Algoritmer & Datastrukture Opgave 1

Introduktion

Rapporten omhandler en kort gennemgang af koden, samt hvordan visse implementeringer virker og benyttes til gennemførsel af opgaven.
Der er gennemtænkt om implementeret følgende:

- BinaryHeap indeholdende Elementer.
- BinaryTree indeholdende heltal.

Kode

Der er udeladt trivielle dele af koden inklusiv klasserne/interfaces ‘Element.java’, ‘PQ.java’, ‘Dict.java’ da disse var udleveret på forhånd.

Heapsort

Koden til at lave Heapsort er udformet på baggrund af det suedo-kode der findes i bogen, og der er ikke lavet noget specielt i forhold til hvordan vi mener en standard implementation kan fungerer. Dog er der tilføjet så Heap’en automatisk bliver gjort større hvis der bliver tilføjet for mange elementer.

ExtractMin trækker det element, med den mindste nøgleværdi ud af heapen.

1. ExtractMin trækker det første element ud af “heapen”.
2. Trækker den det sidste element hen på den første plads i listen
3. Reducere totale antal elementer i listen med 1
4. Udfører rekursivt heapifyMin for det første indeks.

```
@Override
public Element extractMin() {
    if (this.currentElements <= 0)
        return null;

    Element min = heap[0];
    heap[0] = heap[this.currentElements - 1];
    heap[currentElements - 1] = null;
    currentElements--;
    heapifyMin(0);
    return min;
}
```

Insert tager et Element som input. Element er et objekt der indeholder en nøgleværdi, samt noget tilhørende data

algoritmen Insert gør følgende:

1. Tjekker om der er brug for at kalde evolveHeap, i tilfælde af listen er ved at blive for kort
2. Indsætter elementen i indekset “currentElements”, der holder styr på hvor mange elementer der er i listen.

3. øger “currentElements” med 1
4. opretter heltals værdierne “elementIndex” til elements indeks og “parentIndex” til forældre elementets indeks
5. Udfører et loop der kører indtil elementets nøgle er større end dets forældre, hvor den skifter plads med dets forældre.

```
@Override
public void Insert(Element e) {
    if (this.currentElements >= heap.length)
        this.evolveHeap();

    heap[this.currentElements] = e;
    this.currentElements++;
    int elementIndex = (this.currentElements - 1);
    int parentIndex = (this.currentElements - 1) / 2;
    while (elementIndex > 0 && e.key < heap[parentIndex].key) {
        swap(elementIndex, parentIndex);
        elementIndex = parentIndex;
        parentIndex = (elementIndex - 1) / 2;
    }
}
```

HeapifyMin er en metode der rekursivt tjekker om børnenes nøgleværdi til en forældre er mindre end forældrenes nøgle.

1. Finder den mindste værdi af de 2 børn og forældre
    a. mindste nøgle ikke forældres, byttes pladsen med det barn med den mindste værdi
        i. Metoden kaldes med forældrenes nye indeks.
    b. Ellers er problemet løst og det rekursive kald stopper.

```
private void heapifyMin(int i) {
    int l = i * 2 + 1;
    int r = i * 2 + 2;
    int smallest = 0;
    if (l < this.currentElements && heap[l].key < heap[i].key) {
        smallest = l;
    } else {
        smallest = i;
    }
    if (r < this.currentElements && heap[r].key < heap[smallest].key) {
        smallest = r;
    }
    if (smallest != i) {
        swap(smallest, i);
        heapifyMin(smallest);
    }
}
```

Swap bytter 2 elementers plads i heap-listen.

1. opretter et backup-element for indeks i
2. sætter indeks i’s plads i listen lig med indeks j’s element
3. sætter indeks j’s plads i listen lig med backup elementet

```
private void swap(int j, int i) {
    Element holder = heap[i];
    heap[i] = heap[j];
    heap[j] = holder;
}
```

Øger heapen's kapacitet

1. Opretter en ny liste med den dobbelte kapacitet af den nuværende heapliste.
2. kopiere alle elementer fra den nuværende heapliste over i den nye liste, med samme indeks.
3. overskriver den nuværende liste med den nye liste.

```
private void evolveHeap() {
    Element[] elements = new Element[this.currentElements * 2];
    for (int i = 0; i < this.currentElements; i++) {
        elements[i] = this.heap[i];
    }
    this.heap = elements;
}
```

TreeSort

Til det binære søge træ, har vi oprettet en ny klasse som indeholder elementerne til hver node, det er i denne klasse vores implementation er lagt i forhold til at søge og tilføje.

