# _LITfinalLOGO

# **SUMMER EXAMINATIONS 2014**

**Tuesday, 13th May 2014, 14.30 p.m. – 16.30 p.m.**

**KSDEM\_8\_Y2**

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

**Year:** Two

**Subject:** Data Structures & Alogorithms

**Time Allowed:** 2 Hours

**Instructions:**

**1.** Answer **any THREE (3)** Questions.

1. All questions carry equal marks
2. The examination paper is marked out of 75 marks.
3. Start each question on a new page.
4. Write the question number at the top of each page.
5. Circle the numbers of the questions you answer at the

front of your answer book.

**Additional Attachments or Exam Material to accompany this paper:**

### A. None

**Internal Examiners: External Examiners:**  Mr. Des O’Carroll Mr Brian Gillespie

**Q.1.**

**(a)** There are 4 broad categories of recursion. Describe each of the 4 categories.

**(8 marks)**

**(b)** Implement a recursive function to print all the numbers from N down to 1.

The header is as follows:

**void print(int n)**

For example if print (4) is called, the following output will emerge:

**4 3 2 1**

Assume N will always be a positive integer.

**(10 marks)**

**(c)**  Implement a **recursive** version of the Towers of Hanoi function.

**(7 marks)**

**(Total 25 Marks)**

**Q.2.**

(a) Implement the recursive **MergeSort( )** function to sort an array of integers

by filling in the arguments in the function calls below:

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

{

if (first < last)

{

int middle = (first + last) / 2;

MergeSort (…, …, …);

MergeSort (…, …, …);

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

}

} **(6 marks)**

(b) The function Merge( ) above merges sorted array elements between index first and middle with sorted array elements between middle+1 and last. Implement the function Merge().

**(10 marks)**

(c) Carry out a **MergeSort** on the following array of integers, showing the relevant section of the array being sorted with each recursive call, along with the relevant function call. (i.e. MergeSort( ) or Merge( ))

20 35 28 18 14 41 39 3

**(9 marks)**

**(Total 25 Marks)**

**Q.3.**

Given the following definition and declaration:

struct ListType

{

int length;

int info[MAX\_ITEMS];

}

ListType list;

1. Implement a **Binary Search** to search a list of integers by filling in the missing code below:

void BinarySearch (ListType list, int & item, bool & found)

{

int midPoint;

int first = 0;

int last = list.length – 1;

bool moreToSearch = first <= last;

found = false;

while (moreToSearch && !found)

{

…… **Fill in missing code here**

}

}

**(10 marks)**

1. Carry out a Binary Search on **46** for the following list of integers, showing the relevant section of the list being searched with each iteration, along with the values of *first*, *last* and *midPoint*:

23 28 31 38 43 45 46 49 52 57 63 66 69 73 78 82 84 84818888888888 84

**(7 marks)**

1. The **Forgetful** version of Binary Search ‘forgets**’** the possibility that *item* might be found quickly and continues, whether *item* has been found or not, to subdivide the list until what remains has length 1. Modify the code above to implement the **Forgetful** version of Binary Search.

**(8 marks)**

**(Total 25 Marks)**

**Q.4.**

**(a)** Define what is meant by:

(i) a heap

(ii) a priority queue

**(5 marks)**

**(b)** 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( )

**(20 marks)**

**(Total 25 Marks)**