedListElement finger = head;
        while(finger.next()!=null) {
            finger = finger.next();
        }
        finger.setNext(tmp);
    }
    else head = tmp;
    count++;
}

public Object removeLast() {
    SinglyLinkedListElement finger = head;
    SinglyLinkedListElement previous = null;
    while(finger.next()!=null) {
        previous = finger;
        finger = finger.next();
    }
    if(previous==null) head = null;
    else previous.setNext(null);
    count--;
    return finger.value();
}

public Object getLast() {
    SinglyLinkedListElement finger = head;
    while(finger.next()!=null) {
        finger = finger.next();
    }
    return finger.value();
}

public boolean contains(Object value) {
    SinglyLinkedListElement finger = head;
    while((finger!=null) && (!finger.value().equals(value))) {
        finger = finger.next();
    }
    return finger!=null;
}

public int indexOf(Object value) {
    int i = 0;
    SinglyLinkedListElement finger = head;
    while((finger!=null) && (!finger.value().equals(value))) {
        finger = finger.next();
        i++;
    }
```

```
        if(finger==null) return -1;
        else return i;
    }
}



public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}



public abstract class AbstractIterator implements Enumeration, Iterator {
    public AbstractIterator() {}
    public abstract void reset();
    public abstract boolean hasNext();
    public abstract Object get();

    public final Object value() {
        return get();
    }

    public abstract Object next();

    public void remove() throws Exception {
        throw new Exception("Не е реализиран!");
    }

    public final boolean hasMoreElements() {
        return hasNext();
    }

    public final Object nextElement() {
        return next();
    }
}



public class SinglyLinkedListIteraror extends AbstractIterator {
    protected SinglyLinkedList current;
    protected SinglyLinkedList head;

    public SinglyLinkedListIterator(SinglyLinkedList h) {
        head = h;
        reset();
    }

    public void reset() {
        current = head;
```

```
    }

    public boolean hasNext() {
      return current!=null;
    }

    public Object next() {
      Object tmp = current.value();
      current = current.next();
      return tmp;
    }

    public Object get() {
      return current.value();
    }
  }



  public class DoublyLinkedListElement {
    protected Object data;
    protected DoublyLinkedListElement nextElement;
    protected DoublyLinkedListElement prevElement;

    public DoublyLinkedListElement(bject val, DoublyLinkedListElement next,
DoublyLinkedListElement prev) {
      data = val;
      nextElement = next;
      if(nextElement!=null) nextElement.prevElement = this;
      prevElement = prev;
      if(prevElement!=null) prevElement.nextElement = this;
    }

    public DoublyLinkedListElement(Object val) {
      DoublyLinkedListElement(val, null, null);
    }

    public DoublyLinkedListElement next() {
      return nextElement;
    }

    public DoublyLinkedListElement previous() {
      return prevElement;
    }

    public Object value() {
      return data;
    }

    public void setNext(DoublyLinkedListElement next) {
      nextElement = next;
    }
```

```
    public void setPrevious(DoublyLinkedListElement prev) {
        prevElement = prev;
    }

    public void setValue(Object value) {
        data = value;
    }

    public String toString() {
        return "[DoublyLinkedListElement: " + value() + "]";
    }
}


// Реализация на двусвързан списък

public class DoublyLinkedList implements List {
    protected DoublyLinkedListElement head;
    protected DoublyLinkedListElement tail;
    protected int count;

    public DoublyLinkedList() {
        head = tail = null;
        count = 0;
    }

    public void addToHead(Object value) {
        head = new DoublyLinkedListElement(value, head, null);
        if(tail == null) tail = head;
        count++;
    }

    public void addToTail(Object value) {
        tail = new DoublyLinkedListElement(value, null, tail);
        if(head == null) head = tail;
        count++;
    }

    public Object removeFromHead() {
        if(head==null) return null;
        DoublyLinkedListElement tmp = head;
        head = head.next();
        if(head!=null) head.setPrevious(null);
        else tail = null;
        tmp.setNext(null);
        count--;
        return tmp.value();
    }

    ...
}
```

```
public class DoublyLinkedListIterator extends AbstractIterator {
   protected DoublyLinkedListElement head;
   protected DoublyLinkedListElement tail;
   protected DoublyLinkedListElement current;

   public DoublyLinkedListIterator(DoublyLinkedListElement head, DoublyLinkedListElement tail) {
      this.head = head;
      this.tail = tail;
      reset();
   }

