
VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

**School of Computer Science and Engineering (SCOPE)**

# CSE2011-DATASTRUCTURES LAB RECORD

# FALL 21-22

| | | |
|---|---|---|
| **Name of the Student** | **:** | **Mayank Gupta** |
| **Register Number** | **:** | **20BCE1538** |
| **Slot** | **:** | **L27+L28** |
| **Faculty Name** | **:** | **Dr.V.Vani** |

## INDEX

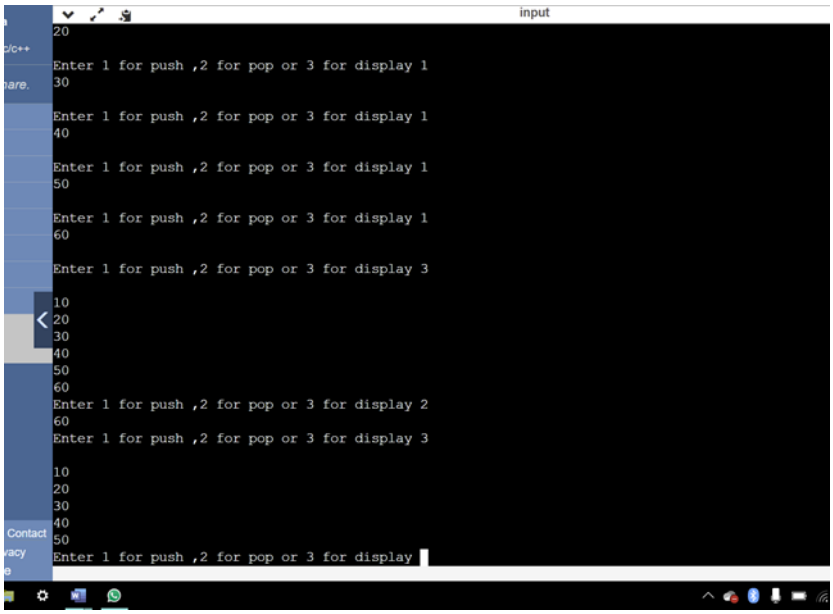
| LIST OF EXPERIMENTS | | |
|---|---|---|
| **S.No** | **Name of the Experiment** | **Page No.** |
| 1. | Implementation of Stack Data Structure | |
| 2. | Implementation of Queue Data Structure | |
| 3. | Implementation of Linked List Data Structure | |
| 4. | Implementation of Searching Algorithms | |
| 5. | Implementation of Sorting Algorithms: Insertion, Bubble and Selection | |
| 6. | Randomized Quick and Merge Sort | |
| 7. | Binary Tree Traversals | |
| 8. | Binary Search Tree | |
| 9. | Depth First Search and Breadth First Search | |
| 10. | Minimum Spanning Tree Algorithm: Prim's and Kruskal | |
| 11. | Single Source Shortest Path Algorithm | |
| **CONTENT BEYOND THE SYLLABUS** | | |
| 1. | Hashing | |

# AIM Implementation of Stack Data Structure

## CODE

```
#include <stdio.h>
int n = 10;
int stack[10];
int top = -1;
int push ()
{
 int X;
 if(top > n-1)
 printf ("stack is full");
 else{
  scanf ("%d",&X) ;
  top++;
  stack[top]=X;}
}

int pop ()
{

 if (top==-1)
 printf("stack is empty");
 else{

    top --;
    printf("%d",stack[top+1]);

 }
}

int display ()
{
 for (int i=0 ; i<=top ; i++)
 printf ("\n%d",stack[i]);
}
```

```
int
main ()
{
 int j=0;
 while (j != 4){
 printf ("\nEnter 1 for push ,2 for pop or 3 for display ");
 scanf ("\n%d", &j);

 switch (j)
  {
  case 1:
    push ();
    break;
  case 2:
    pop ();
    break;
  case 3:
    display ();
    break;
  }
 }
return 0;
}
```



## *isfull()*

Algorithm of isfull() function –

```
begin procedure isfull

  if top equals to MAXSIZE
    return true
  else
    return false
  endif

end procedure
```

## *isempty()*

Algorithm of isempty() function –

```
begin procedure isempty

  if top less than 1
    return true
  else
    return false
  endif

end procedure
```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code –

## *Push Operation*

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

- Step 1 – Checks if the stack is full.
- Step 2 – If the stack is full, produces an error and exit.
- Step 3 – If the stack is not full, increments top to point next empty space.
- Step 4 – Adds data element to the stack location, where top is pointing.

- Step 5 – Returns success.

## *Stack Push Operation*

If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation
A simple algorithm for Push operation can be derived as follows –

```
begin procedure push: stack, data

  if stack is full
    return null
  endif

  top ← top + 1
  stack[top] ← data
```

end procedure

## *Pop Operation*

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps –

- Step 1 – Checks if the stack is empty.
- Step 2 – If the stack is empty, produces an error and exit.
- Step 3 – If the stack is not empty, accesses the data element at which top is pointing.
- Step 4 – Decreases the value of top by 1.
- Step 5 – Returns success.

