City University of Hong Kong

Course Code & Title: EE2331 Data Structures and Algorithms

Semester/year: Semester A, 2010-11

Time Allowed: Two hours

This paper has 10 pages (including this cover page)

Answer ALL questions in this paper.

Material, aids & instruments permitted to be used during examination:

1. Approved calculators

Seat Number: ______________________

Student Number: ______________________

Programme: ______________________

No. of Supplementary Sheets Used: ______________________

For Office Use Only:

Q1	Q2	Q3	Q4	Q5	Q6	Total

Question 1 (10 marks)

(a) Suppose that a hash table is of size 7 and the hash function is defined as $h(i) = i \% 7$, show the content of the hash table after inserting the elements from the following sequence via linear probing.

95, 17, 54, 82, 65, 23

(b) The method in part (a) is not perfect hashing. Peter suggests that mid-square method can be used to improve the hashing of the above number sequence, while John argues using folding method instead. Verify the two approaches and explain which one can achieve perfect hashing for the same sequence in (a).

(Q1a) (4 marks)

	htable[0]	htable[1]	htable[2]	htable[3]	htable[4]	htable[5]	htable[6]
After 1st insertion					95		
After 2nd insertion				17	95		
After 3rd insertion				17	95	54	
After 4th insertion				17	95	54	82
After 5th insertion			65	17	95	54	82
After 6th insertion	23		65	17	95	54	82

(Q1b) (6 marks)

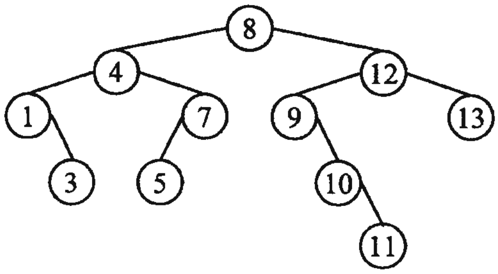
9+5=14	14%7=0
1+7=8	8%7=1
5+4=9	9%7=2
8+2=10	10%7=3
6+5=11	11%7=4
2+3=5	5%7=5

Because the hash values are distinct, folding method can achieve perfect hashing for these numbers.

Question 2 (15 marks)

(a) Given the following **binary search tree**:

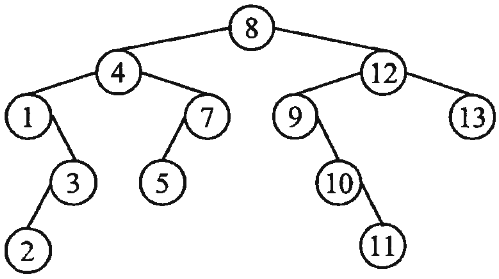
(i) Write down the postorder traversal sequence of the tree.
(ii) Draw the resultant tree after inserting a node of value 2 to the tree.
(iii) Draw the resultant trees (two solutions) after deleting the root from the tree in (i).



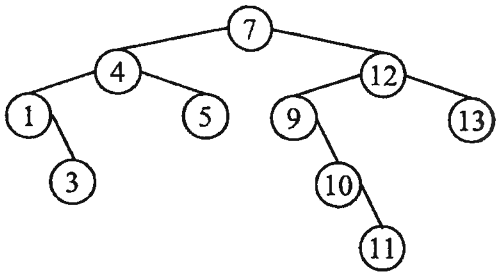
(Q2a) (i) (2 marks)

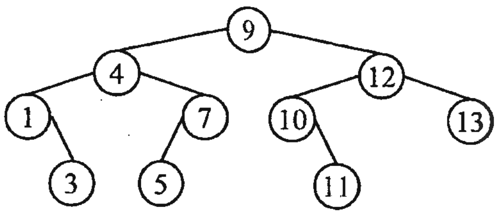
3, 1, 5, 7, 4, 11, 10, 9, 13, 12, 8

(Q2a) (ii) (3 marks)



(Q2a) (iii) (4 marks)

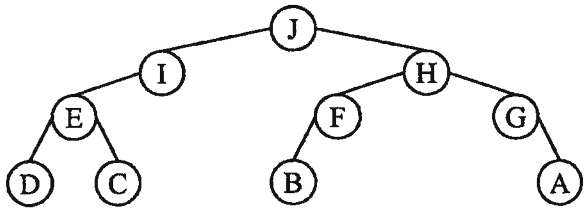




(b) Given the <u>inorder</u> and <u>preorder</u> traversal sequences, reconstruct the corresponding binary tree.

