UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS212 - Data structures and algorithms

Practical 7 Specifications: Skew heap

Release date: 29-04-2024 at 06:00

Due date: 03-05-2024 at 23:59

Total marks: 200

Contents

1 General Instructions

- *Read the entire assignment thoroughly before you start coding.*
- This assignment should be completed individually; no group effort is allowed.
- **To prevent plagiarism, every submission will be inspected with the help of dedicated software.**
- Be ready to upload your assignment well before the deadline, as no extension will be granted.
- You may not import any of the built-in Java data structures. Doing so will result in a zero mark. You may only use native 1-dimensional arrays where applicable. If you require additional data structures, you will have to implement them yourself.
- If your code does not compile, you will be awarded a mark of zero. Only the output of your program will be considered for marks, but your code may be inspected for the presence or absence of certain prescribed features.
- If your code experiences a runtime error, you will be awarded a zero mark. Runtime errors are considered unsafe programming.
- Read the entire specification before you start coding.
- **Ensure your code compiles with Java 8**
- The usage of ChatGPT and other AI-Related software is strictly forbidden and will be considered as plagiarism.

2 Plagiarism

The Department of Computer Science considers plagiarism a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with consent), and copying material (such as text or program code) from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to http://www.library.up.ac.za/plagiarism/index.htm (from the main page of the University of Pretoria site, follow the Library quick link, and then choose the Plagiarism option under the Services menu). **If you have any form of question regarding this, please ask one of the lecturers to avoid any misunderstanding.** Also note that the OOP principle of code reuse does not mean that you should copy and adapt code to suit your solution.

3 Outcomes

On completion of this practical, you will have gained experience with the following:

- Implementing a skew heap.

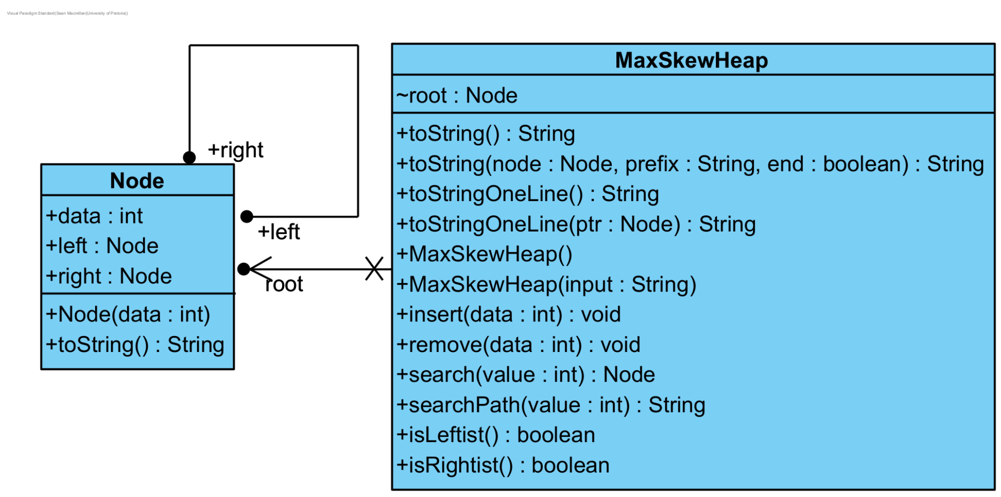
4 Introduction

A leftist heap is a variant of a binary heap. Each node has an npl-value which is the distance to the nearest leaf. In a leftist heap, for each node, the npl-value of the left child is greater than or equal to the npl-value of the right child. This property gives the heap a bias towards the left, hence the name leftist heap. We can also define a rightist heap, where for each node, the npl-value of the right child is greater than or equal to the npl-value of the left child.

Skew heaps are a variant of leftist heaps, where the npl-values are not stored. This is done to create a more memory efficient version of the leftist heap, since the nodes need to store less data. Additionally, it is also slightly more time efficient, since the swaps are always performed in a skew heap, whereas in leftist heaps, there is a comparison first which then determines whether a swap should happen.

5 Tasks

You are tasked with implementing a max skew heap. The skew heaps in the lectures and textbook were min skew heaps, where for each node, the node value is the smallest value in that subtree. For this practical we will implement a max skew heap, where for each node, the node value is the largest value in that subtree.



Node
+data : int
+left : Node
+right : Node
+Node(data : int)
+toString() : String
+right
+left
root
MaxSkewHeap
~root : Node
+toString() : String
+toString(node : Node, prefix : String, end : boolean) : String
+toStringOneLine() : String
+toStringOneLine(ptr : Node) : String
+MaxSkewHeap()
+MaxSkewHeap(input : String)
+insert(data : int) : void
+remove(data : int) : void
+search(value : int) : Node
+searchPath(value : int) : String
+isLeftist() : boolean
+isRightist() : boolean

Figure 1: UML

Don't change the signatures of the given functions or members. You are allowed to add your own variables or functions to both classes.

5.1 Node

- The node class is given to you.
- This class is used as data nodes for the MaxSkewHeap.

