

Laboration 7 – binärt träd, heap och prioritetsskö

Packa upp filerna i **L7_kod.zip** (från kurssidan) i paketet **laboration11**.

Grundläggande uppgifter

1. I metoden ***exercise1*** i **Laboration11.java** skapas ett binärt träd. Rita på ett papper hur trädet som skapas ser ut.

Kontrollera sedan ditt resultat genom att aktivera sista raden i uppgift1():

```
tree.showTree();
```

och anropa ***exercise1*** från main-metoden.

2. I metoden ***exercise2*** i **Laboration11.java** skapas ett binärt träd. Rita på ett papper hur trädet som skapas ser ut.

Kontrollera sedan ditt resultat genom att aktivera sista raden i uppgift1():

```
tree.showTree();
```

och anropa ***exercise2*** från main-metoden.

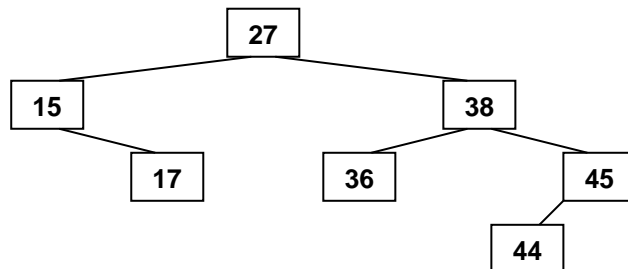
3. I metoden ***exercise3*** i **Laboration11.java** skapas ett binärt träd. Rita på ett papper hur trädet som skapas ser ut.

Kontrollera sedan ditt resultat genom att aktivera sista raden i uppgift1():

```
tree.showTree();
```

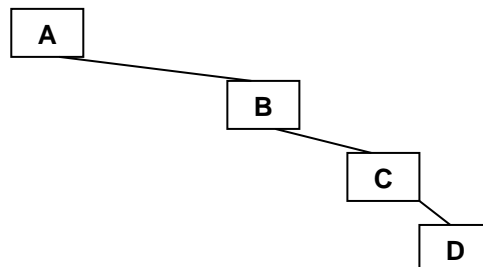
och anropa ***exercise3*** från main-metoden.

- 4a. Skapa ett binärt träd med nedanstående utseende. Placera Integer-objekt i trädet



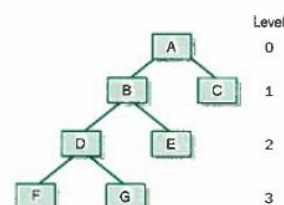
Placera din kod i metoden ***exercise4***.

- 4b. Skapa ett binärt träd med nedanstående utseende. Placera Character-objekt i trädet.



Placera din kod i metoden ***exercise5***.

- 4c. Skapa ett träd som ser ut som trädet på s 625 i läroboken.



5. Komplettera klassen *BTNode* med metoden *search*

public BTNode<E> search(E value)

vilken ska söka efter ett objekt i trädet och returnera en referens till noden som lagrar objektet. Om objektet inte finns i trädet ska *null* returneras.

Du kan följa nedanstående algoritm när du löser problemet:

Deklarera variabeln *node* av typen *BTNode<E>* och initiera variabelns värde till *null*.

Om *value* och *this.value* är samma (använd *equals*-metoden)

tilldela *node* värdet av *this*

Om (*node == null*) och (*this.left != null*) // ej funnet och går det att leta till vänster?

tilldela *node* värdet av det rekursiva anropet *this.left.search(value);*

Om (*node == null*) och (*this.right != null*) // ej funnet och går det att leta till höger?

tilldela *node* värdet av det rekursiva anropet *this.right.search(value);*

Returnera *node*

Testa din lösning genom att lägga till anrop till *search*-metoden i slutet av metoden

exercise1 i Laboration11:

```
BTNode<Integer> res = tree.search( new Integer(108) );
if( res!= null) {
    System.out.println( "Finns" );
} else {
    System.out.println( "Finns ej" );
}
```

Testa med olika värden så du försäkrar dig om att din lösning är bra.

6. Rita upp trädet i exercise 1, exekvera anropet *tree.search(new Integer(108))* med debuggerna och markera i trädet ordningen för exekveringen.

7a. Exekvera *main*-metoden i klassen *ArrayHeap*. Som du ser ordnas *Integer*-objekten i heapen så att lägst värde tas ur heapen först. Klassen *Integer* implementerar *Comparable* på så sätt att lägst värde ska ordnas först.