Inorder sequence: D E C I J B F H G A
Preorder sequence: J I E D C H F B G A

(Q2b) (6 marks)



Question 3 (20 marks)

(a) The following array is an implicit representation of a max heap tree. By using heapsort to sort the array in ascending order, show the content of the array of the first two sorting passes, and underline the numbers that have already been sorted.

{9, 5, 8, 2, 3, 4, 6, 1}

(b) Apply merge sort to rearrange the following array into descending order. Show the content of the array of the first two sorting passes.

{9, 5, 8, 2, 3, 4, 6, 1}

(c) What is the sorting algorithm represented by the pseudo code below? Briefly explain what it does.

```
function sort(array)
     var[] less, greater
     if length(array) ≤ 1
         return array
     select and remove a value n from array
     for each x in array
         if x ≤ n then append x to less
         else append x to greater
     return concatenate(sort(less), n, sort(greater))
```

(d) Give one good example and one bad example of using radix sort. Illustrate your answer with sample input data and explain it briefly.

(Q3a) (6 marks)

{9, 5, 8, 2, 3, 4, 6, 1} // before sorting
{8, 5, 6, 2, 3, 4, 1, 9}
{6, 5, 4, 2, 3, 1, 8, 9}

(Q3b) (4 marks)

{9, 5, 8, 2, 3, 4, 6, 1} // before sorting
{9, 5, 8, 2, 4, 3, 6, 1}
{9, 8, 5, 2, 6, 4, 3, 1}

(Q3c) (5 marks)

It is quicksort algorithm.

The pseudo code picks a value n as pivot and divides the array into two partitions, less and greater. This process is done recursively for the two partitions.

(Q3d) (5 marks)

The time complexity is $O(k \cdot n)$.

k is the no. of digits of the elements. Radix sort's complexity not only depends on the input size n, but also depends on the length of elements.

Question 4 (10 marks)

Complete the function, *reverseQueue*, using recursion algorithm. The function reserves the order of the elements in the input argument *q*. For example:

Before reversing: q = {1,2,3,4}
After reversing: q = {4,3,2,1}

In your implementation, you should only use the standard queue operations as defined below. You are not allowed to declare any additional variables. You can assume that the input queue is not null and is initialized properly.

```
// the name of the Queue structure
typedef struct _queue Queue;
// return 1 if the queue is empty, otherwise return 0
int Queue_isEmpty(Queue *q);
// return 1 if the queue is full, otherwise return 0
int Queue_isFull(Queue *q);
// remove and return the first element from the front of the queue
int dequeue(Queue *q);
// insert an element to the rare of the queue
void enqueue(Queue *q, int e);
```

(Q4) (10 marks)

```
void reverseQueue(Queue *q) {
    int temp;
    if (!Queue_isEmpty(q)) {
        temp = dequeue(q);
        reverseQueue(q);
        enqueue(q, temp);
    }
}
```

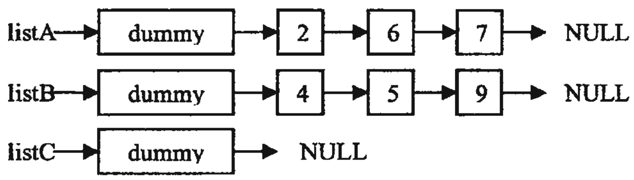
Question 5 (20 marks)

The structure of a singly linked list node is defined as follows:

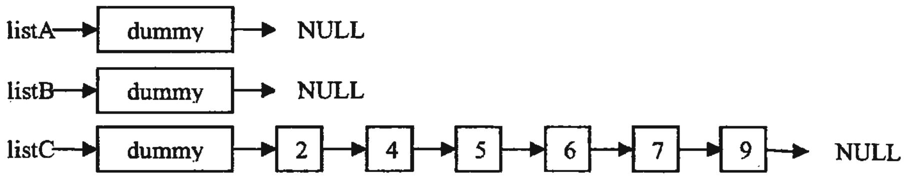
```
typedef struct _node {
    int data;                    // the data field of a list node
    struct _node *next;          // reference to the succeeding node
} Node;
```

(a) Complete the function, *mergeList*, which merges the nodes in *listA* and *listB* into *listC*. All the three lists have a dummy header and *listC* is initially empty. The nodes in *listA* and *listB* are already sorted in ascending order. After merging, all these nodes are inserted to *listC* and they should still be kept in ascending order. For example:

Before merging:



After merging:



In your implementation, you can assume the three arguments *listA*, *listB* and *listC* are not null.

