CS302 – Data Structures using C++

Midterm Exam

Instructor: Dr. Kostas Alexis | Teaching Assistants: Shehryar Khattak, Mustafa Solmaz
Semester: Fall 2018
Date: November 7 2018

Student First Name _____ **Student Last Name** _____
Student NSHE ID _____ **Student E-mail** _____

Grade [XYZ/110]:

- Q1: _____ (/ 25%)
- Q2: _____ (/ 25%)
- Q3: _____ (/ 25%)
- Q4: _____ (/ 25%)
- Q5: _____ (/ 10%)

Final Grade:

Note: Maximum effective grade for this exam is 100. If your grade, say g is greater than 100 then the remaining value (g-100) will be transferred to be added at your final exam.

Question 1 [Topic: Recursion] [Percentage: 25%]: The "Towers of Hanoi" is a mathematical puzzle where one has three pegs and n disks and the goal is to move the entire stack of disks to another rod, obeying the following rules:

1. Only one disk may be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack: a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.

Write a method/function that uses recursion to solve the problem. Your method should use the following input parameters:

1. **int** n: the number of sisks
2. **char** from_rod: name of the source rod (e.g., 'A')
3. **char** to_rod: name of the destination rod (e.g., 'C')
4. **char** aux_rod: name of the auxiliary rod (e.g., 'B')

As a void the declaration may look like:

```
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
```

In your recursive ***"Towers of Hanoi"*** solution, also identify explicitly which is the **"base"** case and explain how the problem becomes **"smaller"** at every recursive step. Include in your solution the ability to **print** a message that instructs a human how to move the disks. This print should look like **"Move disk NUMBER from rod CHAR to rod CHAR"**.

Solution:

- Indicative Solution – Do not consider it as uniquely correct.
- Only the essentials – not extensive discussion

```
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)

{
    if (n == 1)
    {
        printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
    return;
    }
    towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
    printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
    towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}

int main()
{
    int n = 5; // Number of disks
    towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
    return 0;
}
```

Question 2 [Topic: Stacks] [Percentage: 25%]: Consider the following subsets of code:

File: Stack.h

```
#ifndef STACK_H
#define STACK_H

#include <stdexcept>
#include <iostream>

using namespace std;

template <typename DataType>
class Stack {
  public:
    static const int MAX_STACK_SIZE = 8;

    virtual ~Stack();

    virtual void push(const DataType& newDataItem) throw (logic_error) = 0;
    virtual DataType pop() throw (logic_error) = 0;

    virtual void clear() = 0;

    virtual bool isEmpty() const = 0;
    virtual bool isFull() const = 0;

    virtual void showStructure() const = 0;
};

template <typename DataType>
Stack<DataType>::~Stack()
// Not worth having a separate class implementation file for the destuctor
{}

#endif    // #ifndef STACK_H
```

File: StackLinked.h

```
#ifndef STACKARRAY_H
#define STACKARRAY_H

#include <stdexcept>
```

```
#include <iostream>

using namespace std;

#include "Stack.h"

template <typename DataType>
class StackLinked : public Stack<DataType> {

  public:

    StackLinked(int maxNumber = Stack<DataType>::MAX_STACK_SIZE);
    StackLinked(const StackLinked& other);
    StackLinked& operator=(const StackLinked& other);
    ~StackLinked();

    void push(const DataType& newDataItem) throw (logic_error);
    DataType pop() throw (logic_error);

    void clear();

    bool isEmpty() const;
    bool isFull() const;

    void showStructure() const;

  private:

    class StackNode {
      public:
  StackNode(const DataType& nodeData, StackNode* nextPtr);

  DataType dataItem;
  StackNode* next;
    };

    StackNode* top;
};

#endif    //#ifndef STACKARRAY_H
```

Write the implementations of methods **pop (15%/25%)** and **isEmpty (10%/25%)** for the Linked implementation of the Stack. When not possible to pop, a message should appear that explains that pop cannot happen when the stack is empty.

Solution:

- Indicative Solution – Do not consider it as uniquely correct.
- Only the essentials – not extensive discussion

```
template <typename DataType>
DataType StackLinked<DataType>::pop() throw (logic_error)

// Removes the topmost item from a stack and returns it.
{
    if (isEmpty()) {
    throw logic_error("pop() while stack empty");
    }

    StackNode* temp = top;
    top = top->next;

    DataType value = temp->dataItem;
    delete temp;

    return value;
}


template <typename DataType>
bool StackLinked<DataType>::isEmpty() const

// Returns true if a stack is empty. Otherwise, returns false.
{
    return top == 0;
}

template <typename DataType>
void StackLinked<DataType>::clear()

// Removes all the data items from a stack.
{
    for (StackNode* temp = top; top != 0; temp = top) {
    top = top->next;

    delete temp;
    }
    // Invariant: At this point in the code, top == 0.
    // Top does not heed to explicitly set to 0. It was
    // either 0 before the loop, or emerged from the loop as 0.
}
```

Question 3 [Topic: Sorting Algorithms] [Percentage: 25%]: MergeSort.