Testa även den avmarkerade konstruktorn och studera resultatet. Hur fungerar *ReverseComparable*-objekt vid jämförelser?

7b. Komplettera klassen *ArrayHeap<E>* med konstruktorn **public ArrayHeap(E[] elements)** och med metoden *heapify()*. Konstruktorn är given nedan och information/algoritm för *heapify* finner du under konstruktorn.

```
public ArrayHeap(E[] elements) {
    this.list = elements;
    size = list.length;
    comp = new Comp();
    heapify();
}
```

Metoden *heapify* ska bygga heapen från botten, dvs. successivt skapa allt större subträd som var och ett utgör en heap. Vi tänker oss att elementen i arrayen lagras i ett binärt träd och nu ska vi se till att elementen uppfyller kraven för en heap.

Algoritm för heapify():

Lagra positionen för den sista föräldern i trädet i variabeln *parent (int)*. Den sista föräldern är i position (*size - 2*) / 2.

Medan föräldrarnas position är större eller lika med 0 så

Anropa metoden *siftDown* med *parent* som argument

Minska *parent* med 1

När du är färdig ska du skapa en array (t.ex. en *Integer*-array) och testa konstruktorn.

- 7c. Skriv **klassmetoden** `public static <E> void sort(E[] elements)` i klassen `ArrayHeap`. Metoden ska ordna objekten i arrayen `elements` växande genom heapsort.

Du kan följa denna algoritm:

Skapa en `ArrayHeap` med `elements` som argument.

Medan det finns fler element i heapen så

 fyll arrayen `elements` från slutet (bakifrån) med elementen i heapen

Hur fungerar sorteringen? Fundera och googla.

- 7d. Det tråkiga med lösningen i uppgift 7c är att elementen ordnas avtagande. Nu ska du lägga till konstruktorn `public ArrayHeap(E[] elements, Comparator<E> comparator)` i klassen `ArrayHeap`.

```
public ArrayHeap(E[] elements, Comparator<E> comparator) {  
    this.list = elements;  
    size = list.length;  
    comp = comparator;  
    heapify();  
}
```

Nu ska **sort**-metoden ändras så att den nya konstruktorn används. På så sätt kan vi som andra argument ge ett `Comparator`-objekt som ordnar elementen avtagande. Och på så sätt kommer sorteringen bli växande.

```
public static <E> void sort(E[] arr) {  
    ArrayHeap<E> heap = new ArrayHeap<E>(arr, new ReverseComparable<E>());  
    for(int i=arr.length-1; i>0; i--)  
        arr[i] = heap.delete();  
}
```

Testa den nya versionen av sort-metoden. Hur ordnas objekten?

Studera därefter **sort**-metoden. På den första raden används ett objekt av typen

`ReverseComparable` som argument. Studera klassen och kontrollera vad det är för värde som returneras i `compare`-metoden.

8. Klassen **PriorityQueue** är given. Klassen använder en `ArrayHeap` för att lagra elementen i kön. Elementen lagras inte direkt i heapen utan det är **PriorityQueueElement**-objekt som lagras i heapen. Ett sådant objekt kan lagra ett element av typen `E` och har dessutom instansvariabeln `order` vilket anger ordningen som element placerats på heapen. Element med lägre värde i `order` placerades tidigare i heapen än element med högre värde.

Klassen har tre konstruktorer:

- `public PriorityQueue()` och `public PriorityQueue(int initialCapacity)`
Elementens prioritet ges av elementens `Comparable`-implementering. Om två element är lika så avgör `order` vilket element som ska vara först i heapen.
- `public PriorityQueue(int initialCapacity, Comparator<E> comparator)`
Elementens prioritet ges av parametern `comparator`. Om två element är lika så avgör `order` vilket element som ska vara först i heapen.

Din uppgift är att färdigställa klassen **PriorityQueue**, dvs. metoderna `insert`, `delete`, `peek`, `empty` och `size`.

När klassen är färdig kan du köra programmet `TestPriorityQueue`. `main`-metoden innehåller fyra exempel på konstruktion av ett `PriorityQueue`-objekt. Testa dem en åt gången.

Fördjupande uppgifter

9. Komplettera klassen *BTNode* med metoden ***collectPreorder*** vilken ska lägga till samtliga värde-objekt som lagras i trädet i en *ArrayList*. Metoden ska deklarerars så här:

public void collectPreorder(ArrayList<E> list)

Till din hjälp har du metoden *printPreorder* i klassen *BTNode*. Men nu ska inte värdet skrivas ut utan placeras i *ArrayList*-objektet. Använd din egen *ArrayList* (P1 – *collections*). Om du inte har en egen *ArrayList* kan du importera *java.util.ArrayList*.