Stack Pop Operation
Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack

  if stack is empty
    return null
  endif

  data ← stack[top]
  top ← top - 1
  return data

end procedure
```

## Pseudo code:

Push:

Top= top+1[new top value o] A[top]=x
If top >n-1
Printf stack is full Else
Top++ A[top]=x

pop:

if top==-
1 printf stack is empty
else
top--
printf A[top+1]

display:

forloop to displ
ay

# AIM Implementation of Queue Data Structure

## CODE

```
#include <stdio.h>
void insert();
void delete();
void display();
int queue[100];
int back=- 1;
int front=- 1;
int main()
{
   int a;
   char val='Y';
   while(val=='Y')
   {
      printf("1.Insert\n2.Delete\n3.Display\nEnter your option\n");
      scanf("%d",&a);
         switch(a)
         {
            case 1:
               insert();
               break;
            case 2:
               delete();
```

```
                break;
            case 3:
                display();
                break;
            default:
                printf("Try again using 1 or 2 or 3\n");
        }
        printf("Enter Y if you want to proceed else enter N\n");
        scanf("%s",&val);
    }
}
void insert()
{
    int value;
    if(back==99)
        printf("Queue Overflow\n");
    else
    {
        if(front==- 1||front==0)
        {
            front=0;
            printf("Enter the element to the queue\n");
            scanf("%d", &value);
            back=back+ 1;
            queue[back]=value;
        }
    }
}
void delete()
{
    if(front==- 1||front>back)
    {
        printf("Queue Underflow\n");
    }
    else
    {
        printf("Element deleted from queue is %d\n", queue[front]);
        front=front + 1;
    }
}
void display()
{
    int i;
    if(front==- 1)
        printf("Queue is empty\n");
    else
    {
        printf("Queue is");
        for(i=front;i<=back;i++)
        {
            printf(" %d ",queue[i]);
```

```
        }
        printf("\n");
    }
}
```

```
1.Insert
2.Delete
3.Display
Enter your option
1
Enter the element to the queue
4
Enter Y if you want to proceed else enter N
Y
1.Insert
2.Delete
3.Display
Enter your option
1
Enter the element to the queue
5
Enter Y if you want to proceed else enter N

Y
1.Insert
2.Delete
3.Display
Enter your option
1
Enter the element to the queue
7
Enter Y if you want to proceed else enter N
Y
1.Insert
2.Delete
3.Display
Enter your option
8
Try again using 1 or 2 or 3
Enter Y if you want to proceed else enter N
Y
```

```
Enter your option
1
Enter the element to the queue
8
Enter Y if you want to proceed else enter N
Y
1.Insert
2.Delete
3.Display
Enter your option
3
Queue is 4  5  7  8
Enter Y if you want to proceed else enter N
Y
1.Insert
2.Delete
3.Display
Enter your option
2
Element deleted from queue is 4
Enter Y if you want to proceed else enter N
Y
1.Insert
2.Delete
3.Display
Enter your option
2
Element deleted from queue is 5
Enter Y if you want to proceed else enter N
N


...Program finished with exit code 0
```

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- enqueue() − add (store) an item to the queue.
- dequeue() − remove (access) an item from the queue.
- Few more functions are required to make the above-mentioned queue operation efficient. These are −
- peek() − Gets the element at the front of the queue without removing it.
- isfull() − Checks if the queue is full.
- isempty() − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by front pointer and while enqueing (or storing) data in the queue we take help of rear pointer.

Let's first learn about supportive functions of a queue −

## *peek()*

This function helps to see the data at the front of the queue. The algorithm of peek() function is as follows −

Algorithm

```
begin procedure peek
  return queue[front]
end procedure
```

## *isfull()*

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

Algorithm

```
begin procedure isfull

  if rear equals to MAXSIZE
    return true
  else
    return false
  endif

end procedure
```

## *isempty()*

Algorithm

```
begin procedure isempty

  if front is less than MIN  OR front is greater than rear
    return true
  else
    return false
  endif
```

```
end procedure
```

## *Enqueue Operation*

Queues maintain two data pointers, front and rear. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- Step 1 − Check if the queue is full.
- Step 2 − If the queue is full, produce overflow error and exit.
- Step 3 − If the queue is not full, increment rear pointer to point the next empty space.
- Step 4 − Add data element to the queue location, where the rear is pointing.
- Step 5 − return success.

## *Insert Operation*

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation
procedure enqueue(data)

```
  if queue is full
    return overflow
  endif

  rear ← rear + 1
  queue[rear] ← data
  return true

end procedure
```

## *Dequeue Operation*

Accessing data from the queue is a process of two tasks − access the data where front is pointing and remove the data after access. The following steps are taken to perform dequeue operation −

- Step 1 − Check if the queue is empty.
- Step 2 − If the queue is empty, produce underflow error and exit.
- Step 3 − If the queue is not empty, access the data where front is pointing.
- Step 4 − Increment front pointer to point to the next available data element.
- Step 5 − Return success.

Remove Operation
Algorithm for dequeue operation
procedure dequeue

