MCGILL UNIVERSITY – COMP 251

TEST 2

November 17, 2009

Student Number:	
Family Name(s):	
Given Name(s):	

- There are 12 pages in total (including this page).
- You have 60 minutes for this test.
- This test is worth 10% of your final mark.
- Answer each question directly on the test paper, in the space provided. Use the reverse side of the pages for rough work. If you need more space for one of your solutions, use the reverse side of a page and indicate clearly the part of your work that should be marked.
- Refer to the Appendix (pages 11,12) for a list of some algorithms that you might want to use. You can use any of these algorithms, as well as others that have been given in lecture. However, if you want to modify them you need to provide full details of the pseudo-code of the new algorithm.

Question 1 (out of 10)	
Question 2 (out of 10)	
Question 3 (out of 5)	
Question 4 (out of 5)	
Bonus question (out of 5)	
Total (out of 30)	

Question 1 [10]

(a)[2] What is the worst-case running time of Radix sort on an array of length n? State your answer using Θ notation.

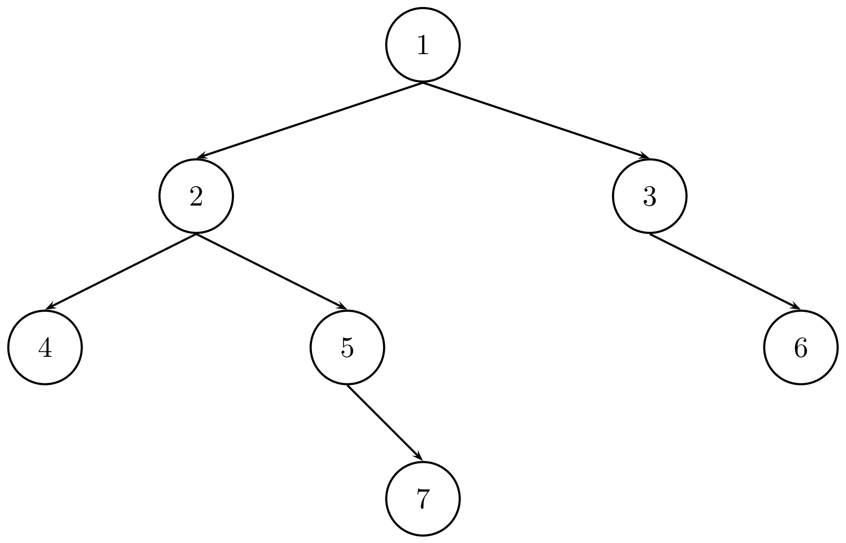
(b)[2] What is the worst-case running time of Bucket sort on an array of length n? State your answer using Θ notation.

(c)[2] Give a precise definition of a strongly connected component of a directed graph.

(d)[2] How can the number of strongly connected components of a directed graph change if a new edge is added to the graph?

Question 1 continues here

(e)[2] Consider the following tree, the number written in each node is the name of the node (node 1 is the root):



For each of the following lists of the nodes of the tree, state whether it is created by an inorder walk, a preorder walk, a postorder walk, or none of these types (you **do not** have to explain why):

(i) 1,2,3,4,6,5,7

(ii) 1,2,5,7,3,6,4

(iii) 4,6,3,7,5,2,1

(iv) 4,2,5,7,1,3,6

Question 2 [10] Given a directed graph G (which might contain cycles) we want to order the vertices of G so that for any two distinct vertices u and v:

if there is a path from u to v, but there is no path from v to u, then u is before v.

(So if there is no path between u and v, or if there are paths both from u to v and from v to u, then u and v can appear in any order.)

In this question you are to give an algorithm that prints out the vertices of G in an order that satisfies the above condition. Furthermore, the algorithm must runs in time $\mathcal{O}(|V| + |E|)$, where $G = (V, E)$.

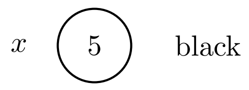
You have to first explain how the algorithm works in (a), then give the pseudo-code for the algorithm in (b). The partial marks for (a) and (b) are awarded **only if your algorithm satisfies the stated running time requirement**. You can use any procedure given in class.

(a)[5] Clearly explain how your algorithm works.

Question 2 continues here

(b)[5] Give the pseudo-code for your algorithm.

