Computer Science B63
University of Toronto Scarborough

Homework Exercise # 2

Handing in and marking

For this exercise, you need to submit your solutions to the pencil-and-paper exercises on crowdmark and your solutions to the programming question on MarkUs. Your pencil-and-paper solutions will be marked with respect to correctness, clarity, brevity, and readability. Your code will be marked with respect to correctness, efficiency, program design and coding style, clarity, and readability. This exercise counts for 10% of the course grade.

Question 1. Weight-Balanced trees [20 MARKS]

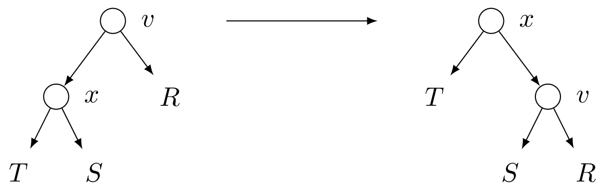
In this question we prove correctness of (part of) the algorithm for rebalancing weight-balanced trees we saw in class, which works as follows, for each node v on the path from newly inserted/deleted node up to the root:

```
if weight(v.right) > weight(v.left) * 3:
  let x = v.right
  if weight(x.left) < weight(x.right) * 2:
    single rotation: counter-clockwise
  else:
    double rotation: clockwise then counter-clockwise
else if weight(v.left) > weight(v.right) * 3:
  let x = v.left
  if weight(x.right) < weight(x.left) * 2:
    single rotation: clockwise
  else:
    double rotation: counter-clockwise then clockwise
else:
  no rotation
```

We will focus on the case

```
else if weight(v.left) > weight(v.right) * 3:
  let x = v.left
  if weight(x.right) < weight(x.left) * 2:
    single rotation: clockwise
```

Your task is to prove that in this case the single clockwise rotation restores the balance.



- Begin by clearly stating your assumptions, together with the justifications for these assumptions.
- State clearly what it means for the rotation to restore the balance.
- Complete the proof!

Question 2. Intersection of Balanced trees [20 MARKS]

Recall the algorithm for finding the union of two weight-balanced trees T_1 and T_2. In this question, you will develop a similar algorithm for finding the intersection of T_1 and T_2:

- a weight-balanced tree T that contains key/value pairs that are in both T_1 and T_2;
- if $(k, v_1) \in T_1$ and $(k, v_2) \in T_2$, then $(k, v_2) \in T$.

First we need to modify the algorithm for splitting a tree T with respect to a key k.

- Provide pseudocode for `split(T, k)` that returns a triple $(T_{<k}, T_{>k}, b)$:
  - $T_{<k}$: a weight-balanced tree that contains all nodes n from T such that $n.key < k$,
  - $T_{>k}$: a weight-balanced tree that contains all nodes n from T such that $n.key > k$, and
  - b: *true*, if the key k appears in T, and $false$, otherwise.

Next we will develop an algorithm for joining two weight-balanced trees without the additional key k.

- Provide pseudocode for algorithm `remove_max(T)` which returns a pair (T', max):
  - T': a weight-balanced tree that contains all key/value pairs of T except for the node max that contains the maximum key in T, and
  - the node max.
- Show how to use `remove_max` to implement the algorithm `join(L, G)`, which takes
  - L: a weight-balanced tree in which all keys are less than those in G, and
  - G: a weight-balanced tree in which all keys are greater than those in L,

  and returns a weight-balanced tree T that contains all key/value pairs from L and G.

Finally provide pseudocode for $intersection(T_1, T_2)$ that returns the intersection of T_1 and T_2.

Question 3. Implementing Heaps [30 MARKS]

We provided you with the starter code for implementing a Minimum Heap. Your task is to implement the functions declared in `minheap.h` that are not already implemented in `minheap.c`. Note that `minheap.c` is the only file you will submit, so make sure your implementation works with the original provided `minheap.h`. Make sure to **carefully study the starter code** before you begin to add your own. The marking scheme for this question is as follows:

- Correctness (includes complexity requirement):
  - `HeapNode getMin(MinHeap* heap)`: 1 mark
  - `HeapNode extractMin(MinHeap* heap)`: 5 marks
  - `void insert(MinHeap* heap, int priority, int id)`: 8 marks
  - `bool decreasePriority(MinHeap* heap, int id, int newPriority)`: 10 marks
- Program design (modular implementation, self-explanatory code, clear logic, no repeated code, no unnecessary code): 4 marks
- Readability and coding style (good use of white space, good naming, etc.): 2 marks
  - You are encouraged (but not required) to use a `lint`er to help check and/or auto-format your code.