5.2 MaxSkewHeap

- root: Node
  - This is the root of the heap.
  - If the heap is empty, this will be null.
- Functions
  - toString(), toStringOneLine()
    - These functions are given to you, and are used on Fitchfork to mark your output.
    - The `toString()` prints out a readable version which is split over multiple lines.
    - The `toStringOneLine()`, prints the heap in one line. This is used to reconstruct heaps using the string constructor.
  - MaxSkewHeap()
    - This is the default constructor.
    - This initialises the heap to an empty heap.
  - MaxSkewHeap(input: String)
    - This is the string constructor.
    - You may assume that the string can be generated by the `toStringOneLine()` function. Thus, you don't need to check whether the string has valid formatting.
    - This function should directly recreate the passed-in string. You don't need to check whether the max heap property is satisfied, as this function should be able to create heaps which break this property. *Note : If a heap is passed-in which violate the max-heap property then some of the other functions will give the wrong output, as all other functions should assume that the current object is a valid max skew heap.*
    - Hints:
      - Null nodes are represented as:

```
{}
```

      - Normal nodes follows the format:

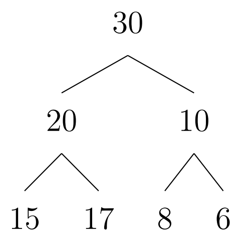
```
{<Data><Left child representation><Right child representation>}
```

      - Example, if we have a heap with 5 as the root, with a left child of 4, and a right child of 3 the string representation is:

```
{5{4{}{}}{3{}{}}}
```

      - It is easiest to implement a recursive solution to this function.
      - When given an input string, split it into three parts. The first part should be the data of the current node, the second part should be the left child's representation, and the third part should be the right child's representation. Use recursion to create the left and right children.
      - To split the string into the left and right parts, iterate through the input string. The first time when the number of open braces equals the number of closing braces, that you means you have found the end of the left child.
  - insert(data: int): void
    - Inserts the passed-in value into the heap. The insertion rule from the lectures and textbook should be used.
    - If a value is passed-in which is already in the heap, then don't insert it, since duplicates are not allowed.

- remove(data: int): void
  - Delete the passed-in value from the heap.
  - In the lectures and textbook, only the root was allowed to be deleted. For this practical, any value in the heap can be deleted.
  - The deletion algorithm follows the same idea as the textbook and lecture slides, where the two children are merged together. The deleted node is then replaced by the result of the merging.
- search(value: int): Node
  - If no node is found with the passed-in value, then return null.
  - Searching was not covered in the lectures or textbook, so use the following searching strategy which exploits the leftist and heap properties of skew heaps:
    - Searching is done using a right to left pre-order depth first traversal.
    - If the current node is smaller than the value you are searching for, then stop searching down this path. This is due to the max heap property, it is impossible for the node you are looking for to be in the subtree(s) of this node.
    - Since skew heaps should be leftist most of the time, this means the right subtree should be smaller. Therefore, we first search down the right subtree, then the left subtree.
- searchPath(value: int): String
  - This prints out a visual representation of the search algorithm. It shows every node that was visited and the order when running the search algorithm.
  - Node values are appended together using "->". If a node is found that matches the passed in value, put it inside square brackets: "[]".
  - If the passed-in value is not found within the tree, then the full search path will be printed, and there should be no square brackets.
  - If the tree is empty, it should return an empty string.
  - Example: Search path for searching for 15



```
30->10->20->17->[15]
```

- isLeftist(): boolean
  - Returns true if the current heap is a leftist heap.
- isRightist(): boolean
  - Returns true if the current heap is a rightist heap.
  - If you are unsure what a rightist heap is, read through the introduction section of this document.

6 Testing

As testing is a vital skill that all software developers need to know and be able to perform daily, approximately 10% of the assignment marks will be assigned to your testing skills. To do this, you will need to submit a testing main (inside the Main.java file) that will be used to test an Instructor-provided solution. You may add any helper functions to the Main.java file to aid your testing. In order to determine the coverage of your testing the jacoco tool. The following set of commands will be used to run jacoco:

```
javac *.java
rm -Rf cov
mkdir ./cov
java -javaagent:jacocoagent.jar=excludes=org.jacoco.*,destfile=./cov/output.exec
   -cp ./ Main
mv *.class ./cov
java -jar ./jacococli.jar report ./cov/output.exec --classfiles ./cov --html
   ./cov/report
```

This will generate output which we will use to determine your testing coverage. The following coverage ratio will be used:

$$\frac{\text{number of lines executed}}{\text{number of source code lines}}$$

and we will scale this ratio according to the size of the class.
The mark you will receive for the testing coverage task is determined using table 1:

Coverage ratio range	% of testing mark
0%-5%	0%
5%-20%	20%
20%-40%	40%
40%-60%	60%
60%-80%	80%
80%-100%	100%

Table 1: Mark assignment for testing

Note the top boundary for the Coverage ratio range is not inclusive except for 100%. Also, note that only the function stipulated in this specification will be considered to determine your mark. Remember that your main will be testing the instructor-provided code and as such it can only be assumed that the functions outlined in this specification are defined and implemented.

7 Upload checklist

The following files should be in the root of your archive

- Main.java
- Any textfiles needed by your Main
- Node.java
- MaxSkewHeap.java

8 Allowed libraries

- None

9 Submission

You need to submit your source files on the FitchFork website (https://ff.cs.up.ac.za/). All methods need to be implemented (or at least stubbed) before submission. Place the above-mentioned files in a zip named uXXXXXXXX.zip where XXXXXXXX is your student number. Your code must be able to be compiled with the Java 8 standard.

For this practical, you will have 5 upload opportunities and your best mark will be your final mark. Upload your archive to the appropriate slot on the FitchFork website well before the deadline. **No late submissions will be accepted!**