Question 3 [5] Consider the following **red-black** tree that consists of a single node x with key $x.key = 5$ whose color is black.



In this question you are to show the result of inserting nodes with keys 2,4,3 (in that order) into this tree using the insertion algorithm given in class.

For each part (a), (b), and (c) below:

- Draw the tree each time it is modified, clearly indicate the color of each node by writing either "black" or "red" next to it.
- For any rotation that you use: clearly name it (either Left-rotate or Right-rotate), state at which node it is performed, and show their effects by redrawing the tree.
- Redraw the tree if you recolor any node.

Note: The mark for (c) is only awarded if the answers for (a), (b), and (c) are correct. The mark for (b) is only awarded if the answers for (a) and (b) are correct.

(a)[1] Show how the tree drawn above changes when a node y with key $y.key = 2$ is inserted into it.

Question 3 continues here

(b)[2] Show how the tree obtained in (a) changes when a node u with key $u.key = 4$ is inserted into it.

(c)[2] Show how the tree obtained in (b) changes when a node v with key $v.key = 3$ is inserted into it.

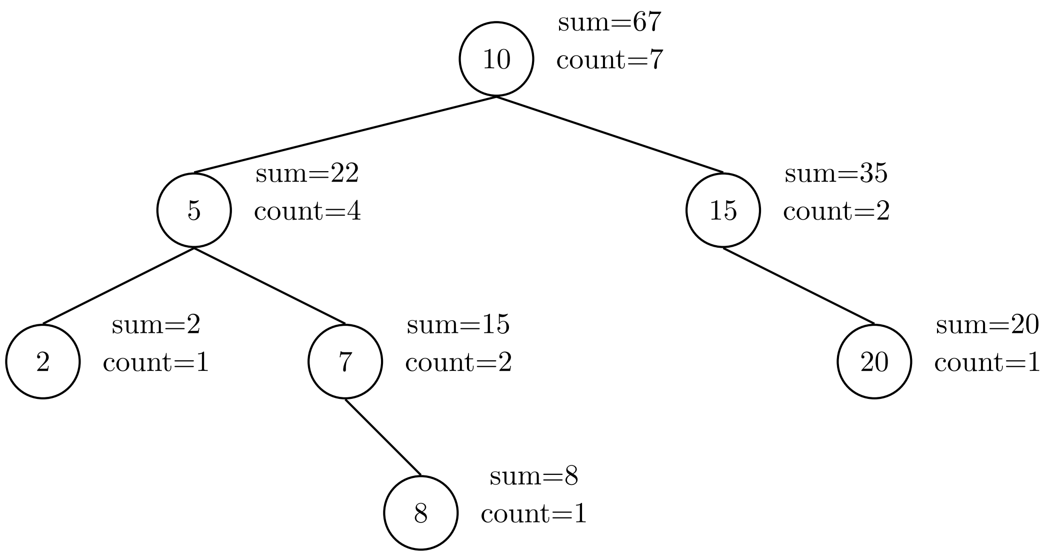
Question 4 [5] Consider the data structure for binary search tree. Recall that the (usual) fields for a node are

- **key**: An integer, the key of the current node.
- **left**: The pointer to the left child of the node. ($x.left$ is NIL if node x has no left child.)
- **right**: The pointer to the right child of the node. ($x.right$ is NIL if node x has no right child.)
- **parent**: The pointer to the parent of the node. ($x.parent$ is NIL if node x is the root of a tree.)

Now we add to each node two new fields:

- **sum**: An integer. This is the sum of all keys (including the key of the current node) in the subtree which is rooted at this node;
- **count**: An integer. This is the number of nodes (including the current node) in the subtree rooted at this node.

An example is given in the picture below. Here the number written inside each node is the key, the fields **sum**, **count** are written next to each node. (The node with key 10 is the root.)



(The new fields are useful for queries that ask for the k-th smallest element, or the average of some subset of elements, but you are **not** asked to implement these queries here. In this question you are asked only to give an algorithm that inserts a new node in a tree. The bonus question asks for an algorithm that deletes a node in a tree.)

Give pseudo-code for an algorithm Insert(T, x) that inserts a new node x into tree T. Your algorithm **must** run in time $\mathcal{O}(h)$, where h denotes the height of the tree T. Remember to update the fields **sum** and **count** for the nodes in the tree.