```
   if queue is empty
     return underflow
   end if

   data = queue[front]
   front ← front + 1
   return true

end procedure
```

# AIM Implementation of Linked List Data Structure

**CODE-**

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

struct node
{
   int value;
   struct node *next;
};

struct node *head, *tail = NULL;

void addNode()
{
   int value;
   printf("Write the number to be added.\n");
   scanf("%d",&value);
   struct node *newNode = (struct node*)malloc(sizeof(struct node));
   newNode->value = value;
   newNode->next = NULL;
   if(head == NULL)
   {
      head = newNode;
      tail = newNode;
   }
   else
   {
      tail->next = newNode;
      tail = newNode;
   }
   printf("Successully added.\n");
}

int countNodes()
{
   int count = 0;
   struct node *current = head;

   while(current != NULL)
   {
      count++;
      current = current->next;
```

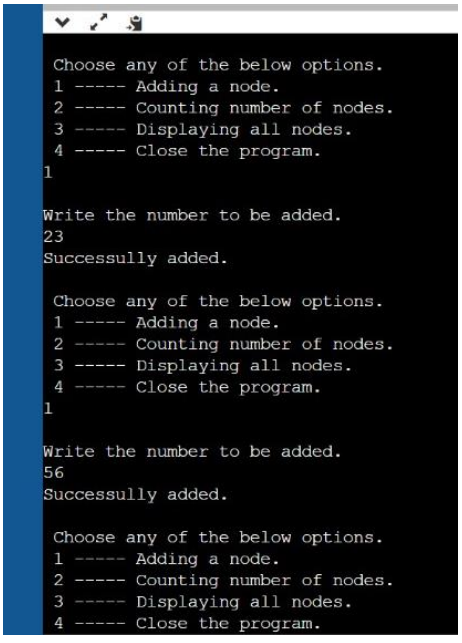
```
    }
    return count;
}

void display()
{
    struct node *current = head;
    if(head == NULL)
    {
        printf("Empty List\n");
        return;
    }
    printf("Nodes of linked list are: \n");
    while(current != NULL)
    {
        printf("%d ", current->value);
        current = current->next;
    }
    printf("\n");
}

int main()
{
    int opt;
     while(1)
    {
        printf("\n Choose any of the below options.");
        printf("\n 1 ----- Adding a node.");
        printf("\n 2 ----- Counting number of nodes.");
        printf("\n 3 ----- Displaying all nodes.");
        printf("\n 4 ----- Close the program.");
        printf("\n");

        scanf("%d",&opt);
        printf("\n");

            switch(opt)
            {
                case 1:
                addNode();
                break;

                case 2:
                countNodes();
                printf("NUmber of Node are: %d",countNodes());
                break;

                case 3:
```

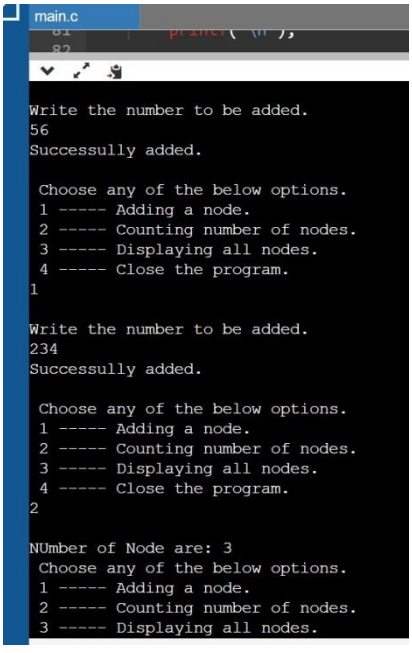
```
            display();
            break;

            case 4:
            exit(1);
            break;

            default:
            printf("Wrong Choice.\n");
        }
    }
}
```

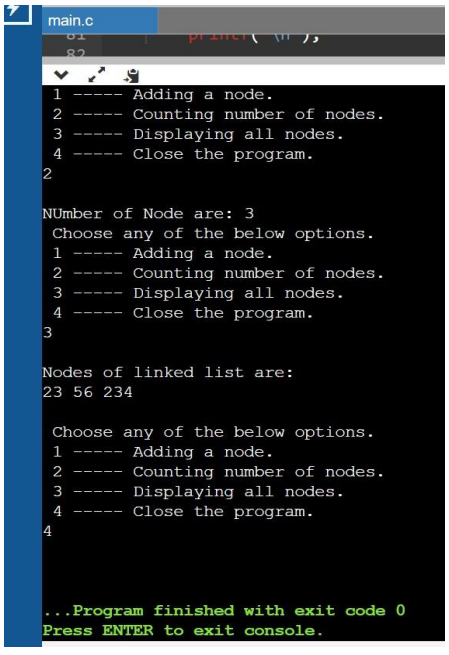
## ALGORITHM

Step 1 - Include all the header files which are used in the program.
Step 2 - Declare all the user defined functions.
Step 3 - Define a Node structure with two members data and next
Step 4 - Define a Node pointer 'head' and set it to NULL.
Step 5 - Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

OUTPUT-



# AIM Implementation of Linked List Data Structure

DOUBLY LINKED LIST

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

struct node {
   int data;
   int key;

   struct node *next;
   struct node *prev;
};

