# COMP3506 Homework 3 – Heaps, Trees, Hashtables

Weighting: 15%

Due date: 25th September 2020, 11:55 pm

## Overview

In this assignment, you will be applying your knowledge of trees, heaps, maps, sets, and hashtables to Java programming questions.

## Marks

This assignment is worth 15% of your total grade. Both COMP3506 and COMP7505 students will be marked on questions 1 to 4 out of **45 marks**.

## Submission Instructions

- Your solutions to Q1 will be submitted via Gradescope to the `Homework 3 - Question 1` submission. You should only submit your completed `QuaternaryHeapsort.java` file.
- Your solutions to Q2 will be submitted via Gradescope to the `Homework 3 - Question 2` submission. You should only submit your completed `StrongHeap.java` file.
- Your solutions to Q3 will be submitted via Gradescope to the `Homework 3 - Question 3` submission. You should only submit your completed `BinaryTreeComparator.java` file.
- Your solutions to Q4 will be submitted via Gradescope to the `Homework 3 - Question 4` submission. You should only submit your completed `LinkedMultiHashSet.java` file.
- No marks will be awarded for non-compiling submissions, or submissions which import non-supported 3rd party libraries. You should follow all constraints laid out in the relevant questions.
- Zip files (or other compressed file formats) won't be marked - you should only submit the relevant `java` file to Gradescope.

## Late Submissions and Extensions

Late submissions will not be accepted. It is your responsibility to ensure you have submitted your work well in advance of the deadline (taking into account the possibility of internet or Gradescope issues). Only the latest submission before the deadline will be marked. See the ECP for information about extensions.

## Academic Misconduct

This assignment is an individual assignment. Posting questions or copying answers from the internet is considered cheating, as is sharing your answers with classmates. All your work (including code) will be analysed by sophisticated plagiarism detection software. Students are reminded of the University's policy on student misconduct, including plagiarism. See the course profile and the School web page: `http://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism`.

## Support Files Provided

The following provided support files you should not modify or submit:

- `BinaryTree.java` – an implementation of a binary tree node. A binary tree is made up of many binary tree nodes linked together.
- `MultiSet.java` – a multiset ADT.

The following support files have been provided to you, but you will need to modify them to complete the relevant questions and then submit them:

- `QuaternaryHeapsort.java` – you need to implement your question 1 solution in this file.
- `StrongHeap.java` – you need to implement your question 2 solution in this file.
- `BinaryTreeComparator.java` - you need to implement your question 3 solution in this file.
- `MultiLinkedHashSet.java` – you need to implement your question 4 solution in this file.

## Notes and Constraints

- You will be marked by a human on your code style (e.g. for following the style guide, and general readability) for all programming questions. You should follow the CSSE2002 style guide as given on Blackboard.
- Your solution will be automatically marked using Java 11. No marks will be awarded for non-compiling submissions, or submissions which import non-supported 3rd party libraries.
- Do not add any public methods, constructors, or fields to Java classes besides those specified in the relevant questions. You may add private helper methods if needed.
- You shouldn't use any additional classes from the Java Collections Framework (e.g. `ArrayList`, `LinkedList`, `HashMap`), unless otherwise stated or already imported for you in the support files (e.g. `Iterator` or `Comparator`).
- Your solutions to all questions should be as efficient as possible (with regards to their time complexity).
- Sample tests have been provided. However, these are far from exhaustive and your actual submissions will be marked on a much larger set of test cases. You should ensure you write your own tests while working on your assignment.

# Questions

1. (10 marks) Recall in-place heapsort described in lectures and tutorials, where we build a binary heap in-place bottom-up within an array. A variant of a binary heap is a quaternary heap which is a heap where nodes can have (at most) 4 children.

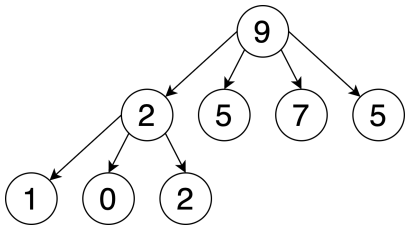
   In this question, you will implement **in-place quaternary heapsort**. This should build a quaternary heap in-place bottom-up, then use this to sort the array in ascending order.

   (a) First, you should implement the helper function `quaternaryDownheap` in `QuaternaryHeapsort.java`. This should perform a downheap operation on a quaternary max heap within an array.

   (b) Using this helper function, implement `quaternaryHeapsort` within `QuaternaryHeapsort.java`.

   (c) State the worst case time and space complexity (in Big-O notation) of all your methods in their Javadoc.

   Below is an example visualisation of a quaternary max heap. Note that it is a complete quaternary tree (leaf nodes of bottommost level are as far left as possible). The array representation of the below heap would be $[9, 2, 5, 7, 5, 1, 0, 2]$.




   Here are some examples of `quaternaryDownheap`. It works on the array representation of a heap and performs the downheap in-place.

   - Input heap $[0, 10, 20, 30, 40]$ and start index 0 would modify the array to $[40, 10, 20, 30, 0]$.
   - Input heap $[1, 2, 3, 4, 0, 10, 20, 30, 40]$ and start index 4 would modify the array to $[1, 2, 3, 4, 40, 10, 20, 30, 0]$.

