Programme Name: **BCS HONS**

Course Code: **CSC 2516**

Course Name: **Data Structure and Algorithm**

Open Book Examination

Date of Submission: **9/28/2021**

Submitted By:

Student Name**: Dipesh Tha Shrestha**

IUKL ID: **041902900028**

Semester**: Fourth Semester**

Intake**: September 2019**

Submitted To:

Faculty Name**: Prakash Chandra Sir**

Department**: LMS**

1.

a. **Explain the concept of binary search algorithm with an example.**

Answer:

A binary search is a type of advanced search algorithm for finding and retrieving data from a sorted list of items. Its main working principle is to divide the data in the list in half until the required value is found and displayed in the search result to the user. Since it follows the technique to eliminate half of the array elements, it is more efficient as compared to linear search for large data. Binary search algorithm finds a given element in a list of elements with Working with the principle of divide and conquer, this search algorithm can be quite fast, but the caveat is that the data has to be in a sorted form. It works by starting the search in the middle of the array and working going down the first lower or upper half of the sequence. If the median value is lower than the target value, that means that the search needs to go higher, if not, then it needs to look on the descending portion of the array.

The binary search works in the following manner:

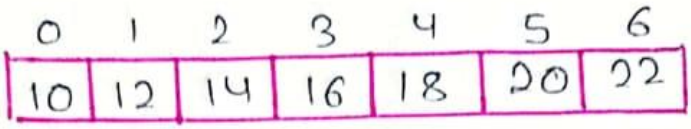
- The search process initiates by locating the middle element of the sorted array of data
- After that, the key value is compared with the element
- If the key value is smaller than the middle element, then searches analyses the upper values to the middle element for comparison and matching
- In case the key value is greater than the middle element then searches analyses the lower values to the middle element for comparison and matching

Example

Q No 1 .

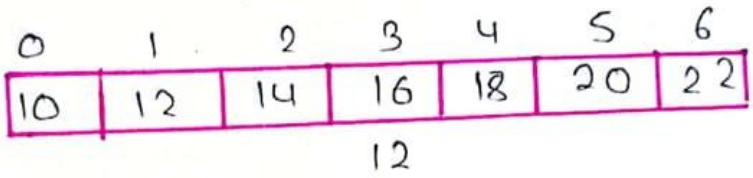
Example:

Let us consider the following list of element are to be search.

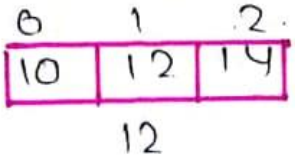


Step 1:

Search element (12) is compared with middle element (16)



Both are not matching. And 12 is smaller than 16 . So we search only in the left sublist (10, 12, 14).



Both are matching. So the result is "element found at index 1".

Therefore, The above is the example of Binary Search Algorithm.

b. **Determine efficiency of binary search algorithm for the example discussed above.**

Answer:

Binary search algorithm finds a given element in a list of elements with O(log n) time complexity where n is total number of elements in the list. The binary search algorithm can be used with only a sorted list of elements. That means the binary search is used only with a list of elements that are already arranged in an order. The binary search cannot be used for a list of elements arranged in random order. This search process starts comparing the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for the left sub list of the middle element. If the search element is larger, then we repeat the same process for the right sub list of the middle element. We repeat this process until we find the search element in the list or until we left with a sub list of only one element. And if that element also doesn't match with the search element, then the result is "Element not found in the list". Binary search works efficiently on sorted data no matter the size of the data. Instead of performing the search by going through the data in a sequence, the binary algorithm randomly accesses the data to find the required element. This makes the search cycles shorter and more accurate. Binary search performs comparisons of the sorted data based on an ordering principle than using equality comparisons, which are slower and mostly inaccurate. After every cycle of search, the algorithm divides the size of the array into half hence in the next iteration it will work only in the remaining half of the array.