//this link always point to first Link
struct node *head = NULL;

//this link always point to last Link
struct node *last = NULL;

struct node *current = NULL;

//is list empty
bool isEmpty() {
   return head == NULL;
}

int length()  {
   int length = 0;
   struct node *current;
```

```
   for(current = head; current != NULL; current = current->next){
      length++;
   }

   return length;
}

//display the list in from first to last
void displayForward() {

   //start from the beginning
   struct node *ptr = head;

   //navigate till the end of the list
   printf("\n[ ");

   while(ptr != NULL) {
      printf("(%d,%d) ",ptr->key,ptr->data);
      ptr = ptr->next;
   }

   printf(" ]");
}

//display the list from last to first
void displayBackward() {

   //start from the last
   struct node *ptr = last;

   //navigate till the start of the list
   printf("\n[ ");
```

```
   while(ptr != NULL) {

      //print data
      printf("(%d,%d) ",ptr->key,ptr->data);

      //move to next item
      ptr = ptr ->prev;

   }

}

//insert link at the first location
void insertFirst(int key, int data) {

   //create a link
   struct node *link = (struct node*) malloc(sizeof(struct node));
   link->key = key;
   link->data = data;

   if(isEmpty()) {
      //make it the last link
      last = link;
   } else {
      //update first prev link
      head->prev = link;
   }

   //point it to old first link
   link->next = head;

   //point first to new first link
```

```
   head = link;
}

//insert link at the last location
void insertLast(int key, int data) {

   //create a link
   struct node *link = (struct node*) malloc(sizeof(struct node));
   link->key = key;
   link->data = data;

   if(isEmpty()) {
      //make it the last link
      last = link;
   } else {
      //make link a new last link
      last->next = link;

      //mark old last node as prev of new link
      link->prev = last;
   }

   //point last to new last node
   last = link;
}

//delete first item
struct node* deleteFirst() {

   //save reference to first link
   struct node *tempLink = head;

   //if only one link
```

```
   if(head->next == NULL){
      last = NULL;
   } else {
      head->next->prev = NULL;
   }

   head = head->next;
   //return the deleted link
   return tempLink;
}

//delete link at the last location

struct node* deleteLast() {
   //save reference to last link
   struct node *tempLink = last;

   //if only one link
   if(head->next == NULL) {
      head = NULL;
   } else {
      last->prev->next = NULL;
   }

   last = last->prev;

   //return the deleted link
   return tempLink;
}

//delete a link with given key

struct node* delete(int key) {
```

```
//start from the first link
struct node* current = head;
struct node* previous = NULL;

//if list is empty
if(head == NULL) {
  return NULL;
}

//navigate through list
while(current->key != key) {
  //if it is last node

  if(current->next == NULL) {
    return NULL;
  } else {
    //store reference to current link
    previous = current;

    //move to next link
    current = current->next;
  }
}

//found a match, update the link
if(current == head) {
  //change first to point to next link
  head = head->next;
} else {
  //bypass the current link
  current->prev->next = current->next;
}
```

```
   if(current == last) {
      //change last to point to prev link
      last = current->prev;
   } else {
      current->next->prev = current->prev;
   }

   return current;
}

bool insertAfter(int key, int newKey, int data) {
   //start from the first link
   struct node *current = head;

   //if list is empty
   if(head == NULL) {
      return false;
   }

   //navigate through list
   while(current->key != key) {

      //if it is last node
      if(current->next == NULL) {
         return false;
      } else {
         //move to next link
         current = current->next;
      }
   }

   //create a link
```

```
   struct node *newLink = (struct node*) malloc(sizeof(struct node));
   newLink->key = newKey;
   newLink->data = data;

   if(current == last) {
      newLink->next = NULL;
      last = newLink;
   } else {
      newLink->next = current->next;
      current->next->prev = newLink;
   }

   newLink->prev = current;
   current->next = newLink;
   return true;
}

void main() {
   insertFirst(1,10);
   insertFirst(2,20);
   insertFirst(3,30);
   insertFirst(4,1);
   insertFirst(5,40);
   insertFirst(6,56);

   printf("\nList (First to Last): ");
   displayForward();

   printf("\n");
   printf("\nList (Last to first): ");
   displayBackward();

   printf("\nList , after deleting first record: ");
```

```
   deleteFirst();
   displayForward();

   printf("\nList , after deleting last record: ");
   deleteLast();
   displayForward();

   printf("\nList , insert after key(4) : ");
   insertAfter(4,7, 13);
   displayForward();

   printf("\nList  , after delete key(4) : ");
   delete(4);
   displayForward();
}
```

**OUTPUT**

```
List (First to Last):
[ (6,56) (5,40) (4,1) (3,30) (2,20) (1,10) ]