Om du lägger till nedanstående rader sist i metoden ***exercise1*** i **Laboration11** ska du få utskriften: 102 74 63 100 110 108

```
ArrayList<Integer> list = new ArrayList<Integer>();  
tree.collectPreorder(list);  
for(int i=0; i<list.size(); i++) {  
    System.out.print(list.get(i) + " ");  
}
```

10. Komplettera klassen *BTNode* med metoden ***collectLevelOrder*** vilken ska lägga till samtliga värden som lagras i trädet i en *LinkedList*. Metoden ska deklarerars så här:

public void collectLevelOrder(LinkedList<E> list).

Använd din egen *LinkedList* (P1 – *collections*). Om du inte har en egen *LinkedList* kan du importera *java.util.LinkedList*.

Om du testar metoden med trädet i *exercise1* ska du få utskriften:

102 74 110 63 100 108

11. Interfacet *Filter* är givet:

```
public interface Filter<E> {  
    public boolean accept( E element );  
}
```

Komplettera klassen *BTNode* med metoden

public void collect(ArrayList<E> list, Filter<E> filter)

vilken ska traversera hela trädet. Traverseringen ska ske *inorder*.

Under traverseringen ska varje värde-objekt vara argument vid anrop till *accept*-metoden i *Filter*-implementeringen. Om *accept*-metoden returnerar *true* ska värde-objektet läggas till i *ArrayList*-objektet.

Om du testar din lösning med nedanstående kod (se till att *IntervalFilter* finns i paketet

laboration11) så ska du få utskriften: 63 74 100 102 108 110

Samtliga värden är ju i intervallet 0-130 och samlandet sker *inorder*.

```
ArrayList<Integer> list = new ArrayList<Integer>();  
tree.collect(list, new IntervalFilter(0,130));  
for(int i=0; i<list.size(); i++) {  
    System.out.print(list.get(i) + " ");  
}
```

Om du ändrar argumenten när *IntervalFilter*-objektet skapas till 70 resp 102 kommer utskriften bli: 74 100 102

12. Skriv en *Filter<Integer>*-implementering, ***EvenNumbers***, vilken returnerar *true* för alla *Integer*-objekt som är delbara med 2 (resten 0 vid division med 2).

Extra

13. Komplettera klassen *BTNode* med metoden
public ArrayList<E> collect(Filter<E> filter)
vilken ska fungera som metoden i uppgift 11 men returnera en *ArrayList* med de värde-objekt för vilka filter-metoden returnerar *true*.
14. Interfacet ***Iterator*** används ofta då en användare ska kunna traversera en datastruktur:

```
public interface Iterator<E> {  
    public boolean hasNext(); // returnera true om det finns fler element att besöka  
    public E next(); // returnerar referens till elementet som står på tur att besökas  
    public void remove(); // tar bort aktuellt element ur datastrukturen. Din variant  
                           // ska kasta ett UnsupportedOperationException  
}
```

Komplettera klassen *BTNode* med metoden
public Iterator<E> iterator(Filter<E> filter)
vilken ska samla de värde-objekt för vilka filter-metoden returnerar *true* i en *Iterator*-implementering.

En lösning: Skriv en klass vilken

* implementerar *Iterator<E>*

* tar en *ArrayList<E>* som argument vid konstruktion. Det är elementen i *ArrayList*-objektet som levereras med *next*-metoden.

Sedan använder du lösningen i Uppgift 13 kombinerat med denna klass i din lösning.

15. Komplettera klassen *BTNode* med metoden
public Iterator<E> iterator()
vilken ska returnera samtliga element i trädet i en *Iterator*-implementering
16. En klass vilken implementerar interfacet ***Iterable***:

```
public interface Iterable<E> {  
    public Iterator<E> iterator()  
}
```

kan användas tillsammans med den förenklade for-loopen. Klassen *BTNode* innehåller metoden *iterator* om du löst uppgift 15. Låt klassen *BTNode* implementera *Iterable<E>* och testa sedan den förenklade for-loopen i *exercise1*:

```
for(Integer nbr : tree) {  
    System.out.print(nbr + " ");  
}
```

