The National Higher School of Artificial Intelligence
Data Structures and Algorithms 2 (12/10/2023)
Midterm Exam Solutions
Duration: 1 hour 40 minutes

Exercise 1 (06 marks)

A. Assume that the following code array a contains *n* values, that the method randomValue takes constant number *c* of computational steps to produce each output value, and that the method goodSort takes *n log n* computational steps to sort the array. Determine the Big-Oh complexity for the following fragments of code taking into account only the above computational steps:

```
for( i = 0; i < n; i++ ) {
    for( j = 0; j < n; j++ )
        a[ j ] = randomValue( i );
    goodSort( a );
}
```

Answer: 2 marks
The innermost loop has linear complexity *c n*, but the next called method goodSort is of higher complexity *n log n*. Because the outermost loop is linear in *n*, the overall complexity of this piece of code is n^2 *log n*.

B. Giving a brief but precise explanation, say what the computational complexity of the following piece of code is:

```
for ( i=1; i < n; i *= 2 ) {
    for ( j = n; j > 0; j /= 2 ) {
        for ( k = j; k < n; k += 2 ) {
            sum += (i + j * k );
        }
    }
}
```

Answer: 2 marks
The running time of the innermost, middle, and outermost loop is proportional to n, log n, and log n, respectively. Thus the overall Big-Oh complexity is $O(n (\log n)^2)$.

C. Derive the recurrence that describes processing time T(n) of the recursive method:

```
public static int recurrentMethod( int[ ] a, int low, int high, int goal ) {
    int target = arrangeTarget( a, low, high );
    if ( goal < target )
        return recurrentMethod( a, low, target-1, goal );
    else if ( goal > target )
            return recurrentMethod( a, target+1, high, goal );
      else
            return a[ target ];
}
```

The range of input variables is $0 \le \text{low} \le \text{goal} \le \text{high} \le \text{a.length} - 1$. Non-recursive method arrangeTarget() has linear time complexity $T(n) = c \cdot n$ where $n = \text{high} - \text{low} + 1$ and returns integer target in the range $\text{low} \le \text{target} \le \text{high}$. The output values of arrangeTarget() are equiprobable in this range, e.g. if low = 0 and high = n − 1, then every target = 0, 1, . . . , n − 1 occurs with the same probability (1/n). The time for performing elementary operations like if–else or return should not be taken into account in the recurrence.

Hint: consider a call of recurrentMethod() for *n* = a.length data items, e.g. recurrentMethod(a, 0, a.length - 1, goal) and analyse all recurrences for T(n) for different input arrays. Each recurrence involves the number of data items the method recursively calls to.

Answer: 02 marks

Using the hint, we can write:

$T(n) = T(0) + c.n$ or $T(n) = T(n - 1) + c.n$ if target = 0

$T(n) = T(1) + c.n$ or $T(n) = T(n - 2) + c.n$ if target = 1

$T(n) = T(2) + c.n$ or $T(n) = T(n - 3) + c.n$ if target = 2

. . . .

$T(n) = T(n - 1) + c.n$ or $T(n) = T(0) + c.n$ if target = n − 1

Summing up, we get $n.T(n) = (T(0) + . . . + T(n - 1)) + n.(c\cdot n)$

And because all the variants are equiprobable, the recurrence is

$$T(n) = (1/n)\,(T(0) + . . . + T(n - 1)) + c\cdot n$$

Exercise 2 (6 marks)

1) Write algorithms to enqueue, dequeue, and search for an element in the queue of this data structure implemented using a Linked List **(01 point for each operation)**.

```
#include <iostream>
using namespace std;
struct Node { //(0.5 point) for the well proposed data structure.
    int data;
    int team;
    Node* next;
    Node(int data, int team) {
        this->data = data;
        this->team = team;
        this->next = NULL;
    }
};
class TeamQueue {
private:
    Node* head;
    Node* tail;
public:
    TeamQueue() {
        head = NULL;
        tail = NULL;
    }
/*(01 point) for the extra necessary function. only for students who
have distinguished between searchElement and searchTeam*/
void Enqueue(int element, int team) {
    Node* newNode = new Node(element, team);
    if (head == NULL) {
        // Queue is empty, add the first element
        //0.25 point
        head = newNode;
        tail = newNode;
    } else {
        // Search for teammates and enqueue accordingly
        //0.75 point
        Node* current = head;
        while (current && current->team != team) {
            current = current->next;
        }
        if(current) {
            // Teammates are present, find the end of the team
            //0.25 point
            while (current->next && current->next->team == team) {
                current = current->next;
            }
            // Insert after the last teammate
            // 0.25 point
            newNode->next = current->next;
            current->next = newNode;
```

```
                    if (current == tail) {
                        // Update tail if the last element is affected
                        // 0.25 point
                        tail = newNode;
                    }

                } else {
                    // No teammates found, insert at the end
                    //0.25 point
                    tail->next = newNode;
                    tail = newNode;
                }
            }

            void Dequeue() {
                if (head == NULL) {
                    cout << "Team queue is empty." << endl;
                    return;
                }
                int element = head->data;
                Node* temp = head;
                head = head->next;
                delete temp;
            }

            bool searchElement(int element) {
                Node* current = head;
                while (current != NULL) {
                    if (current->data == element) {
                        return true;
                    }
                    current = current->next;
                }
                return false;
            }
        };
```

2) Give a brief explanation to say what is the Big-Oh complexity of each of the 3 operations of the previous question.