Q3.1 [18%/25%]: Provide an implementation of the merge method in the mergeSort code provided below.

```
template <class ItemType>
void mergeSort(ItemType theArray[], int first, int last)
{
    if (first < last)
    {
        int mid = first + (last-first)/2;
        mergeSort(theArray,first,mid);
        mergeSort(theArray,mid+1,last);
        merge(theArray,first,mid,last);
    } // end if
} // end mergeSort
```

Q3.2 [2%/25%]: Once you have implemented your merge function, then consider that the following array is requested to be sorted:

[38 , 16 , 27 , 39 , 12 , 27]

Given the aforementioned mergeSort implementation and your implementation of merge present the several steps of recursive calls to mergeSort and the merge steps that take place in order to sort this array using mergeSort.

Q3.3 [5%/25%]: Derive the Big O function for mergeSort if the array to be sorted has total size **n**. In your analysis consider how many comparisons merging requires, how many moves take place (e.g., from the original array to any temporary array), how deep the recursion goes, how many calls to merge take place etc.

Solution:

- Indicative Solution – Do not consider it as uniquely correct.
- Only the essentials – not extensive discussion

3.1

```
template <class ItemType>

void merge(ItemType theArray[], int first, int mid, int last)
{
   ItemType tempArray[MAX_SIZE]; // Temporary array
                                 // Initialize the local indices to indicate the
subarrays
```

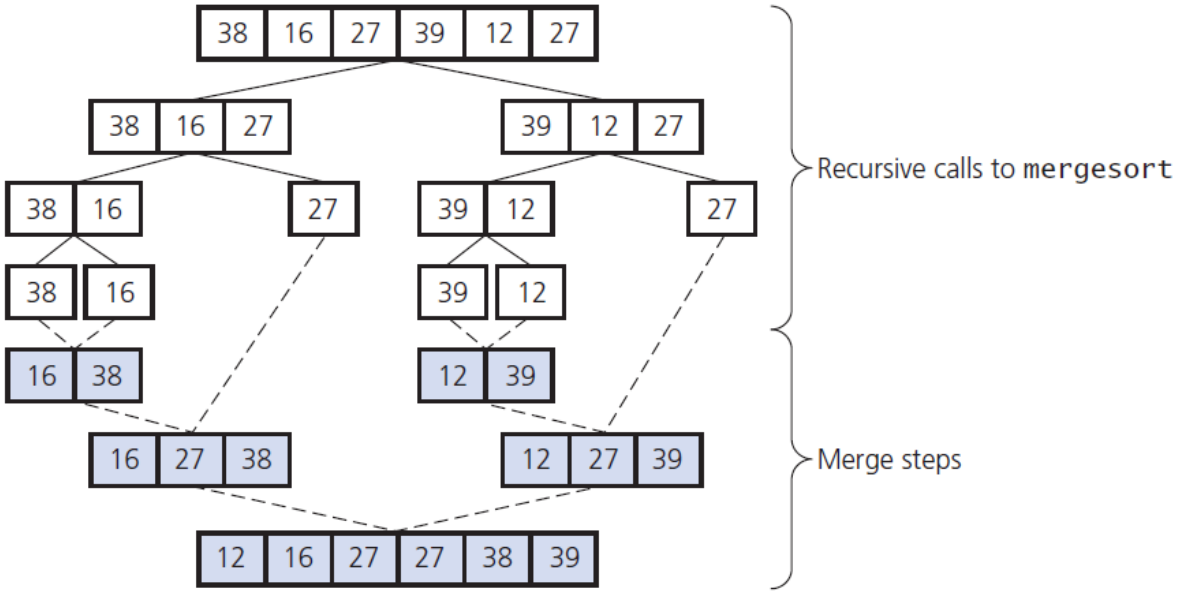
```
int first1 = first;           // Beginning of first subarray
int last1 = mid;              // End of first subarray
int first2 = mid + 1;         // Beginning of second subarray
int last2 = last;             // End of second subarray

// While both subarrays are not empty, copy the
// smaller item into the temporary array
int index = first1;           // Next available location in tempArray
while ((first1 <= last1) && (first2 <= last2))
{
   // At this point, tempArray[first..index-1] is in order
   if (theArray[first1] <= theArray[first2])
   {
      tempArray[index] = theArray[first1];
      first1++;
   }
   else
   {
      tempArray[index] = theArray[first2];
      first2++;
   }  // end if

   index++;
}  // end while

// Finish off the first subarray, if necessary
while (first1 <= last1)
{
   // At this point, tempArray[first..index-1] is in order
   tempArray[index] = theArray[first1];
   first1++;
   index++;
}  // end while
```