Lösningar

Uppgift 4a

T.ex.

```
public void exercise4a() {
    BTNode<Integer> n7 = new BTNode<Integer>( new Integer(17), null, null );
    BTNode<Integer> n6 = new BTNode<Integer>( new Integer(36), null, null );
    BTNode<Integer> n5 = new BTNode<Integer>( new Integer(44), null, null );
    BTNode<Integer> n4 = new BTNode<Integer>( new Integer(15), null, n7 );
    BTNode<Integer> n3 = new BTNode<Integer>( new Integer(45), n5, null );
    BTNode<Integer> n2 = new BTNode<Integer>( new Integer(38), n6, n3 );
    BTNode<Integer> tree = new BTNode<Integer>( new Integer(27), n4, n2 );
}
```

Uppgift 4b

T.ex.

```
public void exercise4b() {
    BTNode<Character> n1 = new BTNode<Character>(new Character('D'),null,null);
    BTNode<Character> n2 = new BTNode<Character>(new Character('C'),null,n1);
    BTNode<Character> n3 = new BTNode<Character>(new Character('B'),null,n2);
    BTNode<Character> tree = new BTNode<Character>(new Character('A'),null,n3);
}
```

Uppgift 4c

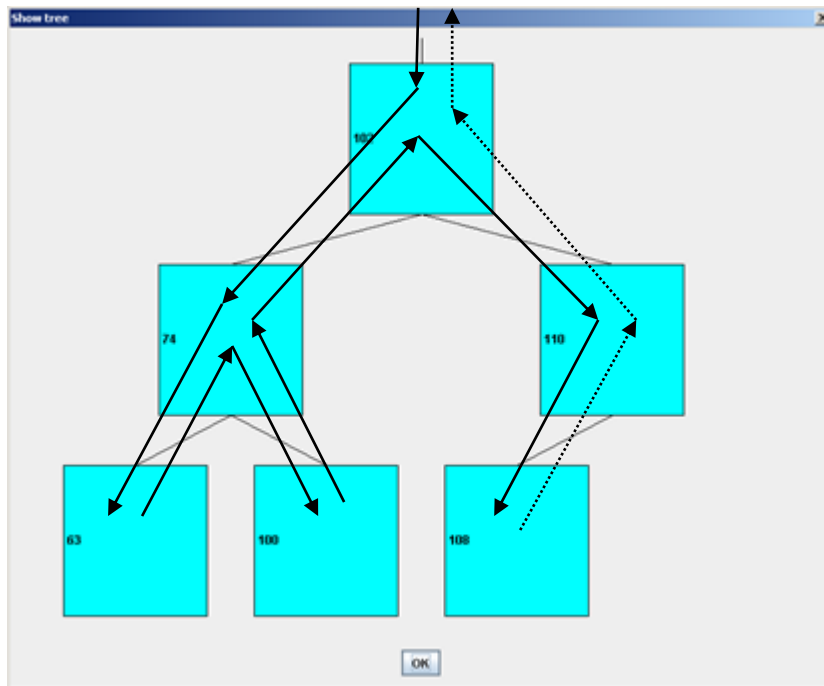
T.ex.

```
public void exercise4c() {
    BTNode<Character> n1 = new BTNode<Character>(new Character('F'),null,null);
    BTNode<Character> n2 = new BTNode<Character>(new Character('G'),null,null);
    BTNode<Character> n3 = new BTNode<Character>(new Character('E'),null,null);
    BTNode<Character> n4 = new BTNode<Character>(new Character('C'),null,null);
    BTNode<Character> n5 = new BTNode<Character>(new Character('D'),n1,n2);
    BTNode<Character> n6 = new BTNode<Character>(new Character('B'),n5,n3);
    BTNode<Character> tree = new BTNode<Character>(new Character('A'),n6,n4);
}
```

Uppgift 5

```
public BTNode<E> search(E value) {
    BTNode<E> node = null;
    if( value.equals(this.value))
        node = this;
    if( (node==null) && (this.left!=null) )
        node = this.left.search( value );
    if( (node==null) && (this.right!=null) )
        node = this.right.search( value );
    return node;
}
```

Uppgift 6



Uppgift 7b

```
public ArrayHeap(E[] list) {
    this.list = list;
    size = list.length;
    comp = new Comp();
    heapify();
}

private void heapify() {
    int parent = (size-2)/2; // last parent
    while(parent >= 0) {
        siftDown(parent);
        parent--;
    }
}
```

Uppgift 7c

```
public static <E> void sort(E[] elements) {
    ArrayHeap<E> heap = new ArrayHeap<E>(elements);
    for(int i=elements.length-1; i>0; i--)
        elements[i] = heap.delete();
}
```

