Syddansk Universitet

Institut for matematik og Datalogi

Algoritmer of Datastrukturer DM507
Hold S17

Projekt del 1

Author:
Mads Petersen
Aslak Johannes Vainio-Pedersen

Instruktor:
Lasse Rode Melbye

6. april 2014

Indhold

1 Introduktion

Vi fik stillet til opgave at lave to implementationer af pseudokode til løsning af sorterings og opbevaringsproblemer. I den første del beskrives løsning ved heapsort på engelsk, i anden del beskrives hvordan vi med treesort har løst opgave 2 i del 1 af projektet. Tredje opgave i projektdel 1 er en fortsættelse af opgave 1 og 2 idet der bliver spurgt til yderlige implementering af kode som overbygning til det allerede eksisterende. Af den grund er opgave 3 indrullet i besvarelsen af opgave 1 og 2, så det hele fremstår som en samlet besvarelse.

Rapporten indeholder kode i bidder og segmenter alt efter praktisk behov, inkluderet med verbatim-miljøet i LaTeX.

2 Heap

2.1 Interface

The first two classes were given, so we will only make a cursory inspection. The first class is just an overview of what methods PQ is to implement. The first one extracts (returns) an element with the method extractMin();, the second one inserts and element and thus needs to be fed something to work.

```
public interface PQ
{
    public Element extractMin();
    public void insert(Element e);
}
```

2.2 Element

The second of the predefined classes defines the Element-object. It contains two pieces of data, a key and arbitrary data. The key is an integer which will also be used to determine the priority of the element.

```
public class Element
{
    public int key;
    public Object data;

    public Element(int i, Object o)
    {
        this.key = i;
        this.data = o;
    }
}
```

2.3 PQHeap

In PQHeap things get interesting, we need to implement methods both to work in the background and to interact with the interface. First we initialize an integer, HeapSize, to keep track of how many entries there are in our heap, as well as the Element-array, an array of elements which have the properties defined in that class. In the constructor we actually make the array, because of java using the index 0 for the first object in an array we have to do some re-working. We chose to add one to the size of the array and not use index 0.

The three definitions of Parent, Left and Right are simply utility-methods more or less copied from the pseudocode in our textbook.

In the extractMin method we have a two-part if-statement. First we check for an empty list (though we were allowed to assume a nonemptylist it is considered good form to take this in to account, as well as making error-checking easier). In the else-part we extract the first element (which is also the one with highest priority, if heap-property is upheld), the we take the element with lowest priority and copy it into the top-priority-slot,

followed by a deletion of that lowest priority-index. Now when we run MinHeapify to uphold heap-property we have a tree like the one before, but without the former top-priority-element.

In the insert-method we use the InsVal integer to keep track of where the newly inserted element is, and we use the Element temp to move around any elements that do not satisfy the property of "Parent-key is higher priority (lower value) than child-key". In the if-statement we check to see that there are actually elements in the heap. In the while loop we check both that we are inserting into a non-root location and that child-keys are lower than parent-keys.

MinHeapify is more or less a direct translation of the pseudo in our textbook. Instead of max we have min and instead of integers directly, we look at a property of our elements in the heapRay-array.

```
public class PQHeap implements PQ
{

int HeapSize;
Element[] heapRay;

public PQHeap(int maxElms)
{
heapRay = new Element[maxElms+1];
HeapSize = 0;
}

public int Parent(int i) {
return i/2;
}

public int Left(int i) {
return i * 2;
}

public int Right(int i) {
return i * 2 + 1;
}


public Element extractMin() {
Element minEl;
if (HeapSize < 1) {
System.out.println("List empty");
return null;
}
else {
minEl = heapRay[1];
heapRay[1] = heapRay[HeapSize];
HeapSize--;
MinHeapify(1);
return minEl;
}
}


public void insert(Element e) {
Element temp;
int InsVal;
HeapSize++;
heapRay[HeapSize] = e;
InsVal = HeapSize;
if (HeapSize == 1) {
heapRay[1] = e;
}
else {
```

```
while (InsVal > 1 && heapRay[InsVal].key < heapRay[Parent(InsVal)].key) {
temp = heapRay[Parent(InsVal)];
heapRay[Parent(InsVal)] = e;
heapRay[InsVal] = temp;
InsVal = Parent(InsVal);
}
}
}

public void MinHeapify(int i) {
int least;
int l;
int r;
Element temp;
l = Left(i);
r = Right(i);
if (l <= HeapSize && heapRay[l].key < heapRay[i].key) {
least = l;
}
else {
least = i;
}
if (r <= HeapSize && heapRay[r].key < heapRay[least].key) {
least = r;
}
if (least != i) {
temp = heapRay[i];
heapRay[i] = heapRay[least];
heapRay[least] = temp;
MinHeapify(least);
}
}

public Element returnElement(int id) {
return heapRay[id];
}

}
```

