Middle East Technical University
Department of Computer Engineering

CENG 213 Data Structures
Final Exam
January 11, 2013
120 min.

Name: KEY

Section: ___________

Q1 (15 pts.)	
Q2 (15 pts.)	
Q3 (15 pts.)	
Q4 (15 pts.)	
Q5 (17 pts.)	
Q6 (23 pts.)	
TOTAL	

Write your answers clearly into the boxes provided only.

Q1. (15 pts.) Time Complexity Analysis

What is the worst case running time (big-O notation) of each of the following operations? (1.5 pts. each)

1. Inserting a new element into an unsorted linked list of *n* elements. *Answer:* O(1)

2. Inserting a new element into a sorted linked list of *n* elements. *Answer:* O(n)

3. Inserting *n* new elements into an empty heap. *Answer:* O(n.logn) *

4. Inserting *n* new elements into an empty binary search tree. *Answer:* $O(n^2)$

5. Finding the maximum element in a complete binary search tree of *n* elements. *Answer:* O(logn)

6. Sorting an already sorted array of *n* integers using bubble sort algorithm. *Answer:* O(n)

7. Sorting an already sorted array of *n* integers using heapsort algorithm. *Answer:* O(n.logn)

8. Deleting an integer from an AVL tree of *n* integers. *Answer:* O(logn)

9. Searching a sorted array of n integers for a particular value using binary search. Answer: O(logn)

10. Printing a full binary tree of *n* elements using inorder traversal. *Answer:* O(n)

* O(n) is also acceptable (assuming "buildHeap" method is used)

Items penalized once for the whole question:

- **Missing "O(…)" around formulas : -0.5 points**
- **Abbreviations (like "lg" for "log") : -0.5 points**

Q2. (15 pts.) (Heap means min heap in this question)

a) What is the smallest key that could be at the leaf level of a heap built from a random order of the keys **1** through **16**? (3 pts.)

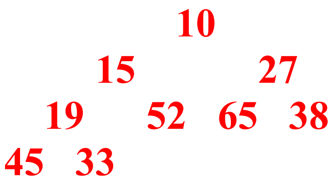
4

b) Insert the following items into an empty heap one by one in the given order. Show the array and tree-structure drawing of the resulting heap: **45, 38, 27, 33, 52, 65, 19, 10, 15** (4 pts.)

Array:

10	15	27	19	52	65	38	45	33

Tree:



c) Draw the heap that would result from the `buildHeap` operation on the initial array given below: (4 pts.)

90	**47**	**75**	**82**	**50**	**8**	**21**	**15**	**31**

(Recall that the buildHeap algorithm builds a heap from bottom up through a sequence of percolateDown operations.)
Resulting heap:

8	15	21	31	50	75	90	82	47

d) Consider the following heap: (4 pts.)

7	**11**	**9**	**18**	**12**	**23**	**15**	**20**

You are going to apply *two* `deleteMin()` operations on this heap. Show the array contents after each `deleteMin`.

9	11	15	18	12	23	20	
11	12	15	18	20	23		

Q3. (15 pts)

a) Consider a hash table of size **11** and the hash function ***h*(*x*) = *x* mod 11**. Suppose the following keys are inserted (in the order shown) into an empty hash table:

```
51, 38, 72, 22, 27, 36, 18, 49
x mod 11:
(7) (5) (6) (0) (5) (3) (7) (5)
```

Draw the resulting hash table, if the collision resolution scheme is (10 pts.)

i. separate chaining

ii. open addressing with *quadratic probing*

Answer to part (i)
Separate chaining:

0		22
1		
2		
3		36
4		
5		49, 27, 38
6		72
7		18, 51
8		
9		
10		

Answer to part (ii)
Quadratic probing:

0	22
1	
2	
3	36
4	
5	38
6	72
7	51
8	18
9	27
10	49

b) Considering your answer to part **(ii)** of **(a)** only, i.e. hash table created using quadratic probing:

i. What is the load factor of the table? (1 pts.)

8/11

ii. What is the average number of probes in a successful search? (2 pts.)

15/8

iii. What is the average number of probes in an unsuccessful search? (2 pts.)

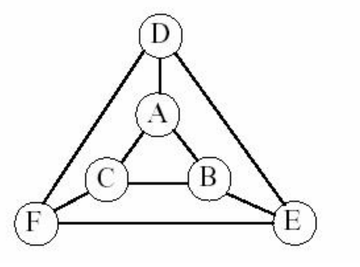
Quadratic probing causes an infinite number of probing if the data is hashed to position 5. Assuming this case needs 11 probes:

2+1+1+2+1+11+4+5+3+3+5= 38/11

(I accepted other assumptions if they are logical)

Q4. (15 pts)

a) Write the results of the depth-first and breadth-first traversals of the given graph starting at node A. If several nodes can be chosen at some step, pick the one alphabetically first.



Depth-first traversal	A-B-C-F-D-E	+2 pts.
Breadth-first traversal	A-B-C-D-E-F	+2 pts.

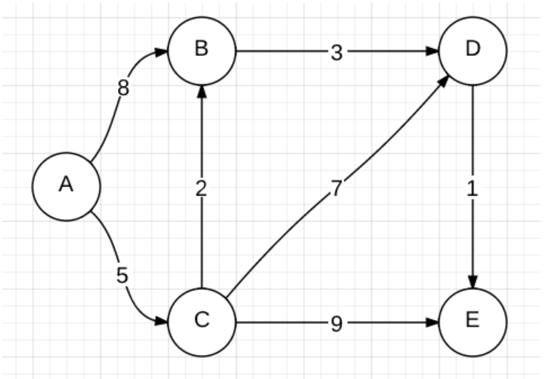
b) Given the iterations of the Dijsktra's algorithm for a directed graph with positive edge weights:

i. Fill in the blank cells in the given iterations table on the right. +3.5 pts.

ii. Write down the shortest path found from A to E.

A-C-B-D-E	+2.5 pts.

iii. Draw the graph for which these iterations are given.



+5 pts.

Iteration 1					
Node	A	B	C	D	E
Distance	0	8	5	∞	∞
Path	-	A	A	-	-
Iteration 2					
Node	A	B	C	D	E
Distance	0	7	5	12	14
Path	-	C	A	C	C
Iteration 3					
Node	A	B	C	D	E
Distance	0	7	5	10	14
Path	-	C	A	B	C
Iteration 4					
Node	A	B	C	D	E
Distance	0	7	5	10	11
Path	-	C	A	B	D
Iteration 5					
No change. *Same as Iteration 4.*					

In a and b.ii, if 1-2 nodes missing/misplaced -1

Q5. (17 pts)

Given the following definition:

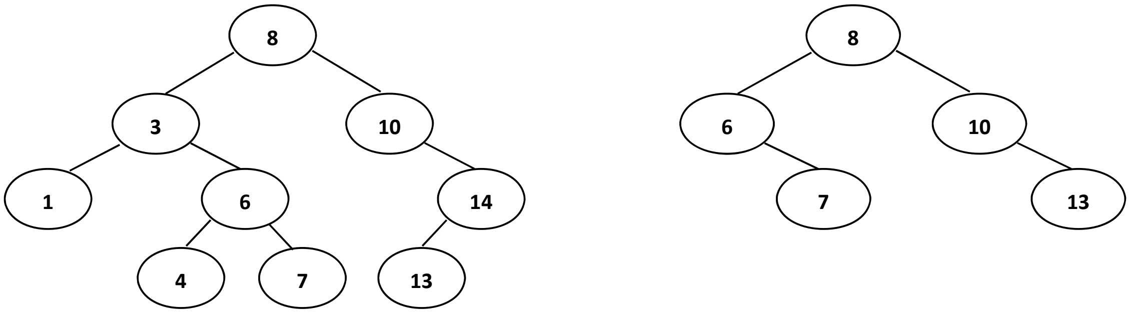
```
    struct TreeNode {
       int item;
       TreeNode *left;
       TreeNode *right;
    }
```

a) Complete the recursive C++ function below to delete all the nodes of the given tree `t`. **You are NOT allowed to declare any other identifier or to use a looping statement.**

```
void delTree(TreeNode* &t) {
  if(!t) return;
   delTree(t->left);
   delTree(t->right);
   delete t;
   t=NULL;
}
```

recursive call on left	+1 pts
recursive call on right	+1 pts
delete out node	+1 pts
NULL assignment	+1 pts

b) Given the root of a binary search tree **`t`** and two positive integer numbers **`min`** and **`max`**, trim the tree such that all the numbers in the new tree are between min and max (inclusive). The resulting tree should still be a valid binary search tree. Thus, if we get the tree on the left as input and we are given **`min`** value as 5 and **`max`** value as 13, then the resulting binary search tree should be the one on the right after removing all the nodes whose value is not between **`min`** and **`max`**:

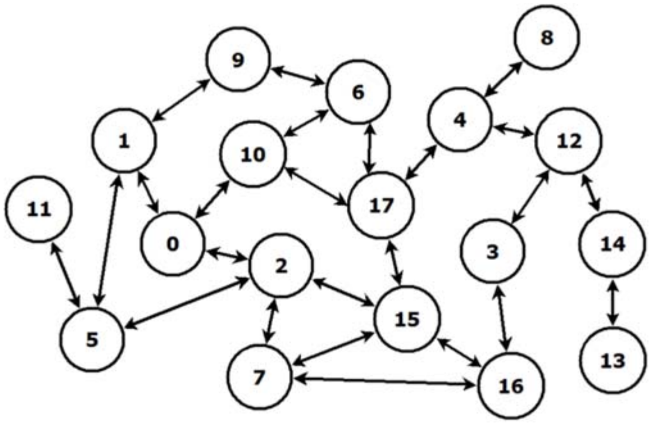


Complete the following recursive C++ function to perform the trim operation explained above. You can use the function developed in part (a). **You are NOT allowed to declare any other identifier or to use a looping statement.**

```
void trimBST(TreeNode* &t, int min, int max) {
   TreeNode *toDelete=t;

    if(!t) return;                                        +1 pts

// if this item is less than min
   if(t->item < min) {                                    +2 pts
     t=t->right;
     delTree(toDelete->left);
     delete toDelete;
     trimBST(t, min, max);
   }
                                                   +1 pts / each

// else if this item is greater than max
   else if(t->item > max) {                               +2 pts
     t=t->left;
     delTree(toDelete->right);
     delete toDelete;
     trimBST(t, min, max);
   }
                                                   +1 pts / each

// else --that is, if this item is in the range [min, max]
   else {
      trimBST(t->left , min, max);
      trimBST(t->right, min, max);
   }
}
```

Q6. (23 pts) Finding a Path Between Two Vertices

Given an unweighted, directed, and strongly connected graph, we will provide code for two functions in this question: the first function computes a path (any path is OK, no need for the shortest path) from a given starting vertex ***s*** to a destination vertex ***d***. And the second function prints the path from ***s*** to the destination vertex ***d***. The data structure that holds a single vertex information is defined below. A global array **V** will contain pointers to all of the properly constructed vertices in the graph. Also, suppose that the constant **NUMV** represents the number of vertices in the graph (vertex IDs will be between 0 and NUMV-1, inclusive). For this question, you don't need to worry about or handle any exceptions that might be thrown in your code. Don't define any variables of your own. Don't define any helper methods or new data types. Don't use recursion. A sample main function, and its output is also provided below.



```
class Vertex
{
  public:
    int    id           ;  // vertex ID (also, index of this vertex in the global array; V)
    int    numNeighbors ;  // number of adjacent vertices
    int  * neighborIds  ;  // an array of IDs of adjacent vertices
    bool   visited      ;  // temporary variable used in path finding
    int    path         ;  // temporary variable used in path finding and path printing
} ;
```

For example, a pointer ***p*** that points to vertex 6 in the given figure will contain the following information:

```
p->id           = 6
p->numNeighbors = 3
p->neighborIds  = {9, 10, 17}
p->visited      = <uninitialized>
p->path         = <uninitialized>
```

Sample main:

```
void main ( void ) {
  computePathBetween( V[3] , V[9] ) ;
  printPathTo(9)                    ;
}
```

One possible output for sample main:

```
3 12 4 17 6 9
```

Important Stack ADT Operations:

```
isEmpty () : boolean
push    ( in  newItem  : StackItemType )
pop     ( out stackTop : StackItemType )
getTop  ( out stackTop : StackItemType )
```

Important Queue ADT Operations:

```
isEmpty  () : boolean
enqueue  ( in  newItem    : QueueItemType )
dequeue  ( out queueFront : QueueItemType )
getFront ( out queueFront : QueueItemType )
```

NOTE: If there are syntax errors, or the code is not valid (for example, API call conventions are not followed), then just the Logic of the code is graded for the marked sections of the solution below.

```
     #define  NUMV  (18)            // Number of vertices

     Vertex * V[NUMV] = { ... } ;  // Global array properly filled with pointers to vertices

     void computePathBetween ( Vertex * s , Vertex * d ) {
       int             i      ;
       Vertex         *v , *w ;
       Queue<Vertex*>  q      ;

       // Don't forget to mark all vertices as unvisited first
                                                                          Mark unvisited
2        for ( i = 0 ; i < NUMV ; i++ ) { V[i]->visited = false ; }        Logic: 1

1+1      s->visited = true ;      Fill info for s
         s->path    = -1   ;

1        q.enqueue( s ) ;         Enqueue s


         while ( ! q.isEmpty() )
         {
1          q.dequeue( v ) ;       Dequeue

                                      Stop if found (for efficiency)
2          if ( v == d )  { break ; }  Logic: 1

1          for ( i = 0 ; i < v->numNeighbors ; i++ )
           {
1            w = V[ v->neighborIds[i] ] ;                 - Loop over neighbors

2            if ( w->visited == false )                   - Fill info for and
             {                                            enqueue unvisited
1+1            w->visited = true  ;                       neighbors.
               w->path    = v->id ;
                                                          Logic: 4
1              q.enqueue( w ) ;
             }
           }

         }
     }

     void printPathTo ( int d ) {
       Stack<int> s ;


1        s.push( d ) ;

1        while ( V[d]->path != -1 )       Push path information onto stack
         {
1+1        s.push( V[d]->path ) ;         Logic: 2
           d = V[d]->path       ;
         }


1        while ( ! s.isEmpty() )          Pop & print until stack is empty
         {
2+1        s.pop( d )      ;              Logic: 2
           cout << d << " " ;
         }


     }
```