2.

a. **Convert the following infix expression into a postfix expression using stack with all the required steps:**

i. (A+B) * C - (D-E) * (F+G)

Q No 2

(a)

$(A+B)^{*}C-(D-E)^{*}(F+G)$

S.N	Scanned	Stack	Postfix Expression	Description
1		(		Start
2	A	(	A	
3	+	(+	A	
4	B	(+	AB	
5	)	(	AB+	Pop from stack when right Paranthese are encounter
6	*	(*	AB+	
7	C	(*	AB+C	
8	-	(-	AB+C*	(*) has higher precedence
9	(	(-(	AB+C*	
10	D	(-(	AB+C*D	
11	-	(-(-	AB+C*D	
12	E	(-(-	AB+C*DE	
13	)	(-	AB+C*DE-	Pop from stack()
14	*	C-*	AB+C*DE-	
15	(	(-*(	AB+C*DE-	
16	F	(-*(	AB+C*DE-F	
17	+	(-*(+	AB+C*DE-F	
18	G	(-*(+	AB+C*DE-FG	
19	)	(-*	AB+C*DE-FG+	
20	)	EMPTY	AB+C*DE-FG+*-	END

∴ The Postfix of $(A+B)^{*}C-(D-E)^{*}(F+G)$ = AB+C*DE-FG+*-

b. **Write C-code to implement all the operations of STACK using a linked list.**

Code:

```
#include<stdio.h>
#include<stdlib.h>

struct Node
{
int data;
struct Node *next;
}*top = NULL;

void push(int);
void pop();
void display();

int main()
{
int choice, value;
printf("\nIMPLEMENTING STACKS USING LINKED LISTS\n");
while(1){
printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
printf("\nEnter your choice : ");
scanf("%d",&choice);
switch(choice)
{
case 1: printf("\nEnter the value to insert: ");
scanf("%d", &value);
push(value);
break;

case 2: pop();
break;

case 3: display();
break;

case 4: exit(0);
break;

default: printf("\nInvalid Choice\n");
}}}

void push(int value)
{
```

```
struct Node *newNode;
newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = value;
if(top == NULL)
newNode->next = NULL;
else
newNode->next = top;
top = newNode;
printf("Node is Inserted\n\n");
}

void pop()
{
if(top == NULL)
printf("\nEMPTY STACK\n");
else{
struct Node *temp = top;
printf("\nPopped Element : %d", temp->data);
printf("\n");
top = temp->next;
free(temp);
}}

void display()
{

if(top == NULL)
printf("\nEMPTY STACK\n");
else
{
printf("The stack is \n");
struct Node *temp = top;
while(temp->next != NULL){
printf("%d--->",temp->data);
temp = temp -> next;
}
printf("%d--->NULL\n\n",temp->data);
}}
```

Output:

```
PS C:\Fourth Semester\Data Structure and Algorithms
ing c\singly linked list\" ; if ($?) { gcc stackusi


IMPLEMENTING STACKS USING LINKED LISTS
1. Push
2. Pop
3. Display
4. Exit

Enter your choice : 1

Enter the value to insert: 12
Node is Inserted

1. Push
2. Pop
3. Display
4. Exit

Enter your choice : 1

Enter the value to insert: 34
Node is Inserted

1. Push
2. Pop
3. Display
4. Exit

Enter your choice : 2
```

```
Enter your choice : 2

Popped Element : 34
1. Push
2. Pop
3. Display
4. Exit

Enter your choice : 12

Invalid Choice
1. Push
2. Pop
3. Display
4. Exit

Enter your choice : 1

Enter the value to insert: 45
Node is Inserted

1. Push
2. Pop
3. Display
4. Exit

Enter your choice : 3
The stack is
45--->12--->NULL
```

3.

A. Explain the concept of circular queue, and its basic operations.

Answer:

Circular Queue is a linear data structure in which operations are carried out according to the FIFO (First In First Out) principle, and the last position is connected back to the first to form a circle. Circular Queues work on the principle of circular increment, which means that when we try to increment the pointer and reach the end of the queue, we go back to the beginning. The circular queue provides memory management, the memory is managed efficiently by placing the elements in a location which is unused. The operating system also uses the circular queue to insert the processes and then execute them.

The basic operations that can be performed on a circular queue are given below:

Front: It is used to get the front element from the Queue. This is similar to the peek operation in stacks, it returns the value of the element at the front without removing it.