2.4 Heapsort

Heapsort is the class that contains the main method. It is thus the file we run first. It contains several major elements. First we initialize array and scanner (we use arraylists to find out the size neeeded in the heap). The first part basically adds entries into the arraylist as long as there are entries to add.

Then we create the heap to correspond in size with the arraylist.

In the next part we insert elements into the heap, they have the key-values that got put in via the scanner before, while keeping all the data-objects as "null". Lastly we extract and print out all the elements with extractMin-commands, thus getting a sorted list.

```
import java.util.Scanner;
import java.util.ArrayList;

public class Heapsort {
public static void main(String[] args){
Scanner inputScanner = new Scanner(System.in);
ArrayList myAlist = new ArrayList(1);
while (inputScanner.hasNextInt()){
myAlist.add(new Integer(inputScanner.nextInt()));
}
if(inputScanner!=null)
inputScanner.close();
```

```
PQHeap myHeap = new PQHeap(myAlist.size());

for (int i=0; i<=myAlist.size()-1; i++ ){
Element myElement = new Element(((Integer)myAlist.get(i)).intValue(), null);
myHeap.insert(myElement);
}

while (myHeap.HeapSize > 0){
Element out = myHeap.extractMin();
System.out.println(out.key);
}
}

}
```

Test

The code has been tested with TestProjectPartI.java which was given. It has also been tested with permutations ranging from 1 to 10 integers manually inserted through terminal. Furthermore Read-File-functionality was tested with .txt files containing 100-1000 random integers separated by spaces, all returning sorted lists as expected. The files have been tested in Eclipse as well as directly in the terminal.

3 Treesort

De 4 java filer Dict.java, DictBinTree.java, DictBinTreeNode.java og Treesort.java som udgør opgave 2 og noget af opgave 3 af dette projekt vil blive beskrevet efterfølgende. Dict.java er blot interfacet der angiver hvilke metoder der mindst skal tilbydes af DictBinTree så denne vil ikke blive beskrevet nærmere.
DictBinTree.java og DictBinTreeNode.java udgør tilsammen strukturen beskrevet i kapitel 12 i "Introduction to Algorithims"(som er den bog der er referet til når der skrives "bogen"eller ligende), DictBinTree indeholder de påkrævede metoder

```
search(int k)
insert(int k)
orderedTraversal()
```

DictBinTreeNode.java udgør knuder i træ-strukturen og består af 4 felter:

```
int key;
DictBinTreeNode left;
DictBinTreeNode right;
DictBinTreeNode parent;
```

Og klassens konstruktor tager 2 argumenter, en integer som er den værdi der skal insættes i træet, og en knude der er "forælder"til knuden:

```
public DictBinTreeNode(int key,DictBinTreeNode parent)
    {
        this.key = key;
        this.parent = parent;
        left = null;
        right = null;
    }
```

Nye knuder insættes med left = null og right = null, da dette er den måde det gøres på i pseudo-koden i kapitel 12. Der ud over tilbyder klassen også 5 metoder, 3 til at sætte left, right of parent til knuden. Og en til at returnere den værdi der er gemt i knuden og en metode der fortæller om knuden har børn eller ej som aldrig endte med at blive brugt i projektet.
DictBinTree indeholder de 3 metoder search, insert og orderedTraversal som skal implementeres i opgave 2 hvoraf insert og orderedTraversal skal bruges til at løse opgave 3.