   Notes:

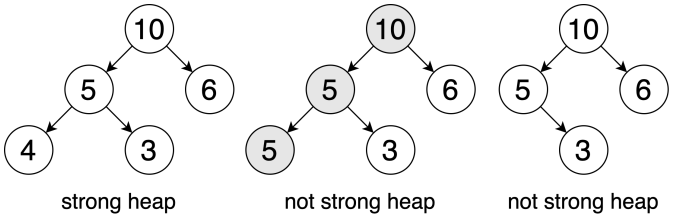
   - You should create and use private helper methods to improve readability of your code.
   - You will be given no marks for any variant of heapsort that doesn't utilise a quaternary heap.

2. (10 marks) We will call a binary tree a **strong heap**[1] if it is a complete binary tree and it satisfies the "strong (max) heap property" as:

   - for *all* nodes $x$, we have $x < \text{parent}(x)$, and
   - for nodes $x$ in level 2 or below, we also have $x + \text{parent}(x) < \text{parent}(\text{parent}(x))$.

   In `StrongHeap.java`, implement `isStrongHeap`. This takes one argument, the root of a binary tree, and returns whether the given binary tree is a strong max heap. State the worst case time and space complexity (in Big-O notation) in the method's Javadoc.

   These are some examples of trees which are and are not strong heaps. The second is not a strong heap because $5+5 \not< 10$. The third is not a strong heap because the tree is not complete, despite satisfying the strong heap property.




   Hints and notes:

   - As a reminder, the root node of a tree is at level 0.
   - A tree with a single node is trivially a strong binary heap.
   - Strong binary heaps are a subset of binary heaps.
   - If a binary tree is not complete, it cannot be a strong heap. If a tree is complete, you must check the strong heap property to determine whether it is a strong heap.
   - Node values can be negative but not null. Negatives should not be treated specially.

[1] This is just terminology used here and has no relevance outside this assignment.

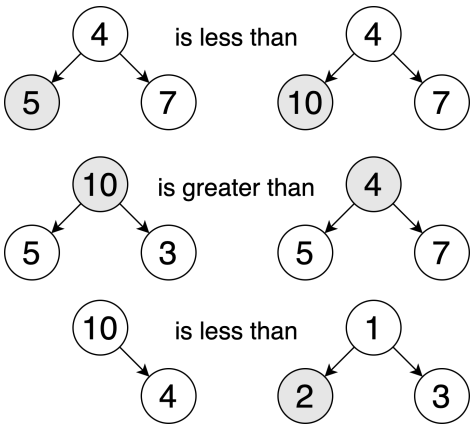
3. (10 marks) This question is about **comparing binary trees**.

   To compare two binary tree nodes, we first compare their left subtrees, then compare their values (with `compareTo`), then compare their right subtrees. If all of these are equal, then the nodes are equal. Otherwise the first non-equal comparison in this order is the result of the comparison. To compare the subtrees, you should recurse and compare them in the same way as described here.

   Regarding nulls (e.g. missing left/right children), a null subtree should compare equal to another null, or less than any non-null subtree.

   Implement the `BinaryTreeComparator` class in `BinaryTreeComparator.java`. This has a single method `compare` which takes two trees $x$ and $y$ then returns $-1, 0, +1$ if $x < y$, $x = y$, or $x > y$ (respectively). State the worst case time and space complexity (in Big-O notation) in the method's Javadoc.

   Here are some examples of trees and how they compare. The highlighted node(s) are those which determine the comparison result. Take note of the last case where the null left subtree is less than the non-null left subtree.




   Hints and notes:

   - You may add private helper methods if needed, but you should not use any instance variables.
   - You can assume that values stored within nodes are not null and implement `Comparable` (so you can use the `compareTo` method).
   - You cannot assume the node itself will be non-null. Nulls should be compared as described above.
   - You should not perform unnecessary comparisons. For example, if the left subtrees differ, you should not compare the right subtrees.
   - Although the `Comparator` interface only requires the return value to be negative/zero/positive, here you must return exactly $-1, 0$, or $+1$.

4. (15 marks) A **multiset** is a collection that extends the idea of a set to have duplicate items. It is able to keep track of the number of occurrences of each duplicate in the set.

   For this question, you will implement `LinkedMultiHashSet`, a multiset that internally uses a resizing array and hashing (based on element's `hashCode`s) to make most operations run in O(1) average time. In addition to the ordinary methods for a set, you will need to implement an `Iterator` which returns elements in a particular order (see below and Javadoc).

   You should also state the worst case time and space complexities of all your methods (in Big-O notation) in their respective Javadocs.

   To gain full marks, you should ensure that:

   - The internal hashtable array should start with the initial `capacity` given as the argument to the constructor of the class.
   - The internal hashtable array should be **doubled** in size whenever an `add` operation would make the array become full. It should not reduce in size.
   - Duplicate occurrences of elements should not take up additional space.
   - You should use `hashCode` to hash objects and `equals` to check equality of elements.
   - When a collision occurs (unequal elements which have the same `hashCode`), you should use **linear probing** to find the next position in the hashtable.
   - The `Iterator` for the multiset should return elements grouped by equivalence (that is, equal elements are returned consecutively). Additionally, these groups of elements should be ordered by when the earliest occurrence of that element was added. Refer to the Javadoc for more details.
   - The methods of `LinkedMultiHashSet` should run in O(1) expected or average time (you can assume well chosen `hashCode` functions for elements passed in). The `next` and `hashCode` methods of the `Iterator` for the set should also run in O(1) time.

   Constraints:

   - Your implementation should all be inside `LinkedMultiHashSet.java`. You may create additional (private) inner classes however.
   - You shouldn't use any additional classes from the Java Collections Framework (e.g. `ArrayList`, `LinkedList`, `HashMap`, `HashSet`). If you wish to utilise similar functionality, you should implement the needed data structures yourself.