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: Treaps

Release date: 07-04-2025 at 06:00
Due date: 11-04-2025 at 23:59
Total marks: 172

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 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.
- Note that plagiarism is considered a very serious offence. Plagiarism will not be tolerated, and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at `https://portal.cs.up.ac.za/files/departmental-guide/`.
- Please note that there is a late deadline which is 1 hour after the initial deadline, but you will lose 20% of your mark.

2 Overview

Treaps are data structures that combine both heaps and trees hence their name. For this practical, treaps are implemented as binary tress where and each node has a value and a priority, priorities will determine which node is closer to the root as in max-heaps. The advantage of such a structure is, binary search is possible as well as balancing based on priorities. Refer to the Class diagram below.

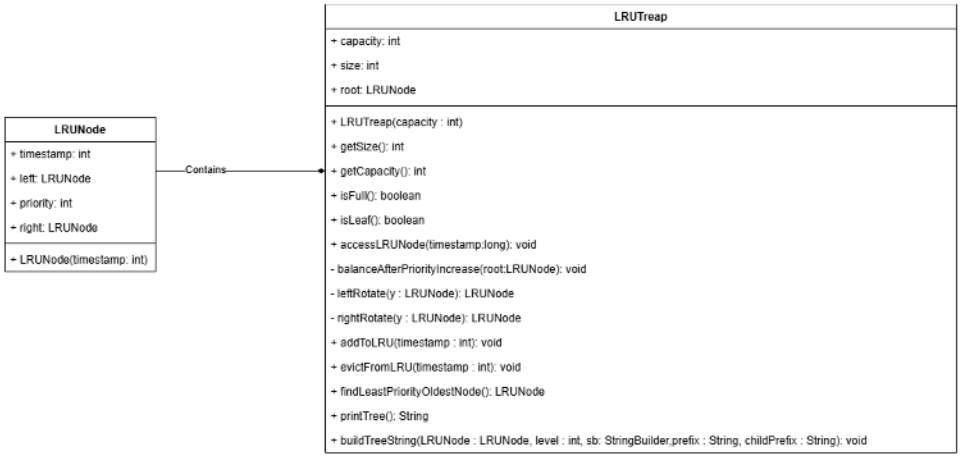

Figure 1: Treap Structure for LRU Cache

3 Tasks

In this practical you will be implementing a Treap data structure that is to be used in a caching system. The caching system adheres to the Least Recently Used(LRU) strategy. When the cache is full and we require additional items in the cache, we undergo a process of eviction where the least recently used item is removed to make space, in addition to this, each item in the Treap is associated with a priority, we will evict nodes based on their priority as well. This means we will evict the nodes that have the least priority and in the case of identical priorities the least recently accessed node. Each node has an integer value timestamp value which will be used to represent the access time.

3.1 LRUTreap

This is the Treap implementation for the cache system, nodes are maintained and ordered by timestamp and priority values.

3.1.1 LRUNode

- A private inner class that represents nodes in the treap, storing timestamp, priority, and references to left and right children.
- Contains the timestamp (serving as the key for BST ordering) and priority value for heap property.

capacity

- Maximum number of nodes the cache can hold

size

- Current number of nodes in the cache, updated during insertions and deletions.

root

- Reference to the root node of the treap structure.

3.1.2 Functions

LRUTreap

- Constructor that initializes the cache with a specified capacity.

getSize()

- Returns the current number of elements in the cache.

getCapacity

- Returns the maximum capacity of the cache.

isFull

- Checks if the cache has reached its capacity.

isLeaf

- Helper method that checks if a node is a leaf node (has no children).

accessLRUNode

- Updates the priority of a node by 1 when it's accessed, increasing its priority in the heap and rebalances the treap as needed.
- accessing has been simplified to just include searching through the tree

balanceAfterPriorityIncrease

- Rebalances the treap after a node's priority has been increased, maintaining the heap property.
- Performs rotations if a child has higher priority than its parent.

rightRotate and leftRotate

- Rotation operations for maintaining the heap property of the treap during rebalancing.

addToLRU

- Adds a new node to the cache with the specified timestamp.
- If the cache is full, evicts the least frequently used (lowest priority) and oldest node and updates the size counter.
- The node to be evicted should be printed with: ("Evicting LRUNode: timestamp = " + toEvict.timestamp + ", priority = " + toEvict.priority) *exclude brackets
- addition of an item that already exists increases the priority of that node.

evictFromLRU

- Removes a node with the specified timestamp from the cache.

findLeastPriorityOldestLRUNode

- Locates the node with the lowest priority or, if multiple nodes have the same priority, the one with the oldest timestamp.
- Used to determine which node to evict when the cache is full.

printTree Do not change this as it will negatively impact your marks

- Prints the entire tree structure in a visual format, showing each node's timestamp and priority.

4 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.

5 Upload checklist

The following files should be in the root of your archive

- LRUTreap.java
- Main.java
- Any textfiles needed by your Main

6 Allowed libraries

- No imports allowed.

7 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!**