   public boolean hasNext() {
      return current!=tail;
   }
}
```

Дървото е рекурсивна структура от данни, състояща се от възли, свързани помежду си. В общия случай възлите имат предшественици и наследници. Има обаче възли, които правят изключение:
Корен – възел, който няма предшественик
Листо – възел без наследници.
Всички вътрешни възли в едно дърво имат наследници и предшественици.
Двоичното кореново дърво се състои от:

- Корен
- Ляво поддърво
- Дясно поддърво

Възможно е лявото или дясното поддърво да е празно. При двоичното дърво един възел има най-много двама наследника (ляв и десен) и точно един предшественик (с изключение на корена).
На всеки връх в дървото може да се съпостави **ниво**. Коренът на дървото има ниво 1 (или 0). Ако един връх има ниво *i*, то неговите наследници имат ниво *i*+1. С други думи, нивото на един връх е броят на върховете, които трябва да бъдат обходени за да се стигне до този връх от корена. Максималното ниво на едно дърво се нарича негова **височина**. Над структурата от данни двоично дърво са възможни следните операции:

- достъп до връх – възможен е пряк достъп до корена и непряк достъп до останалите върхове;
- включване и изключване на връх – възможни са на произволно място в двоичното дърво, резултатът трябва отново да е двоично дърво;
- обхождане – това е метод, позволяващ да се осъществи достъп до всеки връх на дървото един единствен път.

Обхождането е рекурсивна процедура, която се осъществява чрез изпълнение на следните три действия, в някакъв фиксиран ред:

- обхождане на корена;
- обхождане на лявото поддърво;
- обхождане на дясното поддърво.

Има различни видове обхождания. Най-разпространени са няколко. **Смесеното обхождане** (inOrder) – при него се обхожда първо лявото поддърво, после корена и най-накрая дясното поддърво. При **низходящото обхождане** (preOrder) се обхожда първо корена, после лявото поддърво и най-накрая дясното поддърво (или корен, дясно поддърво, ляво поддърво –

зависи от реализацията). **Възходящо обхождане** (postOrder) - първо лявото поддърво, после дясното поддърво и най-накрая корена. **Обхождане по нива** (levelOrder) – дървото се обхожда по нива, като се почва от корена.

Реализация на елемент от двоичното дърво / възел:

```
public class BinaryTreeNode {
   protected Object val;
   protected BinaryTreeNode parent;
   protected BinaryTreeNode left;
   protected BinaryTreeNode right;
   public static final BinaryTreeNode EMPTY = new BinaryTreeNode();

   private BinaryTreeNode() {
      val = null;
      parent = left = right = EMPTY;
   }

   public BinaryTreeNode(Object value) {
      val = value;
      parent = left = right = null;
   }

   public BinaryTreeNode(Object val, BinaryTreeNode left, BinaryTreeNode right) {
      this(val);
      setLeft(left);
      setRight(right);
   }

   public BinaryTreeNode parent() { return parent; }

   public BinaryTreeNode left() { return left; }

   public BinaryTreeNode right() { return right; }

   public Object value() { return val; }

   public void setLeft(BinaryTreeNode newLeft) {
      if(isEmpty()) { return; }
      if(left.parent() == this) {
         left.setParent(null);
      }
      left = newLeft;
      left.setParent(this);
   }
   // Аналогична е функцията setRight!
   public void setParent(BinaryTreeNode newParent) {
      parent = newParent;
   }

   public boolean isEmpty() { return this.equals(EMPTY); }

   public int size() {
```

```
    if(isEmpty()) { return 0; }
    return 1 + left.size() + right.size();
  }

  public int height() {
    if(isEmpty()) return -1;
    return 1 + Math.max(left.height(), right.height());
  }

  public int depth() {
    if(parent = null) return 0;
    return 1 + parent.depth();
  }

  public boolean isLeftChild() {
    if (parent = null) { return false; }
    return (this == parent().left());
  }

  public Iterator iterator() {
    return inOrderIterator();
  }

  public AbstractIterator inOrderIterator() {
    return new BinaryTreeInOrderIterator(this);
  }
}
```

Следва реализация на итератор, който служи за смесено обхождане на дърво (inOrder). Логиките на итераторите за другите обхождания са подобни. Итераторите биха се различавали в няколко функции – reset(), get(), next(), както ***todo*** структурата би била различна – опашка или др.

```
public class BinaryTreeInOrderIterator implements Iterator {
  protected BinaryTreeNode root;
  protected Stack todo;

  public BinaryTreeInOrderIterator(BinaryTreeNode root) {
    todo = new StackList();
    this.root = root;
    reset();
  }

  public void reset() {
    todo.clear();
    BinaryTreeNode current = root;
    while(current!= BinaryTreeNode.EMPTY) {
      todo.push(current);
      current = current.left();
    }
  }