Enqueue: (0.5 point)

The complexity of the Enqueue operation depends on the location where the element is inserted. In the worst case, it might have to traverse the entire team or the entire queue. Big-O Complexity is then: O(n), where n is the number of elements in the queue.

Dequeue: (0.5 point)

The Dequeue operation involves removing the element from the front of the queue. This operation has a constant time complexity since it only involves updating pointers and one call for delete. So Big-O Complexity: O(1).

Search for an element in the Teams Queue: (0.5 point)

The Search operation involves traversing the linked list to find a specific element. In the worst case, it might have to traverse the entire linked list.
Big-O Complexity: O(n), where n is the number of elements in the queue.

Exercise 3 (8 marks)

A) Suppose you are given the following set of values to be inserted into a Binary Search Tree of alphabetic letters: 'a','b','c','d','e','f', 'g'

i)- In what order would you insert these values so as to get the worst-case runtime when searching for an element in the tree after its creation? Why?

Order of insertion : a,b,c,d,e,f,g (0.5 mark)

The nodes are inserted in increasing order of the values. This order ensures that each new node is added to the right of the previous one, forming a linked list. The resulting tree of n nodes will have a height equal to n-1making the search operation O(n) in the worst case. **(0.5 mark)**

(Other possible order : g, f,e,d,c ,b, a "decreasing order")

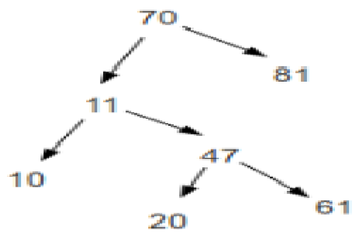
ii)- In what order would you insert these values so as to get the best-case runtime when searching for an element in the tree after its creation? Why?

Order of insertion : d,b,f, a, c,e, g **(0.5 mark)**

This order ensures that at each step, the median element is chosen as the root. Thus , the insertion results in a balanced tree where, at each level, the number of nodes on the left and right subtrees is roughly equal. The resulting tree will have a height minimized of approximately log(n), where n is the number of nodes in the tree. Therefore, the search operation will have a best-case runtime of O(log n) **(0.5 mark)**

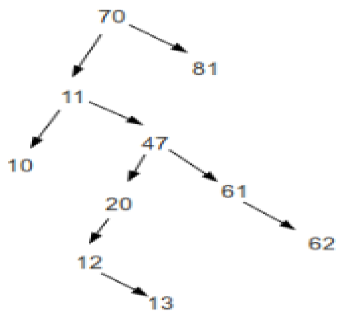
(we can have permutation in the order between (f,b); (a,c); (e;g))

B) Consider the following binary search tree (BST) which stores positive integer values.



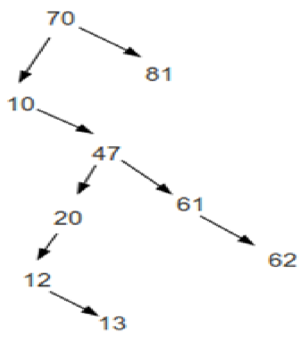
1)- Draw the same BST after inserting the numbers 12,13 and 62 in this order.

BST after insertion of 12,13, 62 **(1mark)**



2)- Redraw the tree after deleting 11.

BST after deletion of 11 **(1 mark)**



3)- Given the partial definition below of a Binary Search Tree, what does the function FBST() do on a non-empty Binary Search Tree?

```
struct BSTree{
int FBST() {
        return FBST( root) -> data ;}
….
private :
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
} ;
TreeNode * root ;
TreeNode * FBST( TreeNode *t )  {
        if( t == nullptr )
                  return nullptr;
        if( t->left == nullptr )
            return t;
        return FBST( t->left );
    }
} ;
```

The function FBST() returns the smallest element in the tree by calling the private function FBST witch returns the smallest node by going all the way to the left. **(1 mark)**

4)- Suppose we want to add the operation *Successor* to the interface of « BSTree ». **(2 marks)**

i) Write a function that returns the successor of a node in a Binary Search Tree. Assume that all the keys are distinct, and a successor of a node *n* is defined as the node with the smallest key *x* in the BST such that *x* is bigger than the value of *n* or *null* if that successor does not exist. For instance, in the above BST the successor of Node 20 is Node 47.

```
  TreeNode* Successor(int target) {
   TreeNode* successor = nullptr;
   TreeNode*   t= root;
   bool exist = false;
      while (t) {
          if (t->data > target) {
              successor =  t;
              t = t->left;
          }
          else
           if (t->data   <  target) {
                 t = t->right;  }
           else {
              // match
              exist = true; //target exist in BST
              if (t->right) {
                  successor = FBST(t->right);}
              break;
           }
      }
  if (! exist) {successor = nullptr;}
  return successor;
  }
```

ii) Giving a brief explanation, estimate the asymptotic running time of your function.

The asymptotic running time of this function is O(h), where h is the height of the BST. In a balanced BST, the height is O(log n), where n is the number of nodes. In the worst case of an unbalanced BST, the height can be O(n), which results in a linear time complexity. **(1 mark)**