Paper Number(s): E1.8
E2.18

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2006

EEE Part II / ISE Part I: MEng, BEng and ACGI

Corrected Copy

Nine

SOFTWARE ENGINEERING: INTRODUCTION, ALGORITHMS AND DATA STRUCTURES

Tuesday 23rd May 2006 2:00pm

There are THREE questions on this paper.

Question 1 is compulsory and carries 40% of the marks.

Answer Question 1 and EITHER Question 2 (carrying 60%) or Question 3 (carrying 60%).

This exam is **open book**

Time allowed: 1:30 hours.

Any special instructions for invigilators and information for candidates are on page 1.

Examiners responsible:

First Marker(s): Cheung, PYK.
Second Marker(s): Constantinides, G

Information for invigilators:

Students may bring any written or printed aids into the examination.

Information for candidates:

Marks may be deducted for answers that use unnecessarily complicated algorithms.

[Question 1 is compulsory]

1. a) The following C++ function computes the number of address bits required to access memory with n locations where n > 1. For example, if n = 218, then the function should return a value of 8. Identify five errors in the C++ code.

```
int addressBits (n) {
int temp;
for (i = 0 ; i < n ;  i = i  * 2);
      temp++ ;
return temp ;
}
```

[7]

b) A set of surnames are inserted into an ordered binary tree. Draw a tree for each of the following lists of surnames assuming that elements of the list are inserted in order.

(i) (Smith, Lee, Patel, Jones, Davidson, Yate, Ko)

(ii) (Lee, Smith, Jones, Ko, Patel, Yate, Davidson)

[6]

c) The Fibonacci numbers F_n are defined as follows:

$F_{i+2} = F_i + F_{i+1}, \quad F_0 = 1, \quad F_1 = 1$

Write a C++ recursive function that returns the Fibonacci number for an input n, where $n \geq 0$. (No marks will be awarded if recursion is not used.)

[6]

d) For each of the following situations, state with reasons which type of loops (i.e. while, do-while or for) would work best:

(i) Reading in the list of failed subjects for a student.

(ii) Calculating the sum of a finite series.

(iii) Testing for a list of items where the list has at least one item.

[7]

e) Construct a parse tree for the following expressions:

(i) (3 + 4)*7 + 5 - 5*9

(ii) ((4 – 6)/8 + 9)*3

[7]

f) Consider the following C++ code segment. With justification, state the values of the integer variables x and y after the code segment is executed.

```
int x = 37;
int y = 99;
int *ptr_x = &x;
int *ptr_y = &y ;
*ptr_x = *ptr_y ;
```

[7]

2. The following is the type declaration for a dynamic doubly-linked list of integers in C++.

```
class Node {
    public:
        int data;
        Node *next;
        Node *prev;
};

typedef Node* NodePtr;
NodePtr hdList  =  NULL;
```

a) Write a C++ function `listLength()` that takes the linked list pointed to by `hdList` and returns number of nodes in the list.

[10]

b) Write a C++ access function `middleItem()` that takes the linked list pointed to by `hdList` and returns the integer halfway along the list. In other words, if the list is of length N, the function returns the integer stored in the $\frac{N}{2}^{th}$ node if N is even, and the $\frac{N+1}{2}^{th}$ node if N is odd. You should use the following method: count the number of nodes in the list using the function `listLength()`, then start from the beginning of the list and traverse the list by the correct number of nodes. If the list is empty, return a value of 0.

[10]

c) Write a new version of the function `middleItem()` that uses the following method. Traverse the list from the head of the list using two pointers. The first pointer advances one node at a time and the second pointer advances two nodes at a time. When the second pointer reaches the end of the list, the first pointer will be pointing to the node required.

[15]

d) State with justification which of the two versions of the function `middleItem()` is more efficient.

[5]

e) Hence, or otherwise, write a C++ function to insert a new node just before the middle item of the linked list.