(b) Complete the function, *reverseList*, which reverses the order of the elements in the input argument *list* by using the following stack structure and its standard operations. This stack structure is designed to store the pointer of linked list node. The input list has a dummy header. In your implementation, you can assume the list is not null.

```
typedef struct _stack Stack;      // the name of the Stack structure
void Stack_init(Stack *s);        // initialize the stack to empty
void Stack_destroy(Stack *s);     // delete all elements from the stack
int Stack_isEmpty(Stack *s);      // return 1 if the stack is empty, otherwise 0
int Stack_isFull(Stack *s);       // return 1 if the stack is full, otherwise 0
Node* pop(Stack *s);              // return an element popped from the stack
void push(Stack *s, Node *p);     // push an element into the stack
```

(Q5a) (10 marks)

```
void mergeList(Node *listA, Node *listB, Node *listC) {

    Node *pA = listA->next;
    Node *pB = listB->next;
    Node *pC = listC;

    while (pA != NULL && pB != NULL) {
        if (pA->data < pB->data) {
            pC->next = pA;
            pA = pA->next;
        } else {
            pC->next = pB;
            pB = pB->next;
        }
        pC = pC->next;
    }

    if (pA != NULL) {
        pC->next = pA;
    }
    if (pB != NULL) {
        pC->next = pB;
    }

    listA->next = NULL;
    listB->next = NULL;

}
```

(Q5b) (10 marks)

```
void reverseList(Node *list) {

    Stack stack;
    Stack *s = &stack;
    Stack_init(s);

    Node *p = list->next;
    while (p != NULL) {
        push(s, p);
        p = p->next;
    }

    p = list;
    while (!Stack_isEmpty(s)) {
        p->next = pop(s);
        p = p->next;
    }

    p->next = NULL;
}
```

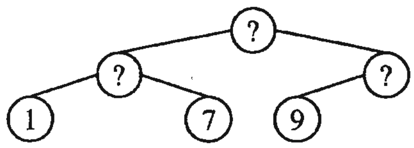
Question 6 (25 marks)

The structure of the tree node of binary tree is defined as follows:

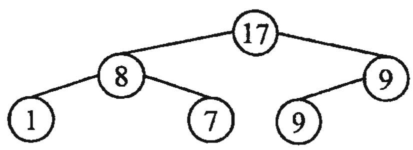
```
typedef struct _treenode {
    int data;
    struct _treenode *left, *right;
} TreeNode;
```

(a) Complete the recursive function, *printBstDescending*, which prints the binary search tree pointed by the input argument *root* in descending order.

(b) Complete the recursive function, *sumChildren*, to update all non-leaf nodes of the tree pointed by the input argument *root* such that the data of every non-leaf node equals to the sum of the data of its two direct children. An example is shown below. In your implementation, you can assume the tree is not empty.

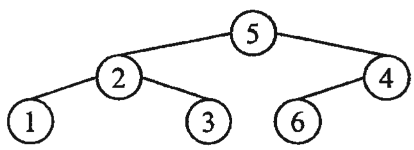


Before the operation



After the operation

(c) Complete the recursive function, *sumLevel*, to compute the sum of the nodes on a given *level* of the tree pointed by the input argument *root*. An example is illustrated below. In your implementation, you can assume the tree is not empty and the input argument *level* is a valid value.



Sum at level 0 = 5

Sum at level 1 = 4 + 2 = 6

Sum at level 2 = 1 + 3 + 6 = 10

(Q6a) (5 marks)

```
void printBstDescending(TreeNode* root) {

    if (root == NULL)
        return;
    printBstDescending(root->right);   //R: go to right subtree
    printf("%d ", root->data);         //V: visit (print) this node
    printBstDescending(root->left);    //L: go to left subtree
}
```

(Q6b) (10 marks)

```
void sumChildren(TreeNode* root) {
    int sum = 0;

    // base case: leaf node
    if (root->left == NULL && root->right == NULL)
        return;

    if (root->left != NULL) {
        sumChildren(root->left);
        sum += root->left->data;
    }

    if (root->right != NULL) {
        sumChildren(root->right);
        sum += root->right->data;
    }

    root->data = sum;
}
```

(Q6c) (10 marks)

```
int sumLevel(TreeNode *root, int level) {
    int sum = 0;

    if (level == 0)
        return root->data;

    if (root->left != NULL)
        sum += sumLevel(root->left, level   1);

    if (root->right != NULL)
        sum += sumLevel(root->right, level - 1);

    return sum;
}
```

- THE END -