3rd Semester (Regular)
Sub & Code DSA/CS-2001
Branch (s) CSE/IT

AUTUMN END SEMESTER EXAMINATION-2016

Data Structure and Algorithm
[CS-2001]

Full Marks: 60 **Time: 3 Hours**

Answer any six questions including question No.1 which is compulsory.
The figures in the margin indicate full marks.
Candidates are required to give their answers in their own words as far as practicable and <u>all parts of a question should be answered at one place only</u>.

1. a) What is the time complexity of the function func()?

```
int func(int n)   {
   int count = 0;
   for (int i = 1; i <= n; i++)   {
      for (int j = 1; j < n; j += i)   {
         count++;
      }
   }
}
```

Ans O(n)

b) Given a single linked list L, write an algorithm/ function to insert an element X after a position P (address of the node) in the list.

Ans:

```
Create a node (named as, node)
node->info = X
node->next = P->next
P->next = node
```

c) Write a function to check whether a linked list is circular linked list or not.

Ans:

```
bool isCircular(struct Node *head)
{
   if (head == NULL)
      return true;
   struct Node *node = head->next;
   while (node != NULL && node != head)
      node = node->next;
   return (node == head);
}
```

d) Convert the expression ((A+B)*C-(D-E)^(F +G)) to equivalent prefix and postfix notations.
Ans: (**1 mark each**)
Prefix: -*+ABC^-DE+FG
Postfix: AB+C*DE-FG+^-

e) The five items: A, B, C, D and E are pushed into a stack, one after the other starting from A. The stack is popped four times and each element is inserted in a queue. Then two elements are deleted from the queue and pushed back onto the stack. Now one item is popped from the stack. What is the item which has been last popped?
Ans: D

f) The inorder and preorder traversal of a binary tree are D B E A F C G and A B D E C F G, respectively. Find the postorder traversal of the binary tree.
Ans: DEBFGCA

g) Write a recursive algorithm to find the number of nodes in a binary tree.

```
int count(*node){
   int c = 1;
   if (node == NULL)
      return 0;
   else{
      c += count(node->left);
      c += count(node->right);
      return c;
    }
}
```

h) Write a pseudo code to minimize the unnecessary comparisons made in bubble sort algorithm.

```
Algorithm BetterBubbleSort(A[0..n − 1])
//The algorithm sorts array A[0..n − 1] by improved bubble sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in ascending order
count ← n − 1              //number of adjacent pairs to be compared
flag ← true                //swap flag
while flag do
        flag ← false
        for j ← 0 to count−1 do
                if A[j + 1] < A[j]
                        swap(A[j], A[j + 1])
                        flag ← true
        count ← count −1
```

i) Consider the array A[]= {65, 45, 85, 15, 35}, apply the insertion sort to sort the array. What is the total number of comparisons required when element 15 reaches the first position of the array?
Ans: 7

j) Consider a hash table with 9 slots. The hash function is h(k) = k mod 9. The collisions are resolved by chaining. The 9 keys are inserted in the order: 50, 37, 64, 24, 29, 42, 57, 45, 73. What are the maximum, minimum chain lengths in the hash table?
Ans: max: 3
Min: 0

2. a) Write a C function to find the elements having minimum difference in a given array.
Ans:
```
int findMinDiff(int arr[], int n) {
  int diff = INT_MAX;
  for (int i=0; i<n-1; i++)
    for (int j=i+1; j<n; j++)
       if (abs(arr[i] - arr[j]) < diff)
            diff = abs(arr[i] - arr[j]);
  return diff;
}
```