[20]

3. The following is the type declaration for a sorted binary tree of strings in C++.

```
class TreeNode {
    public:
        string name;
        TreeNode *left;
        TreeNode *right;
};

typedef TreeNode* TreePtr;
TreePtr tree  =  NULL;
```

The following access functions are already available:

```
// Return a pointer pointing to the left child of the tree
TreePtr leftChild (TreePtr tree) {
     return tree->left;
}

// Return a pointer pointing to the right child of the tree
TreePtr rightChild (TreePtr tree) {
     return tree->right;
}
```

a) Write a C++ function `treeCount()` that takes as input a pointer to the root of a binary tree and returns the number of nodes in the tree. For example, for the binary tree shown in Figure 3.1, `treeCount()` should return a value of 8.

[10]

b) Hence, or otherwise, write a C++ function `treeDifference()` that takes as input a pointer to the root of a binary tree and returns the difference between the number of nodes in its left sub-tree and the number of nodes in its right sub-tree. For the binary tree shown in Figure 3.1, `treeDifference()` should return a value of -1.

[5]

c) Write another C++ function `treeDepth()` that returns the depth of the binary tree. The depth is the number of nodes in the longest branch of the tree from the root to the leaf. For the binary tree shown in Figure 3.1, the longest branch is: peter → sam → ray → phil, and `treeDepth()` should return a value of 4.

[15]

d) A sorted binary tree of strings has been created. If the values returned by `treeDifference()` and `treeDepth()` are the same, what can you deduce about the data stored in the tree and the tree structure?

[10]

e) It is desirable to split a binary tree into two separate trees T1 and T2 which are roughly the same size. A C++ function `treeSplit()` to achieve this has the following function prototype:

```
TreePtr treeSplit (TreePtr treeRoot);
```

The input to the function is the pointer `treeRoot` pointing to the root of the original tree. On exit the function returns a pointer to T2 and `treeRoot` remains unchanged, but the original tree is now trimmed to T1. Figure 3.2 shows the results of `treeSplit()` when applied to the tree in Figure 3.1.

The algorithm of `treeSplit()` is described by the following steps:

1. Assuming that the number of nodes is odd, find the number of nodes of the entire tree;
2. Calculate the desired number of nodes (B) for each sub-tree after splitting;
3. Starting from the root of the tree and if the pointer is not empty, find the numbers of nodes on the left and the right sub-trees ;
4. Traverse to the side that has higher number of nodes;
5. Repeat 3 and 4 until the node count is B or lower, a pointer to T2 is now found;
6. Trim T2 from the original tree to form T1.

Write a `treeSplit()` function in C++.

[20]

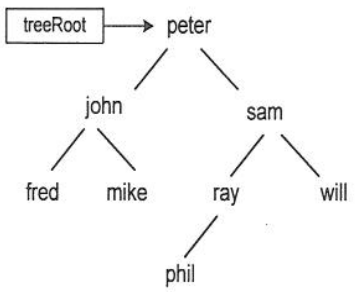


Figure 3.1

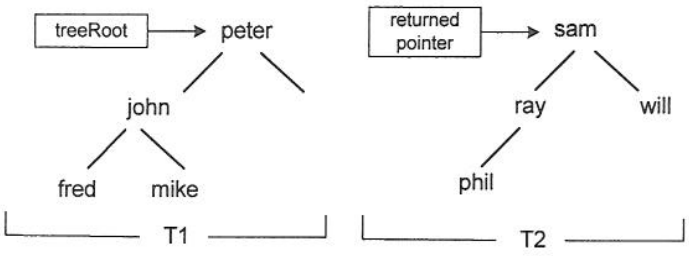


Figure 3.2

[THE END]

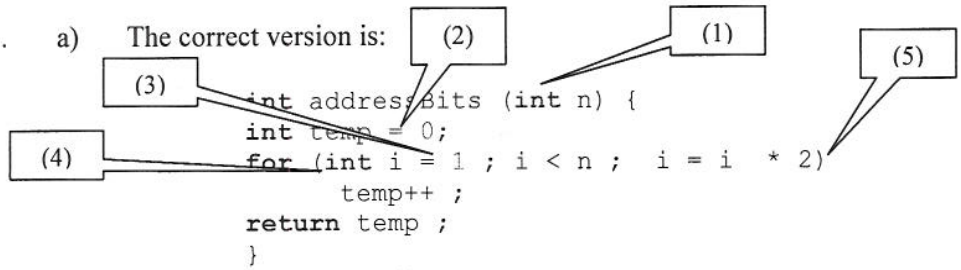
E1.8/ E2.18 Software Engineering: Algorithms & Data Structures

Solutions 2006

All questions are unseen. This is open-book examination.

Question 1 is compulsory. This question covers most of the syllabus.

1. a) The correct version is:



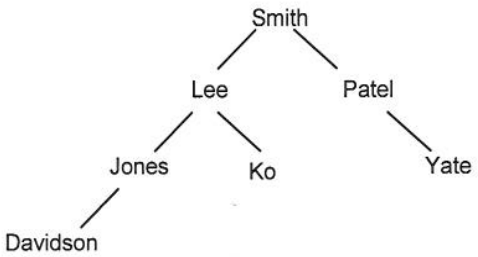

```
int addressBits (int n) {
int temp = 0;
for (int i = 1 ; i < n ;  i = i  * 2)
      temp++ ;
return temp ;
}
```

Errors (1 mark each except (3)):

1) The parameter type is not declared. It should be: `int addressBits (int n){`
2) `temp` is not initialised. It should be initialised to 0.
3) The index value `i` should be initialised to 1, not 0. As it stands, the loop never exits. (2 marks)
4) The index `i` is not declared. Can declare inside for-loop as shown.
5) The for-loop statement should NOT have "`;`" at the end of the line.

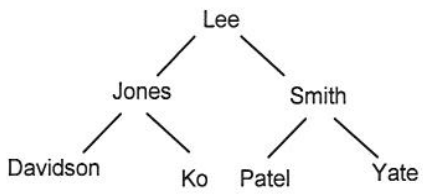
[7]

b) (i) (Smith, Lee, Patel, Jones, Davidson, Yate, Ko)



[3]

(ii) (Lee, Smith, Jones, Ko, Patel, Yate, Davidson)



[3]

c)

```
int fibonacci( int n )
      {
            if ( n == 0 || n == 1 )
                  return 1;
            else
                  return fibonacci(n - 1) + fibonacci(n - 2);
      }
```

[6]

d)

(i) While loop because the list may be empty (i.e student has fail no subject).

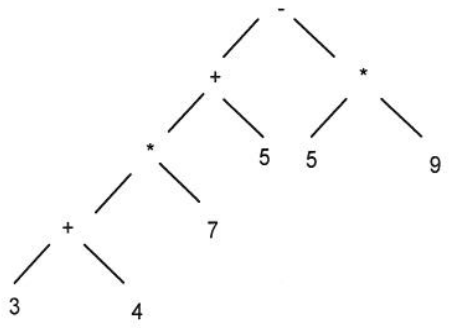
(ii) For loop because the series is finite and therefore the number of iteration is known before hand.

(iii) Do-while loop because there is at least one item in the list and the length of the list not known a priori.

[7]

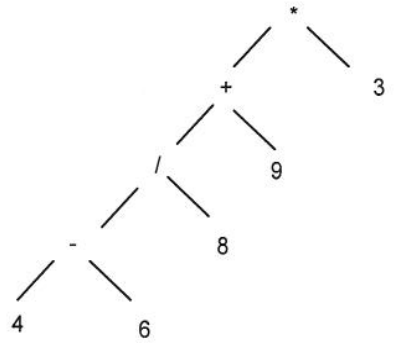
e) The parse trees for the expressions are:

(i) (3 + 4)*7 + 5 - 5*9



[3]

(ii) ((4 – 6)/8 + 9)*3



[4]

f)

```
int x = 37;
int y = 99;
int *ptr_x = &x;     // ptr_x points to x
int *ptr_y = &y ;    // ptr_y points to yx
*ptr_x = *ptr_y ;    // content at ptr_x is overwritten
                     //     with content at ptr_y
```

Therefore x = 99, y = 99.