List (Last to first):
[ (1,10) (2,20) (3,30) (4,1) (5,40) (6,56) ]
List , after deleting first record:
[ (5,40) (4,1) (3,30) (2,20) (1,10) ]
List , after deleting last record:
[ (5,40) (4,1) (3,30) (2,20) ]
List , insert after key(4) :
[ (5,40) (4,1) (7,13) (3,30) (2,20) ]
List , after delete key(4) :
[ (5,40) (4,13) (3,30) (2,20) ]
```

Algo written in comments too

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

- **Link** − Each link of a linked list can store a data called an element.
- **Next** − Each link of a linked list contains a link to the next link called Next.

- **Prev** − Each link of a linked list contains a link to the previous link called Prev.
- **LinkedList** − A Linked List contains the connection link to the first link called First and to the last link called Last.

## CIRCULAR LINKED LIST USING C

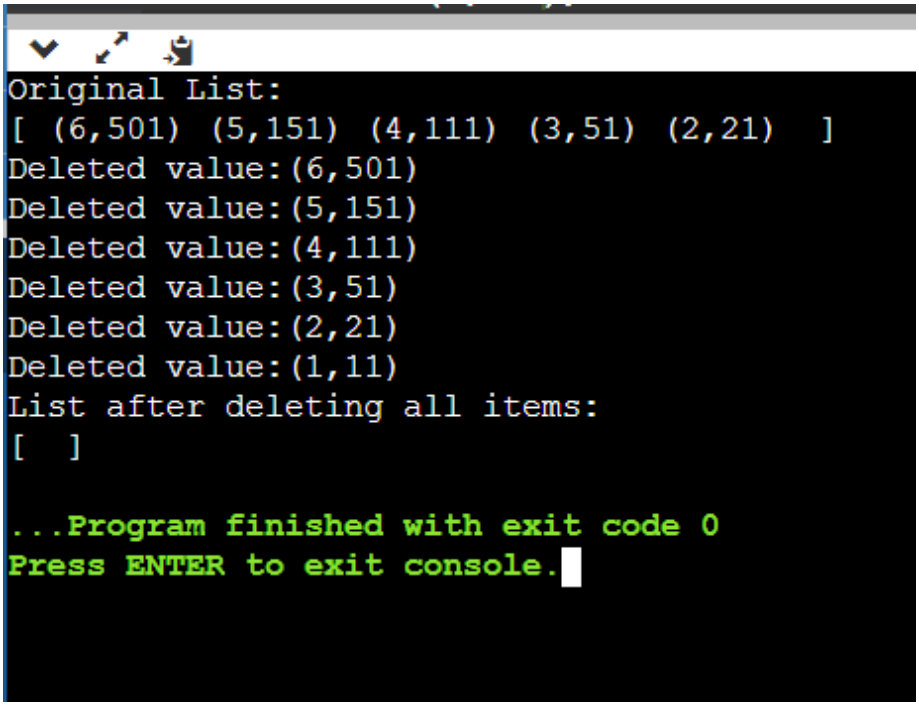
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

struct node {
   int data;
   int key;

   struct node *next;
};

struct node *head = NULL;
struct node *current = NULL;

bool isEmpty() {
   return head == NULL;
}

int length()  {
   int length = 0;

   if(head == NULL) {
      return 0;
   }

   current = head->next;
```

```
   while(current != head) {
      length++;
      current = current->next;
   }

   return length;
}

//insert link at the first location
void insertFirst(int key, int data) {

   //create a link
   struct node *link = (struct node*) malloc(sizeof(struct node));
   link->key = key;
   link->data = data;

   if (isEmpty()) {
      head = link;
      head->next = head;
   } else {
      //point it to old first node
      link->next = head;

      //point first to new first node
      head = link;
   }
}
```

```
//delete first item
struct node * deleteFirst() {

  //save reference to first link
  struct node *tempLink = head;

  if(head->next == head) {
    head = NULL;
    return tempLink;
  }

  //mark next to first link as first
  head = head->next;

  //return the deleted link
  return tempLink;
}

//display the list
void printList() {

  struct node *ptr = head;
  printf("\n[ ");

  //start from the beginning
  if(head != NULL) {
```

```
    while(ptr->next != ptr) {
      printf("(%d,%d) ",ptr->key,ptr->data);
      ptr = ptr->next;
    }
  }

  printf(" ]");
}

int main() {
  insertFirst(1,11);
  insertFirst(2,21);
  insertFirst(3,51);
  insertFirst(4,111);
  insertFirst(5,151);
  insertFirst(6,501);

  printf("Original List: ");

  //print list
  printList();

  while(!isEmpty()) {
    struct node *temp = deleteFirst();
    printf("\nDeleted  value:");
    printf("(%d,%d) ",temp->key,temp->data);
  }
```

```
  printf("\nList after deleting all items: ");
  printList();
}
```



# DOUBLY CIRCULAR 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 list\n 5.delete node at position\n 6.updatenodevalue\n7.search element
\n8.displaylist from beginning\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
length");

      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);
      }
   }
}
```

# Output

