# _LITfinalLOGO

# **SUMMER EXAMINATIONS 2012**

**Tuesday, 15th May 2012, 14.30 p.m. – 16.30 p.m.**

**KSDEM\_8\_Y2**

**Course:** Bachelor of Science (Hons) in Software Development

**Year:** Two

**Subject:** Data Structures and Algorithms

**Time Allowed:** 2 Hours

**Instructions:**

**1.** Answer any **FOUR (4)** Questions.

1. All questions carry equal marks
2. Start each question on a new page.
3. Write the question number at the top of each page.
4. Circle the numbers of the questions you answer at the

front of your answer book.

**Additional Attachments Exam Materials to accompany this paper:**

**A.** None

# **Internal Examiners: External Examiners:** Mr. Des O’Carroll Mr. Paul Powell

**Q.1.**

**(a)** Define what is meant by a Stack data structure.

**(5 marks)**

**(b)** Given the following class declaration for a linked Stack:

template <class ItemType>

struct NodeType

{

ItemType info;

NodeType\* next;

};

template<class ItemType>

class StackType

{

public:

StackType();

~StackType();

void MakeEmpty();

bool IsFull() const;

bool IsEmpty() const;

**void Push(ItemType newItem);**

**void Pop(ItemType& item);**

private:

NodeType<ItemType>\* topPtr;

};

implement the following member functions:

(i) Push( )

// Function: Adds newItem to the top of the stack.

// Pre: Stack has been initialized and is not full.

// Post: newItem is at the top of the stack.

(ii) Pop( )

// Function: Removes top item from the stack and

// returns it in item.

// Pre: Stack has been initialized and is not empty.

// Post: Top element has been removed from stack.

// item is a copy of the removed item.

**(10 marks)**

**(c)** Write a function that compares two linked stacks for equality. The function must take two linked stacks as parameters and returns true if they are identical (and false if not). The stacks should remain unchanged after the function ends. The stacks contain integers. The nodetype definition is similar to the definition in part (b) above.

**(10 marks)**

**(Total 25 Marks)**

**Q.2.**

**(a)** Every recursive process consists of 2 parts. Briefly describe these 2 parts.

**(4 marks)**

**(b)** Consider this function declaration:

void myFunction( int counter)

{

if(counter > 0)

{

cout<<"\*";

myFunction(n - 1);

cout<<"!" ;

}

}

What is the exact output from the above recursive function if it called as myFunction(4)? Explain clearly how you arrived at your answer.

**(6 marks)**

**(c)** Given the following **recursive** version of the Towers of Hanoi function

**void Hanoi (int n, int start, int finish, int temp)**

**{**

**if ( n== 1)**

**{**

**cout << "Move top disk from pole " << start**

**<< " to top of pole"<< finish << endl;**

**}**

**else**

**{**

**Hanoi (n-1, start, temp, finish);**

**cout << "Move top disk from pole " << start**

**<< " to top of pole " << finish << endl;**

**Hanoi (n-1, temp, finish, start);**

**}**

**}**

Trace through the function for moving four disks from pole START to pole FINISH. Assume START = 1, TEMP=2, and FINISH= 3 for the initial call to the function.

**(15 marks)**

**(Total 25 Marks)**

**Q.3.**

**(a)** Give the recursive definition for a binary search tree.

**(6 marks)**

**(b)** Consider this binary search tree:

S

O

L

T

M

A

What is the output if this tree is traversed in

1. Preorder
2. Postorder

**(6 marks)**

**(c)** Assuming the following class declaration for a Binary Search Tree ADT:

template <class ItemType>

struct TreeNode

{

ItemType info;

TreeNode \*left;

TreeNode \*right;

};

// Assume “<” and “==” operators have been overloaded for

// the class ItemType

template <class ItemType>

class TreeType

{

public:

TreeType( );

~TreeType( );

TreeType (const TreeType<ItemType>& originalTree);

void operator=(const TreeType<ItemType>& originalTree);

void MakeEmpty( );

bool IsEmpty( ) const;

bool IsFull( ) const;

int NumberOfNodes( ) const;

void RetrieveItem(ItemType &item, bool &found);

void InsertItem(ItemType item);

void DeleteItem(ItemType item);

void PrintTree(ostream &outFile) const;

private:

TreeNode<ItemType> \*root;

};

Implement the following recursive member function, including any helper or associated functions required:

**template<class ItemType>**

**void TreeType<ItemType>::InsertItem(ItemType item)**

// Calls recursive function Insert to insert item into //tree.

**(13 marks)**

**(Total 25 Marks)**

**Q.4.**

**(a)** Starting with an empty maximal heap (where bigger values represent higher priorities), insert items with priorities 35, 17, 88, 45, 51, 28 and 64 (in this order) into that heap.

Draw the logical view of the heap (i.e. as a binary tree) of your result.

**(5 marks)**

**(b)** Describe the heap **ordering** property as it applies to a maximal heap.

**(5 marks)**

**(c)** The priority queue algorithms on heaps all work by first making simple

structural modification which could violate the order property of the

heap, then travelling through the heap modifying it to ensure that the

heap order property is satisfied everywhere, using either ReHeapUp()

or ReHeapDown().

Assuming the following declarations:

template <class ItemType>

// Assumes ItemType is either a built-in simple type or a

// class with overloaded relational operators.

struct HeapType

{

**void ReheapDown (int root, int bottom);**

void ReheapUp (int root, int bottom);

ItemType \* elements; // Array to be allocated

// dynamically

int numElements;

};

template<class ItemType>

class PQType

{

public:

PQType(int);

~PQType();

void MakeEmpty();

bool IsEmpty() const;

bool IsFull() const;

void Enqueue(ItemType newItem);

**void Dequeue(ItemType& item);**

private:

int numItems;

HeapType<ItemType> items;

int maxItems;

};

**(i)** Implement DeQueue()

**(ii)** Implement ReHeapDown().

**(15 marks)**

**(Total 25 Marks)**

**Q.5**

**(a)** Implement function Merge( ) that is called by MergeSort( ) below.

**void MergeSort (ItemType values[], int first, int last)**

**{**

**if (first < last)**

**{**

**int middle = (first + last) / 2;**

**MergeSort (values, first, middle);**

**MergeSort (values, middle + 1, last);**

**Merge (values, first, middle, middle+1, last);**

**}**

**}**

**(13 marks)**

**(b)** MergeSort the following data set:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **20** | **35** | **28** | **18** | **14** | **41** | **39** | **3** |

**(12 marks)**

**(Total 25 Marks)**