[7]

2. (This question tests student's ability to use linked list. To make this slightly more challenging, a doubly-linked list is used.)

a)

```
int listLength (NodePtr list) {
    if (list == NULL)
        return 0;
    else
        return (listLength(list->next) + 1);
}
```

[10]

b)

```
NodePtr goToMiddle (NodePtr list) {
    int n = listLength(list);
    int m;

    m = (n+1)/2;  // this gives the correct result for both even and odd n
    if (n != 0) {
        for (int i = 1; i < m; i++)     // traverse list by m nodes
            list = list->next;
        return (list->data);
    }
    else
        return NULL;
}

int middleItem (NodePtr list) {
    NodePtr middlePtr = NULL;
    middlePtr = goToMiddle(list);
    if (middlePtr != NULL)
        return middlePtr->data;
    else
        return 0;
}
```

[10]

c)

```
NodePtr goToMiddle (NodePtr list) {
    NodePtr first = list;
    NodePtr second = list;

    while ((second != NULL) && (second->next != NULL)) {
        first = first->next;
        second = second->next->next;
        }
    return first;
}

int middleItem (NodePtr list) {
    NodePtr middlePtr = NULL;
    middlePtr = goToMiddle(list);
    if (middlePtr != NULL)
        return middlePtr->data;
    else
        return 0;
```

[15]

d) The second version is more efficient because for a list of length n, it only requires around n/2 iterations of the while loop during list traversal. The first version calls listLength( ) + n/2 iterations of the for loop. This makes the first version almost 3 times longer than the second version in list traversal.

[5]

e)

```
void insertMiddle (int data,
                   NodePtr &hdList) {
    NodePtr middlePtr = goToMiddle(hdList);
    NodePtr newNode = new Node;


    newNode->data = data;
    if (middlePtr == NULL) {
        newNode->next = NULL;
        newNode->prev = NULL;
        hdList = newNode;
        }
    else {
        newNode->next = middlePtr;
        newNode->prev = middlePtr->prev;
        middlePtr->prev->next = newNode;
        middlePtr->prev = newNode;
    }
}
```

[20]

3. (This question tests student's ability to manipulate a binary tree data structure.)

a)

```
int treeCount (TreePtr tree) {
  if (tree==NULL)
      return 0;
  else
      return (treeCount(leftChild(tree))+treeCount(rightChild(tree))+1);
}
```

[10]

b)

```
int treeDifference (TreePtr tree) {
  return (treeCount(leftChild(tree)) - treeCount(rightChild(tree)));
}
```

[5]

c)

```
int treeDepth (TreePtr tree) {
  int leftDepth = 0;
  int rightDepth = 0;

  if (tree == NULL)
        return 0;
  else {
        leftDepth = treeDepth(leftChild(tree));
        rightDepth = treeDepth(rightChild(tree));
        if (leftDepth > rightDepth)
            return leftDepth + 1;
        else
            return rightDepth + 1;
  }
}
```

[15]

d) If `treeDifference()` and `treeDepth()` return the same value, it means that the tree is completely one sided, i.e. all data reside either entirely on the right branch or the left branch of the tree. This is effective a linked list!

[10]

e) (This part of the question is quite hard.)

```
TreePtr treeSplit (TreePtr treeRoot) {
    int n;
    int m;
    TreePtr temp;
    TreePtr prev;

// find number of nodes in tree
    n = treeCount(treeRoot);
// traverse until treeCount is half
    m = int (n+1)/2;    temp = treeRoot;
    while (temp != NULL) {
        if ( treeCount(leftChild(temp)) > treeCount(rightChild(temp)) ) {
            prev = temp;                    // traverse left
            temp = leftChild(temp);
            if (treeCount(temp) <= m) { // found place to split
              prev->left = NULL;
              return temp;
            }
        }
        else {              // traverse right
            prev = temp;
            temp = rightChild(temp);
            if (treeCount(temp) <= m) { // found place to split
              prev->right = NULL;
              return temp;
            }
        }
    }
}
```

[20]