```
linked list
1.insert at beginning
2.insert at end
3.insert at position
4.sort linked list
5.delete node at position
6.updatenodevalue
7.search element
8.displaylist from beginning
9.display list from end
10.exit
 enter your choice:8

list is empty no elemnts to print
 enter your choice:5

 enter the position which u wanted to delete:2

 empty linked list you cant delete
 enter your choice:6

 enter the value old value:6

 enter the value new value:8

 list is empty no elemnts for updation
 enter your choice:7

enter the value to be searched:57
```

```
list is empty no elemnets in list to search
 enter your choice:1

enter the value to be inserted at first:11

initially it is empty linked list later insertion is done
 enter your choice:3

 enter the value that you would like to insert:5

 enter the position where you have to enter:5

 node cant be inserted as position is exceeding the linkedlist length
 enter your choice:1

enter the value to be inserted at first:56

 the value is inserted at begining
 enter your choice:1

enter the value to be inserted at first:89

 the value is inserted at begining
 enter your choice:2

enter the value that has to be inserted at last:89

enter your choice:2

enter the value that has to be inserted at last:45

enter your choice:
6 number of nodes are there
```

```
89
56
11
89
45
89
enter your choice:4

11
89
89
45
56
11
enter your choice:10
```

**Algorithm(circular doubly)**

- Step 1: IF PTR = NULL.
- Step 2: SET NEW_NODE = PTR.
- Step 3: SET PTR = PTR -> NEXT.
- Step 4: SET NEW_NODE -> DATA = VAL.
- Step 5: SET TEMP = HEAD.
- Step 6: Repeat Step 7 while TEMP -> NEXT != HEAD.
- Step 7: SET TEMP = TEMP -> NEXT.
- Step 8: SET TEMP -> NEXT = NEW_NODE.

**Insertion in doubly linked list**

To insert a node at the end of the list, follow these steps:
1. Create a node, say T.
2. Make T -> next = last -> next;
3. last -> next = T.
4. last = T.

# DSA_lab assignment

Mayank Gupta
20BCE1538

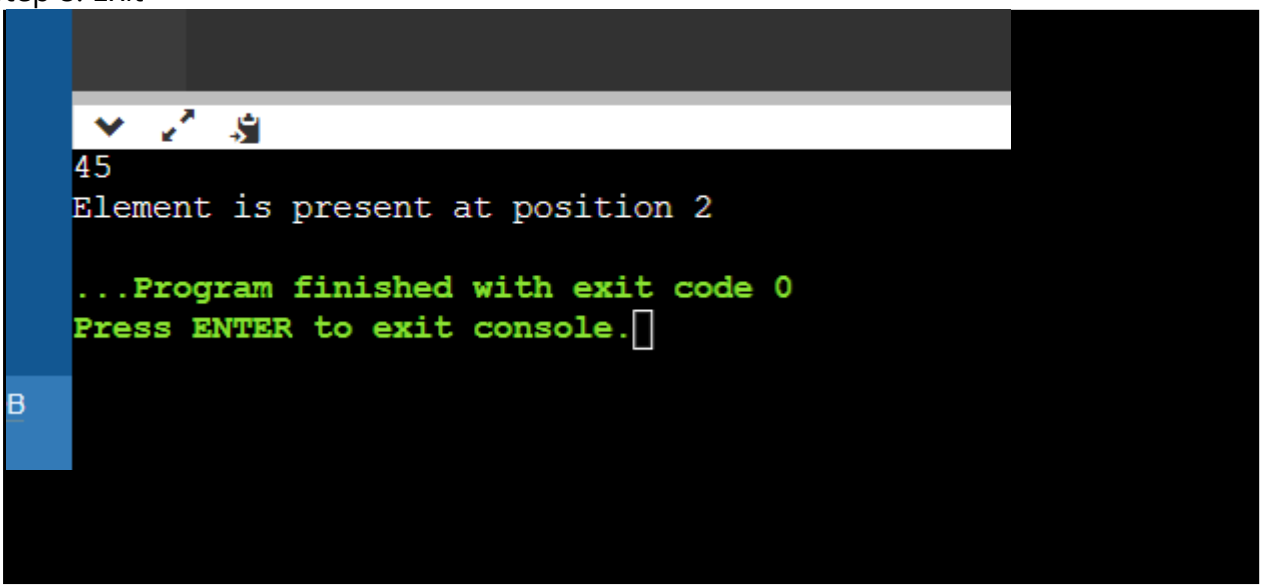
# AIM Implementation of Searching Algorithms

## A) Linear Search:

```
#include <stdio.h>
//linear search function
int search(int arr[], int n, int x) {
   int i;
   for (i = 0; i < n; i++)
   if (arr[i] == x) return i;
   return -1;
}
int main(void) {
   int arr[] = {15, 45, 8, 65, 52, 45, 65, 69, 12, 10};
   int x; scanf("%d", &x); int n = sizeof(arr) / sizeof(arr[0]);
   // Function call
   int result = search(arr, n, x);
   if (result == -1) printf("Element is not present in array");
   else printf("Element is present at position %d", result + 1);
   return 0;
}
```

**ALGORITHM**

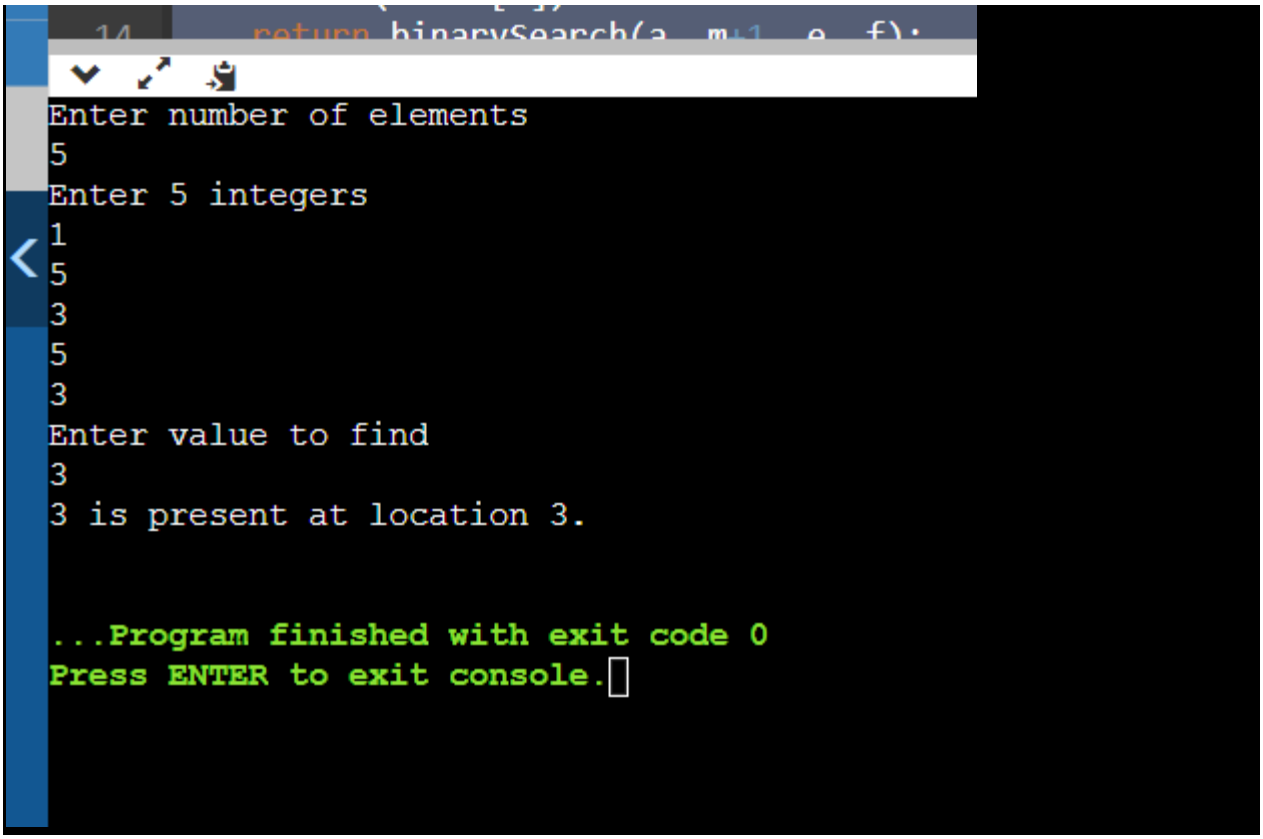
Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit



## B) Binary Search

#include <stdio.h>

```
//linear search function
int binarySearch(int a[], int s, int e, int f) {
  int m;

  if (s > e) // Not found
     return -1;

  m = (s + e)/2;

  if (a[m] == f)  // element found
    return m;
  else if (f > a[m])
    return binarySearch(a, m+1, e, f);
  else
    return binarySearch(a, s, m-1, f);
}
int main()
{
  int c, first, last, n, search, array[100], index;

  printf("Enter number of elements\n");
  scanf("%d", &n);

  printf("Enter %d integers\n", n);

  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);

  printf("Enter value to find\n");
  scanf("%d", &search);

  first = 0;
  last = n - 1;

  index = binarySearch(array, first, last, search);

  if (index == -1)
    printf("Not found! %d isn't present in the list.\n", search);
  else
    printf("%d is present at location %d.\n", search, index + 1);

  return 0;
}
```




## ALGORITHM

1. Step 1: set beg = lower_bound, end = upper_bound, pos = - 1
2. Step 2: repeat steps 3 and 4 while beg <=end
3. Step 3: set mid = (beg + end)/2
4. Step 4: if a[mid] = val
5. set pos = mid
6. print pos
7. go to step 6
8. else if a[mid] > val
9. set end = mid - 1
10. else
11. set beg = mid + 1
12. [end of if]
13. [end of loop]
14. Step 5: if pos = -1
15. print "value is not present in the array"
16. [end of if]
17. Step 6: exit

# AIM Insertion Sort, Selection Sort and Bubble Sort

Mayank Gupta 20BCE1538