Question 4 continues here

Bonus question [5] (**This question is graded as either correct (5/5) or incorrect (0/5).**) Refer to the data structure given in Question 4. Give pseudo-code for the algorithm Delete(T, x) that deletes a node x from tree T. Your algorithm **must** run in time $\mathcal{O}(h)$, where h denotes the height of the tree T. Remember to update the fields **sum** and **count** for the nodes in the tree.

APPENDIX

Here is a list of some algorithms and operations on data structures given in lecture.

Note: if you want to modify any algorithm given here, you need to provide full details of the pseudo-code of the new algorithm.

Stack

- push(S, x): pushes element x to stack S.
- pop(S): pops (and returns) the top of stack S, returns NIL if S is empty.

Queue

- enqueue(Q, x): appends an element x to the end of queue Q.
- dequeue(Q): removes the first element x of Q from Q, returns x (x is NIL if Q is empty).

Linked list

- insert(L, x): inserts element x into linked list L, x becomes L's head.
- delete(L, x): removes element x from linked list L.
- search(L, k): returns an element in L with key k, or NIL if no such element exists.

Binary search tree

- BST-search(x, k): returns a node with key k in the subtree rooted at node x, or NIL if no such node exists.
- BST-min(x): returns the element with minimum key in the subtree rooted at node x.
- BST-max(x): returns the element with maximum key in the subtree rooted at node x.
- BST-successor(T, x): returns the successor of node x in binary search tree T, or NIL if x is the maximum element of T.
- BST-predecessor(T, x): returns the predecessor of node x in binary search tree T, or NIL if x is the minimum element of T.
- BST-insert(T, x): inserts node x into binary search tree T.
- BST-delete(T, x): deletes node x from binary search tree T.
- BST-Transplant(T, u, v): replaces node u by node v: u's parent becomes v's parent, and v becomes the appropriate child of u's parent. (If u is the root of T then now v is the root of T.)

Red-black tree The procedures BST-min, BST-max, BST-successor, BST-predecessor work for red-black trees just as for an ordinary binary search tree. In addition, we have:

- RB-insert(T, x): inserts node x into red-black tree T.
- RB-delete(T, x): deletes node x from red-black tree T.
- Left-rotate(T, x): rotates the link between x and its right child y to the left, making y x's parent, and x y's left child.
- Right-rotate(T, x): rotates the link between x and its left child y to the right, making y x's parent, and x y's right child.

Some algorithms on graphs

- BFS(G, s): performs breadth-first search in graph G starting from vertex s; returns
  - array d, where $d[v]$ is the distance from s to v.
  - array p, where $p[v]$ is the parent of v in the BFS tree rooted at s.
- DFS(G): performs depth-first search in graph G; returns
  - arrays s and f, where $s[v]$ and $f[v]$ are "starting time" and "finishing time" of node v, respectively.
  - array p, where $p[v]$ is the parent of v in the DFS forest.
- DFS-Visit(G, u): This is the subroutine used in DFS(G), it performs the DFS visit starting at node u.
- Acyclic(G): returns TRUE if the directed graph G is acyclic, FALSE otherwise.
- Acyclic-Visit(G, u): This is the subroutine used in the procedure Acyclic(G). It is similar to DFS-Visit, but it outputs false if a back edge is detected.
- Topological-sort(G): outputs a linked list L that contains the vertices of G in topological order if G is acyclic.
- Topo-Visit(G, u, L): This is the subroutine used in the procedure Topological-sort. It performs the DFS visit at node u; each time a node is colored black it is inserted into the linked list L.
- SCC(G): returns array SCC, where $SCC[v]$ is the identifier of the strongly connected component that node v belongs to.
- SCC-Visit1(G, u, L): This is the first subroutine used in the algorithm SCC. It performs the DFS visit at node u; each time a node is colored black it is inserted into the linked list L (this is the same as Topo-Visit(G, u, L)).
- SCC-Visit2(G, u, c, SCC): This is the second subroutine used in the algorithm SCC. It performs the DFS visit at node u. All nodes explored here belong to the strongly connected component with identifier c (so if node v is visited during this subroutine, $SCC[v]$ gets the value c).