b) Describe 3-tuple representation of sparse matrix. Write an algorithm to add two sparse matrices.
Ans:
Describe 3-tuple representation (1 mark)
Algorithm/ function (3 marks)
```
void addsparse(int b1[MAX][3],int b2[MAX][3],int b3[MAX][3]) {
            int t1,t2,i,j,k;
            if(b1[0][0]!=b2[0][0]||b1[0][1]!=b2[0][1]){
                        printf("nYou have entered invalid matrix!!Size must be equal");
                        return;
            }
            t1=b1[0][2];t2=b2[0][2]; i=j=k=0;
            b3[0][0]=b1[0][0]; b3[0][1]=b1[0][1];
            while(i<=t1&&j<=t2){
                        if(b1[i][0]<b2[j][0]) {
                                    b3[k][0]=b1[i][0];
                                    b3[k][1]=b1[i][1];
                                    b3[k][2]=b1[i][2];
                                    k++; i++;}
                        else if(b2[j][0]<b1[i][0]) {
                                    b3[k][0]=b2[j][0];
                                    b3[k][1]=b2[j][1];
                                    b3[k][2]=b2[j][2];
                                    k++; j++; }
                        else if(b1[i][1]<b2[j][1]) {
                                    b3[k][0]=b1[i][0];
                                    b3[k][1]=b1[i][1];
                                    b3[k][2]=b1[i][2];
                                    k++;i++;}
                        else if(b2[j][1]<b1[i][1]) {
```

```
                    b3[k][0]=b2[j][0];
                    b3[k][1]=b2[j][1];
                    b3[k][2]=b2[j][2];
                    k++; j++; }
                else{
                    b3[k][0]=b1[i][0];
                    b3[k][1]=b1[i][1];
                    b3[k][2]=b1[i][2]+b2[j][2];
                    k++; i++; j++;}
            }
            while(i<=t1)  {
                b3[k][0]=b1[i][0];
                b3[k][1]=b1[i][1];
                b3[k][2]=b1[i][2];
                i++;  k++; }
            while(j<=t2)  {
                b3[k][0]=b2[j][0];
                b3[k][1]=b1[j][1];
                b3[k][2]=b1[j][2];
                j++; k++;
            }
            b3[0][2]=k-1;
}
```

Also can be represented by using linked list.

3. a) Write an algorithm/ function to delete the second occurrence of an element X from a double linked list.

Ans:

```
void deleteSecond(int item) {
        struct node *temp, *curr;
        int count = 0;
        temp = curr = head;
        while(curr != NULL){
                if(curr->data == item){
                        count++;
                        temp = curr->prev;}
                if (count == 2) {
                        if(curr->next == NULL)
                                temp->next == NULL;
                        else {
                                curr->next->prev = temp;
                                temp->next = curr->next;
                                free(curr);
                                return;}
                }
                curr = curr->next;
        }
```

```
            if (curr == NULL)
                        printf("Second occurrence of node does not exist");
}
```

b) Write suitable routines/ functions to perform insertion and deletion operations in a deque using linked list.

Ans:

Insertion (2 marks) deletion (2 marks)

```
struct node{
  int data ;
  struct node *next ;
} ;

struct node *front = NULL, *rear = NULL;


void addqatfront(int item) {
  struct node *curr;
  curr = (struct node *) malloc(sizeof(struct node));
  curr -> data = item;
  curr -> next = NULL;
  if (front == NULL)
    rear = front = curr;
  else {
    front->next = curr;
          front = curr;}
 }

void addqatrear(int item) {
  struct node *curr;
  curr = (struct node *) malloc(sizeof(struct node));
  curr -> data = item;
  curr -> next = NULL;
  if (front == NULL)
    rear = front = curr;
  else {
          curr->next = rear;
          rear = curr;}
}

void delqatfront(int *item) {
  struct node *temp = rear, *curr = NULL;
  if (rear == NULL ) {
    printf( "Queue is empty");
    return; }
  else {
          while(temp->next != NULL){
                  curr = temp;
```

```
                    temp = temp -> next; }
            if(curr == NULL) {
                    rear = front = NULL;
            else
                    front = curr;
   }
   *item = temp->data;
   free(temp);
}

void delqatrear(int *item) {
   struct node *temp;
   temp = rear;
   if(rear == NULL) {
      printf("Queue is empty.");
      return; }
   *item = rear->data;
   if(temp->next == NULL)
            rear = front = NULL;
   else
            rear = rear->next;
   free(temp);
}
```

4. a) Write an algorithm to determine whether two binary trees are identical or not.

Ans:

```
bool tree_compare (const node* t1, const node* t2) {
            if (t1 == t2)  return true;
            if ((t1 == NULL) || (t2 == NULL))  return false;
            return ((t1->data == t2->data) && tree_compare (t1->left,  t2->left )
                        && tree_compare (t1->right, t2->right));
}
```

OR

Find inorder and preorder/postorder traversal of both trees and compare the traversals of both trees.

If they are equal then trees are identical.

b) Explain depth first search graph algorithm with necessary data structure.

Ans: **Algorithm (3 marks), Example (1 mark)**

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

It employs the following rules.

- Rule 1 – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- Rule 2 – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- Rule 3 – Repeat Rule 1 and Rule 2 until the stack is empty.

5. a) What is height balanced tree? Construct a balanced tree inserting the elements in sequence: 3, 2, 1, 4, 5, 6, 7, 16, 15, 14. Write the balance factor of every node of the AVL tree that is drawn, to the right of each node.

Ans: **Defn (1 mark), Construction (2 marks), Balance factor(1 mark)**

Height balanced tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

b) Discuss the importance of B-tree and explain the procedure for inserting elements in a B-tree with suitable example.

Ans: **Importance (2 marks), Insertion (2 marks)**

B-Trees

A B-tree is a tree data structure that keeps data sorted and allows searches, insertions, and deletions in logarithmic amortized time. Unlike self-balancing binary search trees, it is optimized for systems that read and write large blocks of data. It is most commonly used in database and file systems.

The B-Tree Rules

Important properties of a B-tree:

- B-tree nodes have many more than two children.
- A B-tree node may contain more than just a single element.

The set formulation of the B-tree rules: Every B-tree depends on a positive constant integer called MINIMUM, which is used to determine how many elements are held in a single node.

Rule 1: The root can have as few as one element (or even no elements if it also has no children); every other node has at least MINIMUM elements.

Rule 2: The maximum number of elements in a node is twice the value of MINIMUM.

Rule 3: The elements of each B-tree node are stored in a partially filled array, sorted from the smallest element (at index 0) to the largest element (at the final used position of the array).

Rule 4: The number of subtrees below a nonleaf node is always one more than the number of elements in the node.
Subtree 0, subtree 1, ...

Rule 5: For any nonleaf node:
An element at index i is greater than all the elements in subtree number i of the node, and
An element at index i is less than all the elements in subtree number i + 1 of the node.

Rule 6: Every leaf in a B-tree has the same depth. Thus it ensures that a B-tree avoids the problem of an unbalanced tree.

Insertion

i. Initialize x as root.
ii. While x is not leaf, do following
   a. Find the child of x that is going to be traversed next. Let the child be y.
   b. If y is not full, change x to point to y.
   c. If y is full, split it and change x to point to one of the two parts of y. If k is smaller

than mid key in y, then set x as first part of y. Else second part of y. When we split y, we move a key from y to its parent x.

ii. The loop in step 2 stops when x is leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.

6. a) Describe the concept of binary search technique. Write a non-recursive algorithm/ function to find the desired item in an array using binary search. Justify, it is efficient as compared to linear search.

Ans: **Describe (1 mark) Algorithm(2 marks) Justification (1 mark)**

In the sequential search, when we compare against the first item, there are at most n−1 more items to look through if the first item is not what we are looking for. Instead of searching the list in sequence, a binary search will start by examining the middle item. If that item is the one we are searching for, we are done. If it is not the correct item, we can use the ordered nature of the list to eliminate half of the remaining items. If the item we are searching for is greater than the middle item, we know that the entire lower half of the list as well as the middle item can be eliminated from further consideration. The item, if it is in the list, must be in the upper half.

```
bool binarySearch(list, item){
        first = 0
        last = len(list)-1
        found = False
        while first<=last and not found {
                midpoint = (first + last)/2
                if (list[midpoint] == item)
                        found = True;
                else
                        if (item < alist[midpoint])
                                last = midpoint-1;
                        else
                                first = midpoint+1;
        }

return found;
}
```

b) Explain how merge sort algorithm works. Show the step-by-step process to arrange the list of elements: 80, 75, 45, 90, 30, 40, 15, 95, 5, 50, 10 in ascending order using merge sort. Also discuss its drawback.

Ans: **Explanation – 2 marks, Steps and drawbacks-2 marks**

Merge sort is based on the divide-and-conquer paradigm.

To sort A[p .. r]:

i. **Divide Step**: If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split A[p .. r] into two subarrays A[p .. q] and A[q + 1 .. r], each containing about half of the elements of A[p .. r]. That is, q is the halfway point of A[p .. r].

ii. **Conquer Step**: Conquer by recursively sorting the two subarrays A[p .. q] and A[q + 1 .. r].

iii. **Combine Step**: Combine the elements back in A[p .. r] by merging the two sorted subarrays A[p .. q] and A[q + 1 .. r] into a sorted sequence. To accomplish this step, define a procedure Merge (A, p, q, r).

```
MergeSort (A, p, r) {
If( p < r) {
        q = (p + r)/2;
        MergeSort (A, p, q)
        MergeSort (A, q + 1, r)
        Merge(A, p, q, r)
}}
```

Drawbacks: additional O(n) memory.

7. a) Illustrate the steps to sort the sequence: 8, 1, 4, 1, 5, 9, 2, 6, 5 by using quicksort, with the middle element as pivot.

Ans:

Do it as per quick sort procedure

b) What is collision in hashing? Explain any two methods to overcome collision problem with suitable example.

Ans:

Hash collision: A situation when the resultant hashes for two or more data elements in the data set U, maps to the same location in the has table, is called a hash collision. In such a situation two or more data elements would qualify to be stored/mapped to the same location in the hash table.

Hash Collision Resolution Techniques:

Open Hashing (**Separate chaining**): Open Hashing, is a technique in which the data is not directly stored at the hash key index (k) of the Hash table. Rather the data at the key index (k) in the hash table is a pointer to the head of the data structure where the data is actually stored. In the most simple and common implementations the data structure adopted for storing the element is a linked-list.

Closed Hashing (**Open Addressing**): In this technique a hash table with pre-identified size is considered. All items are stored in the hash table itself. In addition to the data, each hash bucket also maintains the three states: Empty, Occupied, Deleted. While inserting, if a collision occurs, alternative cells are tried until an empty bucket is found. For which one of the following technique is adopted.

i. Liner Probing
ii. Quadratic probing
iii. Double hashing

8. Short notes on (ALL)

a) Priority queue

Ans:

Priority Queue is an extension of queue with following properties.

i. Every item has a priority associated with it.
ii. An element with high priority is dequeued before an element with low priority.
iii. If two elements have the same priority, they are served according to their order in

the queue.

A typical priority queue supports following operations.

- → insert(item, priority): Inserts an item with given priority.
- → getHighestPriority(): Returns the highest priority item.
- → deleteHighestPriority(): Removes the highest priority item.

Using Array: A simple implementation is to use array of following structure.

```
struct ele {
  int item;
  int priority;
}
```

- ✓ insert() operation can be implemented by adding an item at end of array in O(1) time.
- ✓ getHighestPriority() operation can be implemented by linearly searching the highest priority item in array. This operation takes O(n) time.
- ✓ deleteHighestPriority() operation can be implemented by first linearly searching an item, then removing the item by moving all subsequent items one position back.

b) Divide-and-conquer principle

Ans:

A divide-and-conquer algorithm as having three parts:

i. Divide the problem into a number of subproblems that are smaller instances of the same problem.
ii. Conquer the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
iii. Combine the solutions to the subproblems into the solution for the original problem.

c) Expression Tree

Ans:

Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand. Inorder traversal of expression tree produces infix version of given postfix expression (same with preorder traversal it gives prefix expression).

Evaluating the expression represented by expression tree:

```
Let t be the expression tree
solve(t)
if  t is not null then
     if t.value is operand then
           return  t.value
     A = solve(t.left)
     B = solve(t.right)
      return calculate(A, B, t.value)
```

For constructing expression tree we use a stack. We loop through input expression and do following for every character.

1) If character is operand push that into stack

2) If character is operator pop two values from stack make them its child and push current node again.

At the end only element of stack will be root of expression tree.

d) Threaded Binary Tree

Ans:

Inorder traversal of a binary tree is either be done using recursion or with the use of a stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node.

There are two types of threaded binary trees.

- Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)
- Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

The threads are also useful for fast accessing ancestors of a node.

Following is C representation of a single threaded node.

```
struct Node
{
   int data;
   Node *left, *right;
   bool rightThread;
}
```

Since right pointer is used for two purposes, the boolean variable rightThread is used to indicate whether right pointer points to right child or inorder successor. Similarly, we can add leftThread for a double threaded binary tree.