Rear: It is used to get the rear element from the Queue.

Enqueue: This function is used to insert the new value in the Queue. The new element is always inserted from the rear end.

Dequeue: Dequeue means removing an element from the queue. Since queue follows the FIFO principle we need to remove the element of the queue which was inserted at first. Naturally, the element inserted first will be at the front of the queue so we will remove the front element and let the element behind it be the new front element.

B. Write C-code to Implement circular queue with following operation

i. enqueue( )

ii. dequeue( )

iii. display()

Code:

```
# include<stdio.h>
# define MAX 5

int cqueue_arr[MAX];
int front = -1;
int rear = -1;

void insert(int item)
{
   if((front == 0 && rear == MAX-1) || (front == rear+1))
   {
      printf("Queue Overflow \n");
      return;
```

```
    }
    if (front == -1)
    {
        front = 0;
        rear = 0;
    }
    else
    {
        if(rear == MAX-1)
            rear = 0;
        else
            rear = rear+1;
    }
    cqueue_arr[rear] = item ;
}
void del()
{
    if (front == -1)
    {
        printf("Queue Underflow\n");
        return ;
    }
    printf("Element deleted from queue is : %d\n",cqueue_arr[front]);
    if(front == rear)
    {
        front = -1;
        rear=-1;
    }
    else
    {
        if(front == MAX-1)
            front = 0;
        else
            front = front+1;
    }
}
void display()
{
    int front_pos = front,rear_pos = rear;
    if(front == -1)
    {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements :\n");
```

```
    if( front_pos <= rear_pos )
        while(front_pos <= rear_pos)
        {
            printf("%d ",cqueue_arr[front_pos]);
            front_pos++;
        }
    else
    {
        while(front_pos <= MAX-1)
        {
            printf("%d ",cqueue_arr[front_pos]);
            front_pos++;
        }
        front_pos = 0;
        while(front_pos <= rear_pos)
        {
            printf("%d ",cqueue_arr[front_pos]);
            front_pos++;
        }
    }
    printf("\n");
}
int main()
{
    int choice,item;
    do
    {
        printf("1.Insert\n");
        printf("2.Delete\n");
        printf("3.Display\n");
        printf("4.Quit\n");

        printf("Enter your choice : ");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1 :
                printf("Input the element for insertion in queue : ");
                scanf("%d", &item);

                insert(item);
                break;
            case 2 :
                del();
```

```
            break;
        case 3:
            display();
            break;
        case 4:
            break;
            default:
            printf("Wrong choice\n");
      }
   }while(choice!=4);

   return 0;
}
```

Output:

```
PS C:\Fourth Semester\Data Structure and Algorithms\programmi
ing c\circularqueu\" ; if ($?) { gcc circularqueue.c -o circu
1.Insert
2.Delete
3.Display
4.Quit
Enter your choice : 1
Input the element for insertion in queue : 12
1.Insert
2.Delete
3.Display
4.Quit
Enter your choice : 1
Input the element for insertion in queue : 34
1.Insert
2.Delete
3.Display
4.Quit
Enter your choice : 3
Queue elements :
12 34
1.Insert
2.Delete
3.Display
4.Quit
Enter your choice : 2
Element deleted from queue is : 12
1.Insert
2.Delete
3.Display
4.Quit
Enter your choice : 3
Queue elements :
34
```

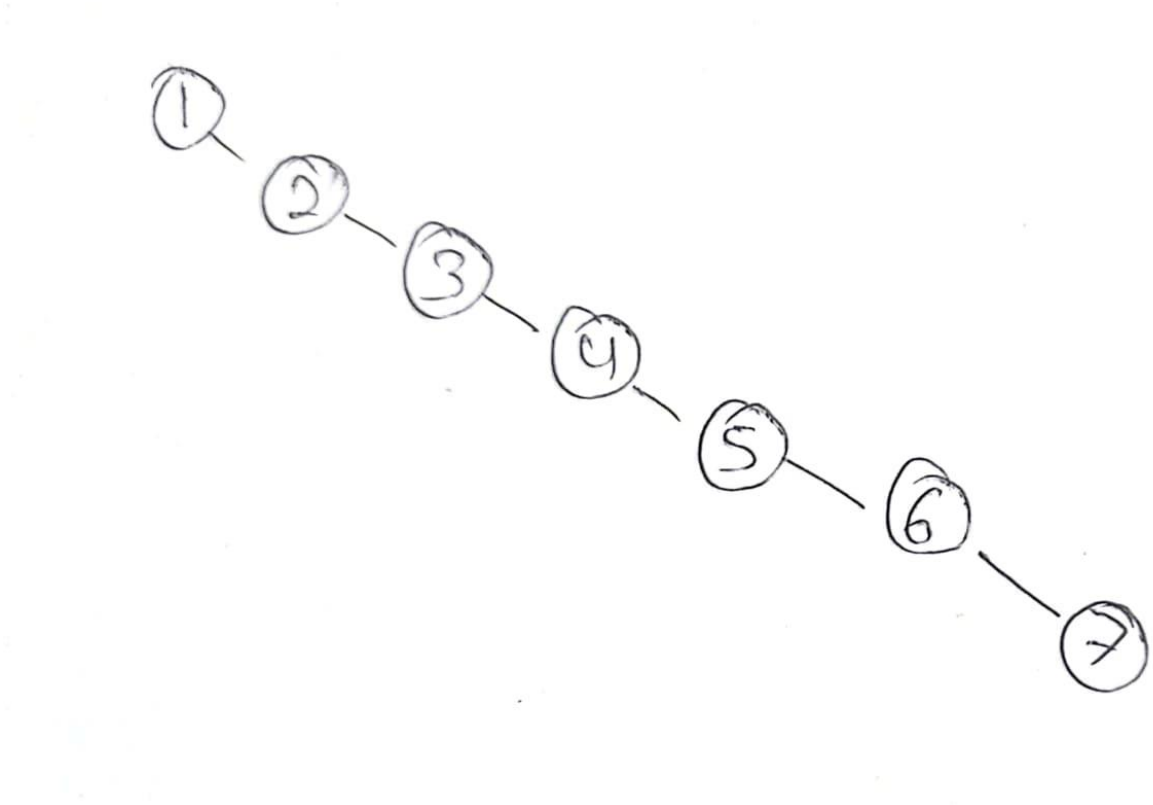
4.

a. **Explain the need and importance of AVL tree with an example.**

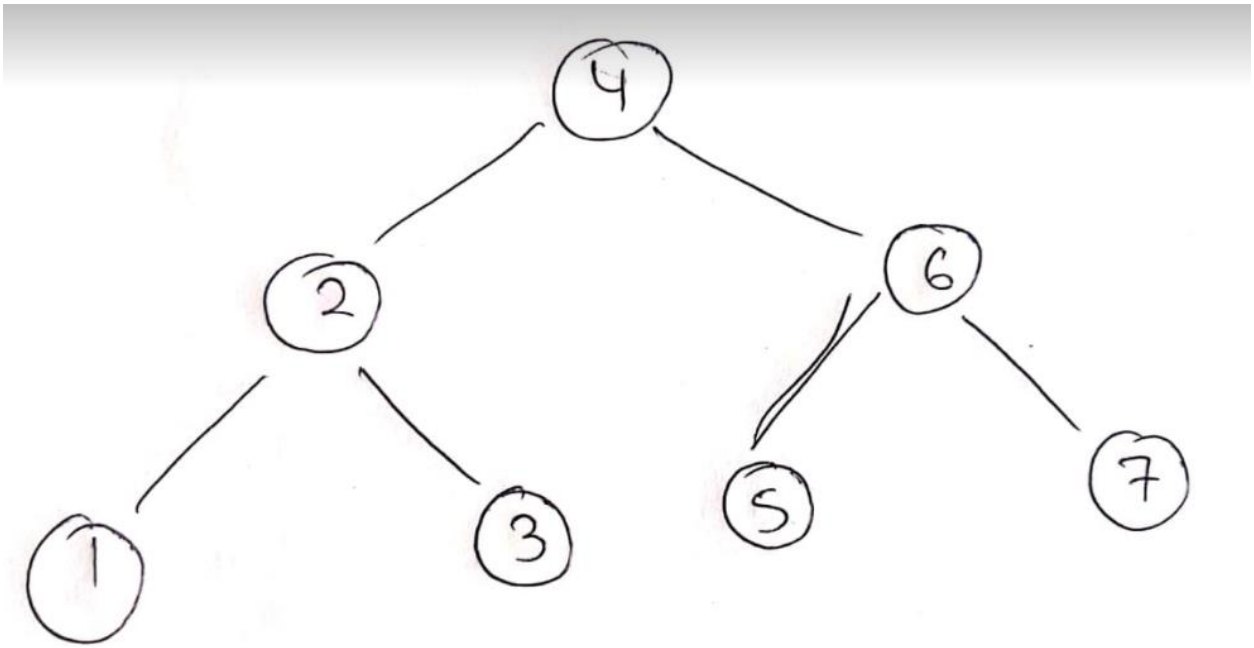
Answer:

AVL 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. The majority of BST operations (such as search, max, min, insert, delete, and so on) take O(h) time, where h is the BST's height. For a skewed Binary tree, the cost of these operations may become O(n). We can guarantee an upper bound of O(Logn) for all of these operations if we ensure that the tree's height remains O(Logn) after each insertion and deletion. An AVL tree's height is always O(Logn), where n denotes the number of nodes in the tree.

The Insertion time in BST gives the impression that it is O(log n), but it is not always log n. Consider a Binary Search Tree that is created by inserting nodes in sorted order (Say Ascending order). This is how the tree will appear:



The above tree is clearly a Binary Search Tree, and it appears to be as good as a List or any Linear Data Structure, so the search takes linear time. In AVL trees, however, we balance the binary search tree whenever a node is added. As a result, for the same nodes, the AVL tree will look like this:



As you can see. This a perfectly balanced BST, which takes no longer than O(log n) time for search operation. So when N is very large number AVL trees have significant advantage over Binary Search Trees.

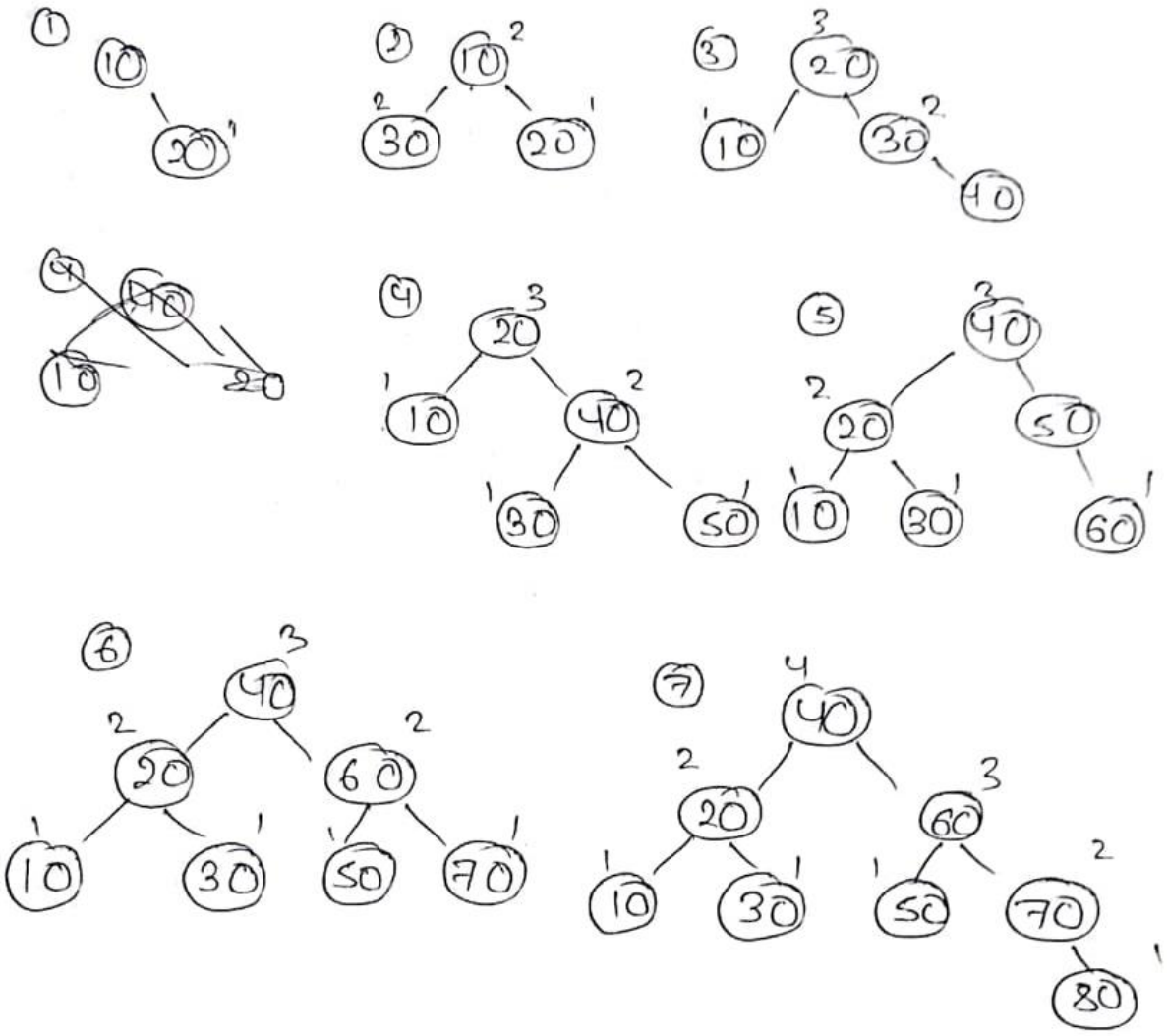
b. **Use the following list of numbers to construct height balanced AVL tree**
10,20,30,40,50,60,70,80
NOTE: show all the insertion and rotation steps.

Dipesh Tra Shrestha.

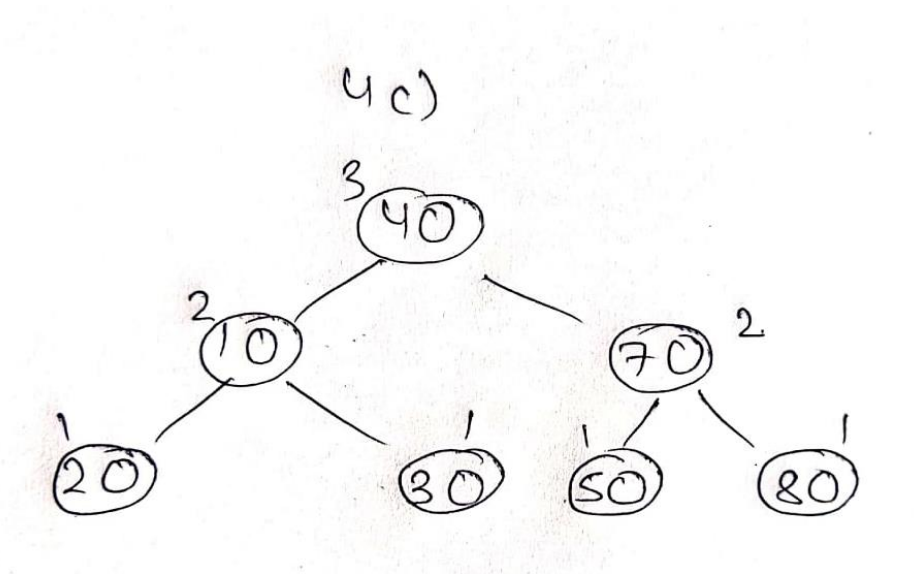
4b

AVL Tree.

Given data 10, 20, 30, 40, 50, 60, 70, 80



c. **Delete the node with value 60 from the AVL tree constructed in question**



Thank You