CSC 212 Final - Fall 2013

College of Computer and Information Sciences, King Saud University
Exam Duration: 3 Hours

04/01/2014

Question 1 [10 points]

Select the most appropriate answer.

1. If we use the folding hash function $H(key) = d_1d_2 + d_3d_4$ where d_1, d_2, d_3 and d_4 are the first 4 digits of the key, then the maximum size of the hash table is:

   a) 199 b) 10000 c) 208 d) None

2. The result of evaluating the postfix expression: 2 3 4 + 1 * +, is:

   a) 10 b) 9 c) 15 d) None

3. The Big-O for the insert method in a binary tree is:

   a) $O(n)$ b) $O(\log n)$ c) $O(n \log n)$ d) $O(1)$

4. In what type of trees must all the leaf nodes be at the same depth?

   a) Binary tree b) B+ tree c) AVL d) Heap

5. The heap implementation of the priority queue compared to the linked one has:

   a) Faster serve but slower enqueue b) Slower serve but faster enqueue c) Faster serve and enqueue d) Slower serve and enqueue

6. The data structure that is appropriate to use when a FIFO (first in first out) behavior is required is:

   a) Stack b) Queue c) Hash d) Heap

7. The data structure that can be used to efficiently sort an array:

   a) Heap b) List c) Queue d) Stack

8. In which of the following strategies can the number of stored keys be larger than the table size?

   a) Coalesced chaining b) Linear rehashing with $c = 1$ c) External chaining d) Linear rehashing with $c = 2$

9. The most efficient data structure for the search operation is:

   a) Hash b) Priority queue c) AVL d) B+ tree

10. Which of the following operations of the Array List implementation may require shifting data:

    a) Retrieve   b) Insert   c) FindFirst   d) a and b

. . . .

Question 2 [16 points]

1. Write the static method *duplicate* that takes as input a non-empty list l and duplicates each element of the list putting the duplicate right after the original. The method signature is: *public static <T >void duplicate(List<T >l)*.

   Example 2.1. *If* $l : A \to A \to B \to C \to B$*, then after the call* duplicate(l)*,* l *becomes:* $A \to A \to A \to A \to B \to B \to C \to C \to B \to B$.

2. Write the method *elncr*, member of the class *LinkedList*, that returns the element having the largest number of **contiguous** repetitions. In case of a tie, the first element to appear in the list is returned. Use the method *equals* to test for equality. Do not call any method other than *equals* and do not use any auxiliary data structure. The method signature is *public T elncr()*.

   Example 2.2. *If* $l : A \to A \to B \to B \to B \to A \to C \to C \to C \to B \to A \to A \to D \to E$*, then the call* l.elncr() *returns* B.

. . . .

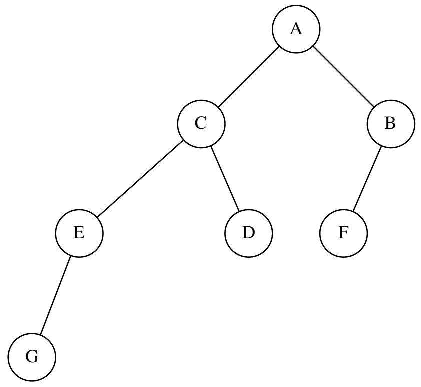
Question 3 [16 points]

1. Write the **recursive** method *nbNonLeaf*, a private member method of the class *BT* (binary tree) that takes as input a node t and returns the number of non-leaf nodes in the subtree rooted at t. The method signature is: *private int nbNonLeaf(BTNode <T >t)*.

2. Write the **recursive** method *nbOneChild*, a private member method of the class *BT* (binary tree) that takes as input a node t and returns the number of nodes in the subtree rooted at t having exactly one child. The method signature is: *private int nbOneChild(BTNode <T >t)*.

   Example 3.1. *In the tree in front, the call to* nbNonLeaf *with the node containing data C as parameter returns 2, whereas the call with the node containing data A as parameter returns 4.*

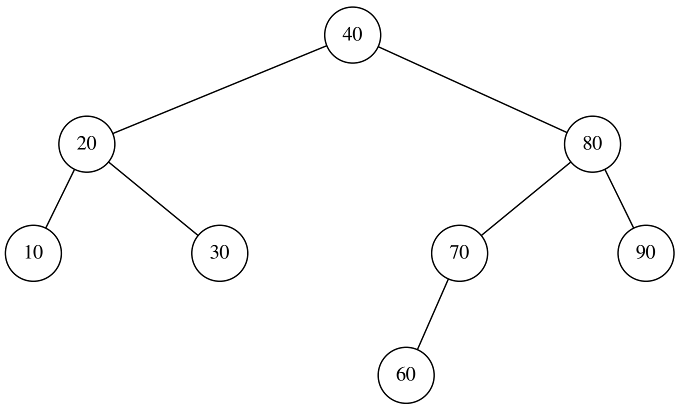
   Example 3.2. *In the tree in front, the call to* nbOneChild *with the node containing data C as parameter returns 1, whereas the call with the node containing data A as parameter returns 2.*



. . . .