  public boolean hasNext() {
```

```
    return !todo.isEmpty();
  }

  public Object get() {
    return ((BinaryTreeNode)todo.getFirst()).value();
  }

  public Object next() {
    BinaryTreeNode old = (BinaryTreeNode)todo.pop();
    Object result = old.value();
    if(!old.right().isEmpty()) {
      BinaryTreeNode current = old.right();
      do {
        todo.push(current);
        current = current.left();
      }
      while (!current.isEmpty());
    }
    return result;
  }
}
```

Има частен случай на двоично дърво, наречен двоично дърво за търсене /двоично наредено дърво. То се различава от двоичното дърво по това, че възлите му са подредени в определен ред. За всеки възел има специфичен начин, по който се определя лявото и дясното му поддърво – всички възли, намиращи се в лявото му поддърво имат по-малка стойност от неговата, а вички възли в дясното поддърво – по-голяма или равна стойност. Това определя по-голямата сложност на реализацията на двоично дърво за търсене, сравнена с тази на обикновеното двоично дърво, тъй като всеки път при добавяне или премахване на елемент от дървото ще трябва да се грижим условието за възлите да не се наруши. Следва реализация на двоично наредено дърво. Специфичните функции като премахване и добавяне на възел на двоично дърво са по-опростени, като зависят от реализацията – примерно винаги се добавят елементи в лявото поддърво на даден възел и при премахване на възел поддърветата му се закачат за най-лявото листо. При двоичното наредено дърво търсенето на даден възел е опростено, тъй като може да се сравнят стойностите на търсения възел и текущия и съответно да се продължи търсенето само в лявото или дясното поддърво, докато при нормалното двоично дърво ще трябва да се обходят всички възли, тъй като няма наредба.

```
public class BinarySearchTree implements Store {
  protected BinaryTreeNode root;
  protected int count;
  protected Comparator ordering;
  public BinarySearchTree() {
    this(new NaturalComparator());
  }

  public BinarySearchTree(Comparator ordering) {
    root = BinaryTreeNode.EMPTY;
    count = 0;
    this.ordering = ordering;
  }

  public boolean isEmpty() {
```

```
    return root.isEmpty();
}

public int size() {
    return count;
}

public void clear() {
    root = BinaryTreeNode.EMPTY;
    count = 0;
}

protected BinaryTreeNode locate(BinaryTreeNode root, Object value) {
    Object rootValue = root.value();
    BinaryTreeNode child;
    if(rootValue.equals(value)) { return root; }
    if(ordering.compare(rootValue, value) < 0) {
        child = root.right();
    }
    else { child = root.left(); }
    if(child.isEmpty()) { return root; }
    else { return locate(child, value); }
}

public boolean contains(Object value) {
    if(root.isEmpty()) { return false; }
    BinaryTreeNode possibleLocation = locate(root, value);
    return value.equals(possibleLocation.value());
}

public void add(Object value) {
    BinaryTreeNode newNode = new BinaryTreeNode(value);
    if(root.isEmpty()) { root = newNode; }
    else {
        BinaryTreeNode location = locate(root, value);
        Object nodeValue = location.value();
        if(ordering.compare(nodeValue, value) > 0) {
            location.setLeft(newNode);
        }
        else {
            if(!location.right().isEmpty()) {
                predecessor(location).setLeft(newNode);
            }
            else {
                location.setRight(newNode);
            }
        }
    }
    count ++;
}

protected BinaryTreeNode predecessor (BinaryTreeNode root) {
    BinaryTreeNode result = root.left();
```

```
        while(!result.right().isEmpty()) {
            result = result.right();
        }
        return result;
    }

    protected BinaryTreeNode removeTop(BinaryTreeNode topNode) {
        BinaryTreeNode left = topNode.left();
        BinaryTreeNode right = topNode.right();
        topNode.setLeft(BinaryTreeNode.EMPTY);
        topNode.setRight(BinaryTreeNode.EMPTY);
        if(left.isEmpty()) { return right; }
        if(right.isEmpty()) { return left; }
        BinaryTreeNode predecessor = left.right();
        if(predecessor.isEmpty()) {
            left.setRight(right);
            return left;
        }
        BinaryTreeNode parent = left;
        while(!predecessor.right().isEmpty()) {
            parent = predecessor;
            predecessor = predecessor.right();
        }
        parent.setRight(predecessor.left());
        predecessor.setLeft(left);
        predecessor.setRight(right);
        return predecessor;
    }

    public Object remove(Object value) {
        if(isEmpty()) return null;
        if(value.equals(root.value())) {
            BinaryTreeNode newRoot = removeTop(root);
            count--;
            Object result = root.value();
            root = newRoot;
            return result;
        }
        else {
            BinaryTreeNode location = locate(root, value);
            if(value.equals(location.value())) {
                count--;
                BinaryTreeNode parent = location.parent();
                if(parent.right()==location) parent.setRight(removeTop(location));
                else parent.setLeft(removeTop(location));
                return location.value();
            }
        }
        return null;
    }
}
```