Node.java - klassen

```
public Node left;
public Node right;
public Node parent;

public int key;

public Node(Node parent, int key){
    this.key = key;
    this.parent = parent;
}
```

Node.java klassen indeholder 4 attributter; det venstre barn, det højre barn og dets forældre. Forældren bliver ikke brugt på nuværende tidspunkt, men er tilføjet i tilfælde af den skal bruges i fremtiden. Derudover indeholder den sin faktiske værdi i form af Key.

```
public Node findKey(int key){
    if(key == this.key){
        return this;
    }else if(key < this.key){
        return this.left != null ? this.left.findKey(key) : null;
    }else{
        return this.right != null ? this.right.findKey(key) : null;
    }
}
```

Da hver node har en referance til sine børn, skal man bare spørge rod-noden om at finde en given værdi, og den vil så returnere den faktiske node, hvis den findes. Først bliver der tjekket på om den givne node er den samme som den key der bliver søgt efter hvis nej, bliver det barn som enten er større eller mindre end den givne key spurgt, forudsat denne ikke er null, den kalder så samme metode på sine børn og på den måde bliver den korteste rute til værdien fundet.

```
public void addNode(int key){
    if(key < this.key){
        if(this.left == null){
            this.left = new Node(this, key);
        }else{
            this.left.addNode(key);
        }
    }else if(key > this.key){
        if(this.right == null){
            this.right = new Node(this, key);
        }else{
            this.right.addNode(key);
        }
    }else{
        System.out.println("Duplicate key: " + key);
        throw new InvalidParameterException("Duplicate key: " + key);
    }
}
```

Når der skal tilføjes en ny værdi udfører man 2 simple trin
- Hvis værdien er større end den man har, fortæller man det højre barn at den skal tilføje værdien til sine børn, medmindre der ikke er et højre barn så bliver det oprettet på den nuværende node
- Hvis værdien er mindre end den man har, gør man det same, bare på det venstre barn.

DictBinTree.java-klassen

```
@Override
public void insert(int k) {
    if(this.root == null){
        this.root = new Node(null, k);
    }else{
        this.root.addNode(k);
    }
}
```

Når der skal tilføjes en node bliver insert kaldt, denne tjekker om roden er null, hvis den er bliver noden oprettet som rod-noden.

```
@Override
public int[] orderedTraversal() {
    ArrayList<Integer> list = new ArrayList<Integer>();
    inorderTreeWalk(this.root, list);

    int[] new_list = new int[list.size()];
    for(int i = 0; i < new_list.length; i++){
        new_list[i] = list.get(i);
    }
    return new_list;
}
```

orderedTraversal er implementeret som et kald til en rekursiv metode, der bliver sendt en arrayList med som parametre og det er så denne elementerne bliver tilføjet til.

```java
private void inorderTreeWalk(Node x, ArrayList<Integer> list){
    if(x != null){
        inorderTreeWalk(x.left, list);
        list.add(x.key);
        inorderTreeWalk(x.right, list);
    }
}
```

Da x godt kan være null (i tilfælde af vi er nået bunden af vores træ) bliver der først tjekket på dette. I et søgetræ er de mindste værdier altid helt til venstre, så derfor bliver der kaldt den rekursive metode på alle de venstre børn først, når man så er nået bunden, bliver værdien tilføjet og man kører alle de højre børn igennem, på den måde får man alle elementer i træet fra venstre mod højre og de er sorteret fra mindste til højeste.

```java
@Override
public boolean search(int k) {
    if(this.root == null){
        return false;
    }
    return this.root.findKey(k) != null;
}
```

search spørger roden om at finde en given key, denne spørg så sine børn som beskrevet tidligere.

Test

Der er blevet udført test gennem et tilfældigt antal inputs, med forskellige nøgleværdier liggende mellem $-1 * 10^6$ og 10^6.
Der er specifikt til det binære søgetræ tilføjet et tjek, så man ikke kan tilføje en værdi som allerede findes.
Det gælder for begge programmer at hvis der bruges tal-værdier som ikke kan passe i en integer så vil det fejle.
Der er endvidere blevet benyttet testen fra opgave siden (“http://www.imada.sdu.dk/~rolf/Edu/DM507/F14/”), hvilket også gav de forventede resultater.