3.2



3.3

- Each merge step merges theArray[first..mid] and theArray[mid+1..last].
- If the total number of items in the two array segments to be merged is n, then merging the segments requires at most $n - 1$ comparisons.
- In addition, there are n moves from the original array to the temporary array, and n moves from the temporary array back to the original array. Thus, each merge step requires $3 \times n - 1$ major operations.
- Each call to mergeSort recursively calls itself twice. Each call to mergeSort halves the array. If n is a power of 2, the recursion goes $k = log2n$ levels deep. If n is not a power of 2, there are $1 + log2n$ levels of recursive calls to mergeSort.
- The original call to mergeSort calls merge once. Then merge merges all n items and requires $3 \times n - 1$ operations. At level m of the recursion, $2m$ calls merge to occur, each of these calls merges $n/2m$ items and so requires $3 \times (n/2m) - 1$ operations. Together, the $2m$ calls to merge require $3 \times n - 2m$ operations.
- Thus each level of recursion requires $O(n)$ giving a total of $O(nlogn)$

Question 4 [Topic: Lists] [Percentage: 25%]: Consider the following header file for the class **LinkedList**:

```
#ifndef LINKED_LIST_
#define LINKED_LIST_

#include "ListInterface.h"
#include "Node.h"
#include "PrecondViolatedExcept.h"

template<class ItemType>
class LinkedList : public ListInterface<ItemType>
{
private:
   Node<ItemType>* headPtr; // Pointer to first node in the chain;
                            // (contains the first entry in the list)
   int itemCount;           // Current count of list items

   // Locates a specified node in this linked list.
   // @pre  position is the number of the desired node;
   //       position >= 1 and position <= itemCount.
   // @post  The node is found and a pointer to it is returned.
   // @param position  The number of the node to locate.
   // @return  A pointer to the node at the given position.
   Node<ItemType>* getNodeAt(int position) const;

public:
   LinkedList();
   LinkedList(const LinkedList<ItemType>& aList);
   virtual ~LinkedList();

   bool isEmpty() const;
   int getLength() const;
   bool insert(int newPosition, const ItemType& newEntry);
   bool remove(int position);
   void clear();

   /** @throw PrecondViolatedExcept if position < 1 or
                                       position > getLength(). */
   ItemType getEntry(int position) const throw(PrecondViolatedExcept);

   /** @throw PrecondViolatedExcept if position < 1 or
                                       position > getLength(). */
   void replace(int position, const ItemType& newEntry)
                              throw(PrecondViolatedExcept);
```

```
}; // end LinkedList

#include "LinkedList.cpp"
#endif
```

Q4.1 [20%/25%]: We want to write a destructor that uses recursion so that it deletes each node of the underlying linked chain. We start by writing:

```
template<class ItemType>
LinkedList<ItemType>::~LinkedList()
{
   deallocate(headPtr);
   headPtr = nullptr;
   itemCount = 0;
}  // end destructor
```

You are requested to provide the implementation of the private method **deallocate** such that each node is deleted.

Q4.2 [5%/25%]: Given a nonempty list that is an instance of LinkedList, at what position does an insertion of a new entry require the fewest operations.

Solution:

- Indicative Solution – Do not consider it as uniquely correct.
- Only the essentials – not extensive discussion

4.1

```
template<class ItemType>

void LinkedList<ItemType>::deallocate(Node<ItemType>* nextNodePtr)
{
   if (nextNodePtr != nullptr)
   {
      Node<ItemType>* nodeToDeletePtr = nextNodePtr;
      nextNodePtr = nextNodePtr->getNext();
      delete nodeToDeletePtr;
      deallocate(nextNodePtr);
   }  // end if
}  // end deallocate
```

4.2

An insertion at position 1 of the list is implemented as an insertion at the beginning of the linked chain. This insertion can be accomplished without a search of the chain for the desired position. Therefore, it requires the fewest operations.

Question 5 [Topic: Mixed] [Percentage: 10%]: Respond to the questions provided below.

Q5.1 [2.5%/10%]: What is the role of the method peek() in the implementation of the ADT Stack. Does the amount of items stored in the Stack change after a call of peek()?

A5.1:

Q5.2 [2.5%/10%]: Write (1+5)^4 (infix notation) in postfix notation

A5.2:

Q5.3 [2.5%/10%]: Which of the following sorting algorithms, in its typical implementation, provides optimal performance when applied on a sorted – or almost sorted – array? Explain your answer shortly.

A: Merge Sort
B: Insertion Sort
C: Quick Sort

A5.3:

Q5.4 [2.5%/10%]: Convert the infix expression a/b*c to postfix form by using a stack. Be sure to account for left-to-right association. Show the status of the stack after each step. For every step write: a) what is the character processed, b) what is the postfix expression so-far, c) what are the elements of the stack ("aStack") used to do the conversion.

A5.4: