METU Department of Computer Engineering
CENG 213 Data Structures
Midterm I
November 15, 2012
110 min.

ID: _____

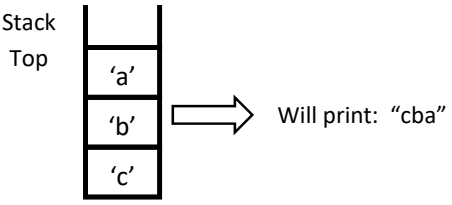
Name: _____

Section: _____

Q1 (18 pts)	
Q2 (20 pts)	
Q3 (18 pts)	
Q4 (26 pts)	
Q5 (18 pts)	
TOTAL	

Q1. (18 pts) Consider the following public `Stack` interface:

```
#include "StackException.h"
template <class StackItemType>
class Stack {
public:
   Stack();
   Stack(const Stack& aStack);
   ~Stack();
   bool isEmpty() const;
   void push(StackItemType newItem);
   void pop() throw(StackException);
   void topAndPop(StackItemType& stackTop) throw(StackException);
   void getTop(StackItemType& stackTop) const throw(StackException);
...
}
```



Write a **recursive** C++ function **`PrintBackwards()`** which prints the contents of a given stack **`S`** in reverse order. The function should not declare or use any other stacks. The function may modify the contents of the given stack during its execution, but when its task is finished, the stack should contain the original elements in their original order. The function should use only the given stack interface.

```
template <class StackItemType>
void PrintBackwards(Stack<StackItemType> & S) {

if ( S.isEmpty() )      // Providing a base case for recursion              (3 pts)
{
  return ;
}

StackItemType temp ;    // Declaring a variable to store the top element    (2 pts)

try                     // Handling possible exceptions, any handler is OK  (2 pts)
{
  S.topAndPop( temp ) ; // Popping & storing top element                    (2 pts)
}
catch ( StackException & e )
{
  cout << "topAndPop: Exception occurred" << endl ;
}

PrintBackwards( S ) ;   // Calling recursively to print rest of the stack   (3 pts)

cout << temp ;          // Printing top element using overloaded << operator (3 pts)

S.push( temp ) ;        // Restoring given stack back to original state     (3 pts)

// If a working but irrelevant (e.g., non-recursive) answer is given, then 5 pts.
// If the logic is wrong, then 1 pt for each portion of the correct algorithm above.

}
```

Q2. (20 pts)

a) What is the **worst-case** running time of each of the code fragments? Give your answers using the *Big-Oh* notation.

Fragment	Answer

`i=1;`
`while (i<n*n)`
`  i = 2*i;`
	$O(\log_2 n)$

`i=1;`
`while (i<n*n)`
`  i = i+(n/2);`
	$O(n)$

`i=1;`
`while (i<n*n) {`
`  j = 1;`
`  while (j<n)`
`  j=j+1;`
`  i=2*i;`
`}`
	$O(n \log_2 n)$

`x = 1;`
`for (i = 0; i <= n-1; i++) {`
`  for (j = 1; j <= x; j++)`
`  cout << j << endl;`
`  x = x * 2;`
`}`
	$O(2^n)$

b) Give the **worst-case** complexity of the best algorithm for following operations in *Big-Oh* notation.

Operation	Answer
Deleting a node from a singly linked list when a pointer to that node is given.	$O(n)$
Deleting a node from a doubly linked list when a pointer to that node is given.	$O(1)$
Inserting a new item at the end of a singly linked list without any tail pointer.	$O(n)$
Searching an item in an unsorted array.	$O(n)$
Searching an item in a sorted array.	$O(\log_2 n)$
Sorting an array of n integers using key comparisons.	$O(n \log_2 n)$

Q3. (18 pts)

a) Consider the following sorting function:

```
void MySort(int a[], int n) {
   int t;
   bool flag = true;

   for (int i = 1; (i < n) && flag; i++) {
      flag = false;
      for (int j=0; j < n-i; j++)
         if (a[j] > a[j+1]){     ///// MARKED
            t = a[j];
            a[j] = a[j+1];
            a[j+1] = t;
            flag = true;
         }
    }
}
```

When this function is called with the integer array **a** = {2, 7, 3, 4, 6, 10, 9, 8, 7, 6} and **n** = 10;

i. How many times does the outer loop iterate?

Number of iterations: ___**6**_____

ii. How many times does the marked and underlined statement execute when the loop counter "**i**" is 2?

Number of executions: ____**8**_____(# of swaps = 3)_____

iii. How many times does the marked and underlined statement execute when the loop counter "**i**" is 4?

Number of executions: _____**6**____(#of swaps = 1)_____

b) Consider the following partition function which is used in quicksort:

```
void partition(int theArray[], int first, int last, int &pivotIndex) {
   int pivot = theArray[first];
   int lastS1 = first;
   int firstUnknown = first + 1;

   for (; firstUnknown <= last; ++firstUnknown)
     if (theArray[firstUnknown] < pivot) {
         ++lastS1;
         swap(theArray[firstUnknown], theArray[lastS1]);    ///// MARKED
     }
   swap(theArray[first], theArray[lastS1]);
   pivotIndex = lastS1;
}

void quicksort(int theArray[], int first, int last) {
   int pivotIndex;
   if (first < last) {
      partition(theArray, first, last, pivotIndex);
      quicksort(theArray, first, pivotIndex-1);
      quicksort(theArray, pivotIndex+1, last);
   }
}
```

i. When `partition` function is called at some instance of quicksort with the integer array **a** = {6, 7, 4, 9, 2, 3, 1, 5, 8, 10} and **`first`** = 0, **`last`** = 9; how many times does the marked and underlined statement execute? Show the contents of the array after this call.

Number of Executions: ___**5**_____ __

Array: ____**5 4 2 3 1 6 7 9 8 10**_____

ii. Give an array with five elements (1,2,3,4, and 5) such that it will be a ***worst case*** example for ***the version of quicksort in this question***. What is the number of key comparisons at this worst case?

Array: __**1 2 3 4 5** _____

Number of Key Comparisons: ___**10**_____

Q4. (26 pts) Assume that you are given implementations of a linked-list ADT with a dummy (sentinel) head node, where each proper (non-sentinel) node keeps a single element of the list, and also its iterator ADT. They are summarized as follows:

```
template <typename T> class iterator:
```
`T& retrieve() ....` Returns a reference to the object stored at the node pointed by this iterator.
`void advance() ...` Moves this iterator to the next node; if no such node exists, it becomes a NULL iterator.
`bool isValid() ...` Returns true if it points to a node; returns false if it is a NULL iterator.

```
template <typename T> class List:
```
`iterator<T> zeroth() ...` Returns an iterator for the dummy head node in the list.
`iterator<T> first() ....` Returns an iterator for the first node (after the dummy head node) in the list; if no such node, it returns a NULL iterator.
`void insert(T x, iterator<T> i) ...` Inserts a node with element **x** after the node pointed by **i**.

Complete the following function which takes two sorted linked-lists, merges them and returns the resulting list.
You may add code only into the proper boxes provided. You are not allowed to declare any other variables..!

```
// Merge L1 and L2 and return the resulting list
template <typename T>
List<T> merge(List<T>& L1, List<T>& L2){
    List<T> r;
    iterator<T> i1=L1.first(), i2=L2.first(), ri=r.zeroth();
    // While both lists have more elements                          +3
    while (i1.isValid() && i2.isValid()) {                          +2
          if (i1.retrieve() < i2.retrieve()) {                      +2
              l3.insert(i1.retrieve(),ri);                          +2
              i1.advance();                                         +1
              ri.advance();                                         +1
          }
          else {                                                    +1
              l3.insert(i2.retrieve(),i3);                          +2
              i2.advance();                                         +1
              i3.advance();                                         +1
          }

    }
    // Process the remaning elements of L1, if any
     while(i1.isValid()) {                                          +1
          l3.insert(i1.retrieve(),i3);                              +2
          i1.advance();                                             +1
          i3.advance();                                             +1
     }

    // Process the remaning elements of L2, if any
     while(i2.isValid()) {                                          +1
          l3.insert(i2.retrieve(),i3);                              +2
          i2.advance();                                             +1
          i3.advance();                                             +1
     }

    // Return the result
    return r;
}
```

Q5. (18 pts) Consider a linked-list structure with no dummy (sentinel) node, containing nodes defined as:

```
template <typename T>
struct Node {
    T data;
    Node *next;
};
```

Complete the following function that removes the nodes with duplicate elements in a sorted linked-list. The first node of the list is pointed by the parameter **p**. For example, the list [3, 4, 4, 4, 4, 6, 7, 7, 8] becomes [3, 4, 6, 7, 8] after calling this function. You may add code only into the proper boxes provided. You are not allowed to declare any other variables..!

```
template <typename T>
void removeDups(Node<T> *p) {
    Node<T> *q;
    // If no elements, do nothing                    +2
    if (!p) return;

    // Traverse the list
    while( p->next ) {                               +2
        // Compare current node with next
        if (p->data == p->next->data) {              +4
            // Delete the dups: Update links & free
            q = p->next->next;                       +2
            free(p->next);                           +3
            p->next = q;                             +5
        }
        else
            // Advance only if not a dup
            p = p->next;                             +2
    }
}
```

Deleting wrong nodes -4
Syntax -1
Bad formation of loop + recursion -2
Link broken -3
Variable declared → **Ignore statements that contain these variables**