Uppgift 7d

```
public static <E> void sort(E[] arr) {
    ArrayHeap<E> heap = new ArrayHeap<E>(arr, new ReverseComparable<E>());
    for(int i=arr.length-1; i>0; i--)
        arr[i] = heap.delete();
}
```

Uppgift 8

```
public class PriorityQueue<E> implements Queue<E>{
    private ArrayHeap<PriorityQueueElement<E>> heap;

    public PriorityQueue() {
        this(20);
    }

    public PriorityQueue(int initialCapacity) {
        heap = new ArrayHeap<PriorityQueueElement<E>>(initialCapacity);
    }

    public PriorityQueue(int initialCapacity, Comparator<E> comparator) {
        heap = new ArrayHeap<PriorityQueueElement<E>>(initialCapacity, new
PriorityQueueComparator<E>(comparator));
    }

    // Lägg till ett PriorityQueueElement i heapen
    public void enqueue(E data) {
        heap.insert(new PriorityQueueElement<E>(data));
    }

    // Returnera elementet som lagras i PriorityQueueElement-objektet. Anropa
delete-metoden i ArrayHeap.
    public E dequeue() {
        return heap.delete().getElement();
    }

    // Returnera elementet som lagras i PriorityQueueElement-objektet. Anropa peek-
metoden i ArrayHeap.
    public E peek() {
        return heap.peek().getElement();
    }

    public boolean empty() {
        return size()==0;
    }

    public int size() {
        return heap.size();
    }
}
```


Uppgift 9

```
public void collectPreorder(ArrayList<E> list) {
    list.add(this.value);
    if(this.left!=null)
        this.left.collectPreorder(list);
    if(this.right!=null)
        this.right.collectPreorder(list);
}
```

----- Alternativ lösning med Action-implementering -----

```
public void collectPreorder(ArrayList<E> list) {
    preorder( new AddToArrayList<E>( list ) );
}
```

```
public class AddToArrayList<E> implements BTNode.Action<E> {
    private ArrayList<E> list;

    public AddToArrayList(ArrayList<E> list) {
        this.list = list;
    }

    public void action(E value) {
        list.add(value);
    }
}
```

----- Alternativ lösning med anonym Action-implementering -----

```
public void collectPreorder(final ArrayList<E> list) {
    Action<E> a = new Action<E>() {
        public void action(E value) {
            list.add(value);
        }
    };
    preorder(a);
}
```

Uppgift 10

```
public void collectLevelOrder(LinkedList<E> list) {
    ListQueue<BTNode<E>> queue = new ListQueue<BTNode<E>>();
    BTNode<E> node;
    queue.enqueue(this);
    while(!queue.empty()) {
        node = queue.dequeue();
        list.add(node.value);
        if(node.left!=null)
            queue.enqueue(node.left);
        if(node.right!=null)
            queue.enqueue(node.right);
    }
}
```

Uppgift 11a

```
public void collect( ArrayList<E> list, Filter<E> filter ) {
    if(this.left!=null) {
        this.left.collect(list, filter);
    }
    if(filter.accept(this.value)) {
        list.add(this.value);
    }
    if(this.right!=null) {
        this.right.collect(list, filter);
    }
}
```

Uppgift 12

```
public class EvenNumbers implements Filter<Integer> {
    public boolean accept(Integer element) {
        int nbr = element.intValue();
        return (nbr % 2) == 0;
    }
}
```

Uppgift 13

```
public ArrayList<E> collect( Filter<E> filter ) {
    ArrayList<E> list = new ArrayList<E>();
    collect(list,filter);
    return list;
}
```

Uppgift 14

```
import java.util.Iterator;

public class ArrayListIterator<E> implements Iterator<E> {
    private ArrayList<E> list;
    int index=0;

    public ArrayListIterator( ArrayList<E> list ) {
        this.list = list;
    }

    public boolean hasNext() {
        return index<list.size();
    }

    public E next() {
        return list.get(index++);
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}

// I BTreeNode
public Iterator<E> iterator( Filter<E> filter ) {
    return new ArrayListIterator<E>(collect(filter));
}
```

Uppgift 15

```
public Iterator<E> iterator() {
    return new ArrayListIterator<E>( collect( new TrueFilter() ));
}

// Inre klass i BTreeNode
private class TrueFilter implements Filter<E> {
    public boolean accept(E value) {
        return true;
    }
}
```