Course Code	:	**MCS-021**
Course Title	:	**Data and File Structures**
Assignment Number	:	**MCA(3)/021/Assignment/2018-19**
Maximum Marks	:	**100**
Weightage	:	**25%**
Last Dates for Submission	:	**15th October, 2018 (for July, 2018 batch)** **15th April, 2019 (for January, 2019 batch)**

QUESTION 1 :

WRITE AN ALGORITHM THAT ACCEPTS A BINARY TREE AS INPUT AND PRINTS ITS HEIGHT TO STANDARD OUTPUT.

Algorithm:

```
1. If tree is empty then return 0

2. Else

     (a) Get the max depth of left subtree recursively  i.e.,

          call maxDepth( tree->left-subtree)

     (a) Get the max depth of right subtree recursively  i.e.,

          call maxDepth( tree->right-subtree)

     (c) Get the max of max depths of left and right

          subtrees and add 1 to it for the current node.

         max_depth = max(max dept of left subtree,

                             max depth of right subtree) + 1

     (d) Return max_depth
```

Implementation

```
#include<stdio.h>

#include<stdlib.h>
```

```
/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Compute the "maxDepth" of a tree -- the number of
    nodes along the longest path from the root node
    down to the farthest leaf node.*/
int maxDepth(struct node* node)
{
   if (node==NULL)
       return 0;
   else
   {
       /* compute the depth of each subtree */
       int lDepth = maxDepth(node->left);
       int rDepth = maxDepth(node->right);

       /* use the larger one */
       if (lDepth > rDepth)
           return(lDepth+1);
       else return(rDepth+1);
   }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
```

```
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                                malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

int main()
{
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Hight of tree is %d", maxDepth(root));

    getchar();
    return 0;
}
```


QUESTION 2:

WRITE AN ALGORITHM FOR THE IMPLEMENTATION OF A B TREE.

B-Tree can be defined as follows...

B-Tree is a self-balanced search tree with multiple keys in every node and more than two children for every node.

Here, number of keys in a node and number of children for a node is depend on the order of the B-Tree. Every B-Tree has order.

B-Tree of Order m has the following properties...

- **Property #1** - All the **leaf nodes** must be **at same level**.
- **Property #2** - All nodes except root must have at least **[m/2]-1** keys and maximum of **m-1** keys.
- **Property #3** - All non leaf nodes except root (i.e. all internal nodes) must have at least **m/2** children.
- **Property #4** - If the root node is a non leaf node, then it must have **at least 2** children.
- **Property #5** - A non leaf node with **n-1** keys must have **n** number of children.
- **Property #6** - All the **key values within a node** must be in **Ascending Order**.

Operations on a B-Tree

The following operations are performed on a B-Tree...

1. **Search**
2. **Insertion**
3. **Deletion**

Search Operation in B-Tree

In a B-Ttree, the search operation is similar to that of Binary Search Tree. In a Binary search tree, the search process starts from the root node and every time we make a 2-way decision (we go to either left subtree or right subtree). In B-Tree also search process starts from the root node but every time we make n-way decision where n is the total number of children that node has. In a B-Ttree, the search operation is performed with **O(log n)** time complexity. The search operation is performed as follows...

- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with first key value of root node in the tree.
- **Step 3:** If both are matching, then display "Given node found!!!" and terminate the function
- **Step 4:** If both are not matching, then check whether search element is smaller or larger than that key value.
- **Step 5:** If search element is smaller, then continue the search process in left subtree.

- **Step 6:** If search element is larger, then compare with next key value in the same node and repeate step 3, 4, 5 and 6 until we found exact match or comparision completed with last key value in a leaf node.
- **Step 7:** If we completed with last key value in a leaf node, then display "Element is not found" and terminate the function.

Insertion Operation in B-Tree

In a B-Tree, the new element must be added only at leaf node. That means, always the new keyValue is attached to leaf node only. The insertion operation is performed as follows...

- **Step 1:** Check whether tree is Empty.
- **Step 2:** If tree is **Empty**, then create a new node with new key value and insert into the tree as a root node.
- **Step 3:** If tree is **Not Empty**, then find a leaf node to which the new key value cab be added using Binary Search Tree logic.
- **Step 4:** If that leaf node has an empty position, then add the new key value to that leaf node by maintaining ascending order of key value within the node.
- **Step 5:** If that leaf node is already full, then **split** that leaf node by sending middle value to its parent node. Repeat tha same until sending value is fixed into a node.
- **Step 6:** If the spilting is occuring to the root node, then the middle value becomes new root node for the tree and the height of the tree is increased by one.

QUESTION 3:

WRITE A NOTE OF NOT MORE THAN 5 PAGES SUMMARIZING THE LATEST RESEARCH IN THE AREA OF "SEARCHING TECHNIQUES". REFER TO VARIOUS JOURNALS AND OTHER ONLINE RESOURCES. INDICATE THEM IN YOUR ASSIGNMENT.

INTRODUCTION

Searching for an element and arranging the elements in a particular order are the everyday jobs. Search technology covers a large area. For the "big players" google and bing they're changing things every day. For google you can find these changes at Google Groupsfor Bing http://help.bing.microsoft.com.

At the level of a single organization open source tools and their commercial compliments are becoming more capable very rapidly. Apache Solr and Elasticsearch are both Apache 2 licensed (open source) and can scale to (at least) Billions of documents across around dozen machines (maybe far fewer depending on document structure and size). Solr has been implementing statistical capabilities into the search, and is now enhancing it's Natural Language Capabilities. Elastic has been more focused on making search easier and faster for things that look like log files.

SEARCHING ALGORITHMS

Searching technique is used to find the location of any data item in given list of data items. The searching techniques can be categorized into two parts:

Internal Searches: if file or table is kept in primary memory this type of searching is called internal search.

External Searches: if file or table is stored in secondary memory (hard disk, floppy, tape etc.) this type of searching is called External Search.

LINEAR SEARCH

It is a simple search algorithm. In this a sequential search is made over all data items one after other. Every data item is scanned and if the match is found then that particular data item is sent back, else the search continues till the list ends, shown in Fig. 12.

- Worst Case: Θ(n)Θ(n); search key not present or last element
- Best Case: Θ(1)Θ(1) ; first element
- No. of comparisons: Θ(n)Θ(n) in worst case & 11 in best case

Algorithm

Linear Search (Array B, Value y)

1. Assign 1 to i
2. if i> n, move to step 7
3. if B[i] = y, move to step 6
4. now assign i + 1 to i
5. move to Step-2

6. print the value of y found at location I and move to step 8
7. Print element not found
8. Exit

Advantages

- It is simple and conventional method of searching data. The linear or sequential name implies that the items are stored in a systematic manner.
- The elements in the list can be in any order. i.e. The linear search can be applied on sorted or unsorted linear data structure.

Disadvantage

- This method is insufficient when large number of elements is present in list.
- It consumes more time and reduces the retrieval rate of the system.

BINARY SEARCH

It is a fast search algorithm as the run-time complexity is O(log n). Divide and conquer is the basic principle behind this search algorithm. The data collection must be in the sorted form for better functioning of this algorithm. It searches for a particular data item by equating it to the middle most data item of the collection. If match is found, then the location of data item is returned. If the middle data item is bigger than the search element, then the data item is searched to the left of the middle data item. Else it searched to the right of the middle data item. This process is repeated until the size of the sub array reduces to zero. The objective of this search is to use the information that the array is sorted and reduce the time complexity to O(Logn), shown in Fig. 13.

- Worst case/Average case : Θ(logn)Θ(logn)
- Best Case : Θ(1)Θ(1) ; when key is middle element
- No. of comparisons : Θ(logn)Θ(logn) in worst/average case & 11 in best case

Algorithm

1. Assign n-1 to max and 0 to min.

2. Compute mid as the mean of max and min.
3. If array[mid] is equal to the search element, then stop and return mid.
4. If the mid is less than the search element then assign mid+1 to min.
5. If mid is greater than search element then assign mid-1 to max.
6. Go back to step 2.

Advantages

- The binary search is fast as compared to linear search.
- Time complexity is high in both average and worst case.
- Most suitable for sorted arrays.

Disadvantages

- The main disadvantage of binary search is that it works only with an array of sorted order.

HASH SEARCH

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data. Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from. The time complexity is O(n), shown in Fig. 14.

Algorithm

1. Get the key k to be searched
2. Set counter j = 0
3. Compute hash function h[k] = k % SIZE
4. If the key space at hashtable[h[k]] is occupied

(4.1) Compare the element at hashtable[h[k]] with the key k.

(4.2) If they are equal

(4.2.1) The key is found at the bucket h[k]

(4.2.2) Stop
Else

(4.3) The element might be placed at the next location given by the quadratic function

(4.4) Increment j

(4.5) Set h[k] = (k + (j * j)) % SIZE, so that we can probe the bucket at a new slot, h[k].

(4.6) Repeat Step 4 till j is greater than SIZE of hash table

5. The key was not found in the hash table

6. Stop.

Advantages

- Hashing provides a more reliable and flexible method of data retrieval than any other data structure.
- Hash tables are efficient when large number of data items can be predicted in advance, so that the bucket array can be allocated once with the optimum size and never resized.

Disadvantages

- Hash tables are not effective when the number of entries is very small
- Listing all n entries in some specific order generally requires a separate sorting step

INTERPOLATION SEARCH

Interpolation search algorithm is improvement over Binary search. The binary search checks the element at middle index. But interpolation search may search at different locations based on value of the search key. The elements must be in sorted order in order to implement interpolation search, shown in Fig 15. The time complexity of interpolation search

- Best case: O(log(log(n)))
- Worst case: O(n)

Advantages

- It requires less time when compared to binary search
- Number of steps required to search an element will be comparatively less than binary search.

Disadvantages

- The calculation of x is complicated and requires more time.
- The elements must be in sorted order.

EXPONENTIAL SEARCH

Exponential search is also called Galloping search, doubling search. Its implementation is similar to implementation of binary search. As the name refers the search starts from first element and continues by doubling the upper limit of the range till the element at that position is larger than the search key. Then it either searches the key using binary search or starts another galloping. The time complexity of exponential search is O(log(i)), where i is the value of index, shown in Fig 15.

Algorithm

1. Initialize the index value to 1.
2. Check the index value with the search key.
3. If the index value is greater than the value of search key.
4. Else, double the value of index till the value of element at the index position is greater than the value of search key.
5. Apply the binary search technique on the array of elements to find the search key.

Advantages

- Exponential search can be used for infinite set of elements.
- It can be applied on large set of data items.
- The range\size of the array can be calculated if it is unknown.

TERNARY SEARCH

Ternary search is a searching algorithm where the search key is searched by dividing the given array into three parts. The implementation is similar to binary search. The time complexity is O (log (n)). It works on the principle of divide and conquer technique, shown in Fig 17.

Algorithm

1. Initialize the values of start to 0 and end to n-1, where n is the size of array.
2. Calculate mid1=start+(end-1)/3, mid2=end-(end-1)/3
3. Check if the search key is equal to array[mid1] or array[mid2].
4. If the search key is found to be equal,then stop the process.
5. If key<array[mid1], assign end to mid1-1 or if key>array[mid1], assign start to mid1+1 and repeat step 2.
6. Repeat the step 5 for mid2

Advantages

- It has more space efficiency.
- Comparatively less comparisons are done in best cases than in binary search.

QUESTION 4.

WRITE AN ALGORITHM FOR THE IMPLEMENTATION OF A CIRCULARLY DOUBLY LINKED LIST.

```
/*
 * C Program to Implement Circular Doubly Linked List
 */
#include <stdio.h>
#include <stdlib.h>

struct node
```

```
{
    int val;
    struct node *next;
    struct node *prev;
};
typedef struct node n;

n* create_node(int);
void add_node();
void insert_at_first();
void insert_at_end();
void insert_at_position();
void delete_node_position();
void sort_list();
void update();
void search();
void display_from_beg();
void display_in_rev();

n *new, *ptr, *prev;
n *first = NULL, *last = NULL;
int number = 0;

void main()
{
    int ch;

    printf("\n linked list\n");
    printf("1.insert at beginning \n 2.insert at end\n 3.insert at position\n4.sort linked lis
t\n 5.delete node at position\n 6.updatenodevalue\n7.search element \n8.displaylist from begin
ning\n9.display list from end\n10.exit ");

    while (1)
    {

        printf("\n enter your choice:");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1 :
            insert_at_first();
            break;
        case 2 :
            insert_at_end();
            break;
        case 3 :
            insert_at_position();
            break;
        case 4 :
            sort_list();
            break;
        case 5 :
            delete_node_position();
```

```
            break;
        case 6 :
            update();
            break;
        case 7 :
            search();
            break;
        case 8 :
            display_from_beg();
            break;
        case 9 :
            display_in_rev();
            break;
        case 10 :
            exit(0);
        case 11 :
            add_node();
            break;
        default:
            printf("\ninvalid choice");
        }
    }
}
/*
 *MEMORY ALLOCATED FOR NODE DYNAMICALLY
 */
n* create_node(int info)
{
    number++;
    new = (n *)malloc(sizeof(n));
    new->val = info;
    new->next = NULL;
    new->prev = NULL;
    return new;
}
/*
 *ADDS NEW NODE
 */
void add_node()
{

    int info;

    printf("\nenter the value you would like to add:");
    scanf("%d", &info);
    new = create_node(info);

    if (first == last && first == NULL)
    {

        first = last = new;
        first->next = last->next = NULL;
        first->prev = last->prev = NULL;
```

```
    }
    else
    {
        last->next = new;
        new->prev = last;
        last = new;
        last->next = first;
        first->prev = last;
    }
}
/*
 *INSERTS ELEMENT AT FIRST
 */
void insert_at_first()
{

    int info;

    printf("\nenter the value to be inserted at first:");
    scanf("%d",&info);
    new = create_node(info);

    if (first == last && first == NULL)
    {
        printf("\ninitially it is empty linked list later insertion is done");
        first = last = new;
        first->next = last->next = NULL;
        first->prev = last->prev = NULL;
    }
    else
    {
        new->next = first;
        first->prev = new;
        first = new;
        first->prev = last;
        last->next = first;
        printf("\n the value is inserted at begining");
    }
}
/*
 *INSERTS ELEMNET AT END
 */
void insert_at_end()
{

    int info;

    printf("\nenter the value that has to be inserted at last:");
    scanf("%d", &info);
    new = create_node(info);

    if (first == last && first == NULL)
    {
```

```
        printf("\ninitially the list is empty and now new node is inserted but at first");
        first = last = new;
        first->next = last->next = NULL;
        first->prev = last->prev = NULL;
    }
    else
    {
        last->next = new;
        new->prev = last;
        last = new;
        first->prev = last;
        last->next = first;
    }
}
/*
 *INSERTS THE ELEMENT AT GIVEN POSITION
 */
void insert_at_position()
{
    int info, pos, len = 0, i;
    n *prevnode;

    printf("\n enter the value that you would like to insert:");
    scanf("%d", &info);
    printf("\n enter the position where you have to enter:");
    scanf("%d", &pos);
    new = create_node(info);

    if (first == last && first == NULL)
    {
        if (pos == 1)
        {
            first = last = new;
            first->next = last->next = NULL;
            first->prev = last->prev = NULL;
        }
        else
            printf("\n empty linked list you cant insert at that particular position");
    }
    else
    {
        if (number < pos)
            printf("\n node cant be inserted as position is exceeding the linkedlist length");

        else
        {
            for (ptr = first, i = 1;i <= number;i++)
            {
                prevnode = ptr;
                ptr = ptr->next;
                if (i == pos-1)
                {
                    prevnode->next = new;
```

```
                    new->prev = prevnode;
                    new->next = ptr;
                    ptr->prev = new;
                    printf("\ninserted at position %d succesfully", pos);
                    break;
                }
            }
        }
    }
}
/*
 *SORTING IS DONE OF ONLY NUMBERS NOT LINKS
 */
void sort_list()
{
    n *temp;
    int tempval, i, j;

    if (first == last && first == NULL)
        printf("\nlinked list is empty no elements to sort");
    else
    {
        for (ptr = first,i = 0;i < number;ptr = ptr->next,i++)
        {
            for (temp = ptr->next,j=i;j<number;j++)
            {
                if (ptr->val > temp->val)
                {
                    tempval = ptr->val;
                    ptr->val = temp->val;
                    temp->val = tempval;
                }
            }
        }
        for (ptr = first, i = 0;i < number;ptr = ptr->next,i++)
            printf("\n%d", ptr->val);
    }
}
/*
 *DELETION IS DONE
 */
void delete_node_position()
{
    int pos, count = 0, i;
    n *temp, *prevnode;

    printf("\n enter the position which u wanted to delete:");
    scanf("%d", &pos);

    if (first == last && first == NULL)
        printf("\n empty linked list you cant delete");

    else
```

```
    {
        if (number < pos)
            printf("\n node cant be deleted at position as it is exceeding the linkedlist leng
th");

        else
        {
            for (ptr = first,i = 1;i <= number;i++)
            {
                prevnode = ptr;
                ptr = ptr->next;
                if (pos == 1)
                {
                    number--;
                    last->next = prevnode->next;
                    ptr->prev = prevnode->prev;
                    first = ptr;
                    printf("%d is deleted", prevnode->val);
                    free(prevnode);
                    break;
                }
                else if (i == pos - 1)
                {
                    number--;
                    prevnode->next = ptr->next;
                    ptr->next->prev = prevnode;
                    printf("%d is deleted", ptr->val);
                    free(ptr);
                    break;
                }
            }
        }
    }
}
/*
 *UPDATION IS DONE FRO GIVEN OLD VAL
 */
void update()
{
    int oldval, newval, i, f = 0;
    printf("\n enter the value old value:");
    scanf("%d", &oldval);
    printf("\n enter the value new value:");
    scanf("%d", &newval);
    if (first == last && first == NULL)
        printf("\n list is empty no elemnts for updation");
    else
    {
        for (ptr = first, i = 0;i < number;ptr = ptr->next,i++)
        {
            if (ptr->val == oldval)
            {
                ptr->val = newval;
```

```
                printf("value is updated to %d", ptr->val);
                f = 1;
            }
        }
        if (f == 0)
            printf("\n no such old value to be get updated");
    }
}
/*
 *SEARCHING USING SINGLE KEY
 */
void search()
{
    int count = 0, key, i, f = 0;

    printf("\nenter the value to be searched:");
    scanf("%d", &key);

    if (first == last && first == NULL)
        printf("\nlist is empty no elemnets in list to search");
    else
    {
        for (ptr = first,i = 0;i < number;i++,ptr = ptr->next)
        {
            count++;
            if (ptr->val == key)
            {
                printf("\n the value is found at position at %d", count);
                f = 1;
            }
        }
        if (f == 0)
            printf("\n the value is not found in linkedlist");
    }
}
/*
 *DISPLAYING IN BEGINNING
 */
void display_from_beg()
{
    int i;
    if (first == last && first == NULL)
        printf("\nlist is empty no elemnts to print");
    else
    {
        printf("\n%d number of nodes are there", number);

        for (ptr = first, i = 0;i < number;i++,ptr = ptr->next)
            printf("\n %d", ptr->val);
    }
}
/*
```

```
 * DISPLAYING IN REVERSE
 */
void display_in_rev()
{
    int i;
    if (first == last && first == NULL)
        printf("\nlist is empty there are no elments");
    else
    {
        for (ptr = last, i = 0;i < number;i++,ptr = ptr->prev)
        {
            printf("\n%d", ptr->val);
        }
    }
}
```