DictBinTree.java

search(int k)

```
public boolean search(int k)
    {
        if(root != null)
        {
            DictBinTreeNode search = root;
            while(search.key != k)
            {
                if (search.key <= k)
                {
                    if(search.right != null)
                        search = search.right;
                    else return false;
                }
                else
                {
                    if(search.left != null)
                        search = search.left;
                    else return false;
                }

            }
            return true;
        }
        return false;
    }
```

Den starter ved roden, hvis roden ellers er sat, ellers returnerer den altid false. Derefter går den til knuden der er til højre hvis værdien der søges efter er større end eller lig værdien i roden og ellers til venstre, og således fortsætter den til værdien der søges efter er lig værdien i den pågælende knude eller der mødes en blindgyde i.e. en knude hvor værdien er forskellig fra søge-værdien og knuden intet barn har på den pågælende side.

insert(int k)

```
public void insert(int k)
    {
        DictBinTreeNode insert = root;
        if(insert != null)
        while(insert != null)
        {
            if(k < insert.key)
            {
                if(insert.left == null)
                {
                    insert.left = new DictBinTreeNode(k,insert);
                    insert = null;
                }
                else insert = insert.left;
            }
            else
            {
                if(insert.right == null)
                {
                    insert.right = new DictBinTreeNode(k,insert);
                    insert = null;
                }
                else insert = insert.right;
            }
```

```
        }
        else
        {
            root = new DictBinTreeNode(k,null);
        }

        nodes++;
    }
```

Denne metode gør stort set det samme som search, ud over at den indsætter en ny knude når den finder en blindgyde, og hvis der ingen rod er, så indsættes knuden automatisk som rod. Der ud over bliver nodes forøget med 1 hver gang insert bliver kaldt, dette felt bliver brugt til at holde styr på hvor mange værdier der er gemt i træet.

orderedTraversal

```
public int[] orderedTraversal()
    {
        helper = new ArrayList(nodes);
        treeWalk(root);
        int[] result = new int[nodes];
        int i = 0;
        while(helper.size() > 0)
        {
            result[i] = (int)helper.get(0);
            helper.remove(0);
            i++;
        }
        return result;
    }

    private void treeWalk(DictBinTreeNode start)
    {
        if(start != null)
        {
            treeWalk(start.left);
            helper.add(start.key);
            treeWalk(start.right);
        }
    }
```

Denne metode benytter feltet helper som er en ArrayList der bliver initialiseret med størrelse lig variablen nodes da dette er antallet af knuder i træet. der efter bliver treeWalk kaldt, denne metode er en direkte oversættelse af den rekursive "INORDER-TREE-WALK(x)"pseudo-kode på side 288 i bogen. Denne metode bruges så til at insætte værdierne i ArrayListen helper istedet for at udskrive dem, her efter bruger orderedTraversal helper feltet til at indlæse værdierne i et integer array en efter en, hvorefter de slettes fra ArrayListen, til sidst returneres arrayet.

Treesort.java

```
public static void main(String[] args)
    {
        Dict myTree = new DictBinTree();
        Scanner inputScanner = new Scanner(System.in);
        while(inputScanner.hasNextInt())
        {
            myTree.insert(inputScanner.nextInt());
        }
        int[] numbers = myTree.orderedTraversal();
        for(int i = 0;i < numbers.length; i++)
        {
            System.out.print(numbers[i]+" ");
```

```
        }
    }
}
```

Treesort består af en enkelt main metode, som laver et nyt binært træ og et scanner object der scanner standard input. Så længe der ikke bliver scannet noget fra standard input som ikke kan tolkes som en integer så kører while loopet og insætter værdierne i træet, når der bliver scannet noget der ikke er en integer, stopper loopet og resultatet bliver udskrevet på skærmen ved at læse arrayet fra start til slut.

Test

Koden er teste med TestProjectPartI.java og den er testet ved at insætte 25 tal genereret med java.util.Random fra 0 - 1000 og derefter en ordnet gennemgang side om side med de indsatte tal, printet på skærmen, alt sammen gjort i netbeans IDE. Der ud over er der også testet i kommado prompt med manuel indskrivning af tilfældige tal og testet ved brug af redirection fra en fil med tilfældige tal adskilt af mellemrum.