The runtime complexity requirements for your functions are as follows (here n is the size of the heap):

- `getMin`: constant time.
- `extractMin`: $\mathcal{O}(\log n)$ time.
- `insert`: $\mathcal{O}(\log n)$ time.
- `decreasePriority`: $\mathcal{O}(\log n)$ time.

Question 4. Breadth-First Search [10 MARKS]

The breadth-first tree found by breadth-first search depends on how vertices are ordered in the adjaceny lists. Of the following directed graph (call it G):

$$\begin{tikzcd}
 & 2 \arrow[r] \arrow[rd] & 5 \\
1 \arrow[ru] \arrow[r] \arrow[rd] & 3 \arrow[ru] \arrow[rd] & 6 \\
 & 4 \arrow[ru] \arrow[r] & 7
\end{tikzcd}$$

- Both trees below are possible breadth-first trees, both starting at $\textcircled{1}$.

$$\begin{tikzcd}
 & 2 \arrow[r] \arrow[dr] & 5 \\
1 \arrow[ur] \arrow[r] \arrow[dr] & 3 & 6 \\
 & 4 \arrow[r] & 7
\end{tikzcd}
\qquad
\begin{tikzcd}
 & 2 & 5 \\
1 \arrow[ur] \arrow[r] \arrow[dr] & 3 \arrow[ur] & 6 \\
 & 4 \arrow[ur] \arrow[r] & 7
\end{tikzcd}$$

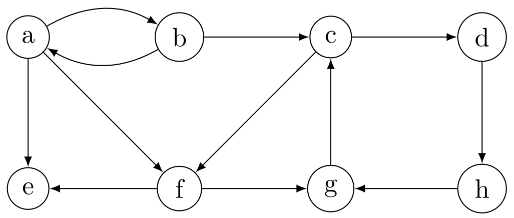
Give an adjacency-list representation of G such that breadth-first search starting at $\textcircled{1}$ results in the left tree, and another representation that results in the right tree.

- Explain why the following **cannot** be a breadth-first tree of G starting at $\textcircled{1}$.

$$\begin{tikzcd}
 & 2 \arrow[dr] & 5 \\
1 \arrow[ur] \arrow[r] \arrow[dr] & 3 \arrow[ur] & 6 \\
 & 4 \arrow[r] & 7
\end{tikzcd}$$

Question 5. Strongly Connected Components [20 MARKS]

This question is about the algorithm for finding strongly connected components (SCCs) we saw in class. Consider the directed graph G and its adjacency lists representation below:



a	b → e → f → ·
b	a → c → ·
c	d → f → ·
d	h → ·
e	·
f	e → g → ·
g	c → ·
h	g → ·

- Show the corresponding adjacency lists representation of G^T, the transpose of G.
- Show the result of running Depth-First Search on G starting with node a:
  - for each vertex v in G, indicate its discovery time $d(v)$ and finish time $f(v)$,
  - R: the list of vertices in order of decreasing finish times.
- Show the result of running Depth-First Search on G^T: the SCCs found by the algorithm, in order.
- Show the component graph of G. (The component graph has a vertex v_i for each SCC C_i of G. It has an edge (v_i, v_j) if G contains an edge from some vertex in C_i to some vertex in C_j.)
- We denote the component graph of G by G^{SCC}. Prove that for any directed graph G, the transpose of the component graph of G^T is the same as the component graph of G. That is, $((G^T)^{SCC})^T = G^{SCC}$.
- Anya and Albert got tired of going through the steps of this complicated algorithm, and are thinking that it could be made easier. Specifically, they want to use the original graph G in the second DFS, but visit the vertices in order of increasing finish times. Will Anya and Albert's simplified algorithm work? Give a proof for your answer.