Question 4 [12 points]

Perform the following operations on the AVL tree below: (1) **Insert 65**. (2) **Insert 25**. (3) **Delete 80**. Each operation is **independent** of the others and must be performed on the original tree. Make sure to mention the rotation performed (none, single, double).



. . . .

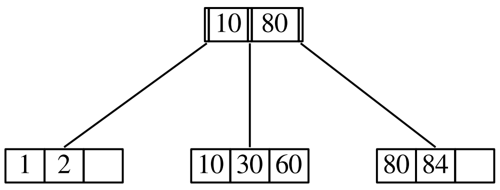
Question 5 [12 points]

1. Insert the following keys in a min-heap: 8, 1, 12, 10, 5, 18 and 1. The heap is initially empty. Insert the keys from left to right. Draw the heap **after** inserting **all keys**.
2. Delete two keys from the previous heap. Draw the heap **after** deleting **all keys**.

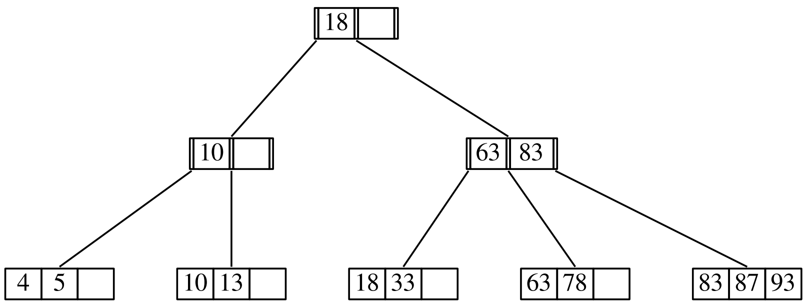
. . . .

Question 6 [16 points]

1. Insert the keys 15, 7, 90 and 75 in the B+ tree ($m = 3$) shown below. Draw the tree **after every** operation.



2. Delete keys 78 and 4 from the B+ tree ($m = 3$) shown below. Draw the tree **after every** operation.



. . . .

Question 7 [12 points]

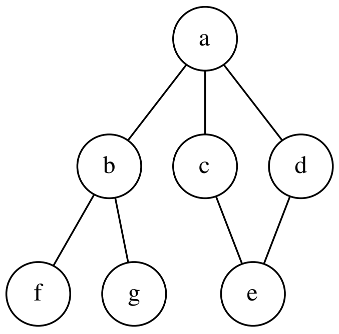
Use the hash function $H(key) = key \mod 7$ and store the keys: 13, 9, 16, 26, 23, 17 and 24 in a hash table using the following techniques (draw the table **after** inserting **all keys**):

1. Linear rehashing with c = 1.
2. External chaining.
3. Coalesced chaining with cellar size 3. Show clearly the links and the final position of the *epla*.

. . . .

Question 8 [6 points]

Draw the adjacency matrix and adjacency list for the following graph (order the nodes alphabetically):



. . . .

A ADT List specification

- FindFirst ( ): **requires**: list L is not empty. **input**: none. **results**: first element set as the current element. **output**: none.
- FindNext ( ): **requires**: list L is not empty. Current is not last. **input**: none. **results**: element following the current element is made the current element. **output**: none.
- Retrieve (Type e): **requires**: list L is not empty. **input**: none. **results**: current element is copied into e. **output**: element e.
- Update (Type e): **requires**: list L is not empty. **input**: e. **results**: the element e is copied into the current node. **output**: none.
- Insert (Type e): **requires**: list L is not full. **input**: e. **results**: a new node containing element e is created and inserted after the current element in the list. The new element e is made the current element. If the list is empty e is also made the head element. **output**: none.
- Remove ( ): **requires**: list L is not empty. **input**: none. **results**: the current element is removed from the list. If the resulting list is empty current is set to NULL. If successor of the deleted element exists it is made the new current element otherwise first element is made the new current element. **output**: none.
- Full (boolean flag): **requires**: none. **input**: none. **results**: if the number of elements in L has reached the maximum number allowed then flag is set to true otherwise false. **output**: flag.
- Empty (boolean flag): **requires**: none. **input**: none. **results**: if the number of elements in L is zero, then flag is set to true otherwise false. **output**: flag.
- Last (boolean flag): **requires**: L is not empty. **input**: none. **results**: if the last element is the current element then flag is set to true otherwise false. **output**: flag.