```
#include <iostream>

using namespace std;
void swap(int *arr, int n,int &x,int &y){
   int temp=y;
   y=x;
   x=temp;
   cout<<x<<" is swapped with "<<y<<endl;
   for(int i=0;i<n;i++){
      cout<<arr[i]<<" ";
   }
   cout<<endl;
}


void insertionSort(int *arr, int n, int s){
   int key,j;
   for(int i=1;i<n;i++){
      key = arr[i];
      j=i;
      for(int i=0;i<n;i++){
      cout<<arr[i]<<" ";
      }
      cout<<endl;
      cout<<"-> the key taken is "<<key<<endl;
      while(j>0 && arr[j-1]>key){
         cout<<"Right shift "<<arr[j-1]<<" by 1 posn"<<endl;
         arr[j]=arr[j-1];
         j--;
      }
      cout<<"place the key at the posn of "<<arr[j]<<endl;
      arr[j]=key;
   }
}
void bubbleSort(int *arr, int n){
   for(int i=0;i<n;i++){
```

```
      for(int j=0;j<n-i-1;j++){
         if(arr[j]>arr[j+1]){
            swap(arr,n,arr[j],arr[j+1]);
         }
      }
   }
}

void display(int *arr, int n){
   cout<<"Final Correct Output"<<": ";
   for(int i=0;i<n;i++){
      cout<<arr[i]<<" ";
   }
   cout<<endl;
}
//11 10 7 9 '5' 6 4 8
void selectionSort(int arr[],int n){
   int int_min;
   for(int i=0;i<n-1;i++){
      int index=0;
      int_min=999;
      for(int j=i;j<n;j++){
         if(arr[j]<int_min){
            int_min=arr[j];
            index=j;
         }
      }
   swap(arr,n,arr[index],arr[i]);
   }
}
int main()
{
   int n;
   cin>>n;
   int arr[n];
   for(int i=0;i<n;i++){
      cin>>arr[i];
   }
   int arr1[n];
   for(int i=0;i<n;i++){
      arr1[i]=arr[i];
   }
```

```
   int s=0; //no. of swaps
   cout<<"Bubble sort"<<endl;
   bubbleSort(arr1,n);
   display(arr1,n);
   for(int i=0;i<n;i++){
      arr1[i]=arr[i];
   }
   cout<<"Insertion sort"<<endl;
   insertionSort(arr1,n,s);
   display(arr1,n);
   for(int i=0;i<n;i++){
      arr1[i]=arr[i];
   }
   cout<<"Selection sort"<<endl;
   selectionSort(arr1,n);
   display(arr1,n);
   //cout<<"no. of swaps" << s;
   return 0;
}
```

## ***Output***

- **Bubble Sort Output**

```
Bubble sort
10 is swapped with 11
10 11 7 9 5 6 4 8
7 is swapped with 11
10 7 11 9 5 6 4 8
9 is swapped with 11
10 7 9 11 5 6 4 8
5 is swapped with 11
10 7 9 5 11 6 4 8
6 is swapped with 11
10 7 9 5 6 11 4 8
4 is swapped with 11
10 7 9 5 6 4 11 8
8 is swapped with 11
10 7 9 5 6 4 8 11
7 is swapped with 10
7 10 9 5 6 4 8 11
9 is swapped with 10
7 9 10 5 6 4 8 11
5 is swapped with 10
7 9 5 10 6 4 8 11
6 is swapped with 10
7 9 5 6 10 4 8 11
4 is swapped with 10
7 9 5 6 4 10 8 11
8 is swapped with 10
7 9 5 6 4 8 10 11
5 is swapped with 9
7 5 9 6 4 8 10 11
6 is swapped with 9
7 5 6 9 4 8 10 11
4 is swapped with 9
7 5 6 4 9 8 10 11
8 is swapped with 9
7 5 6 4 8 9 10 11
5 is swapped with 7
5 7 6 4 8 9 10 11
6 is swapped with 7
5 6 7 4 8 9 10 11
4 is swapped with 7
5 6 4 7 8 9 10 11
4 is swapped with 6
5 4 6 7 8 9 10 11
4 is swapped with 5
4 5 6 7 8 9 10 11
Final Correct Output: 4 5 6 7 8 9 10 11
Insertion sort
```

- **Insertion Sort Output**

```
Insertion sort
11 10 7 9 5 6 4 8
-> the key taken is 10
Right shift 11 by 1 posn
place the key at the posn of 11
10 11 7 9 5 6 4 8
-> the key taken is 7
Right shift 11 by 1 posn
Right shift 10 by 1 posn
place the key at the posn of 10
7 10 11 9 5 6 4 8
-> the key taken is 9
Right shift 11 by 1 posn
Right shift 10 by 1 posn
place the key at the posn of 10
7 9 10 11 5 6 4 8
-> the key taken is 5
Right shift 11 by 1 posn
Right shift 10 by 1 posn
Right shift 9 by 1 posn
Right shift 7 by 1 posn
place the key at the posn of 7
5 7 9 10 11 6 4 8
-> the key taken is 6
Right shift 11 by 1 posn
Right shift 10 by 1 posn
Right shift 9 by 1 posn
Right shift 7 by 1 posn
place the key at the posn of 7
5 6 7 9 10 11 4 8
-> the key taken is 4
Right shift 11 by 1 posn
Right shift 10 by 1 posn
Right shift 9 by 1 posn
Right shift 7 by 1 posn
Right shift 6 by 1 posn
Right shift 5 by 1 posn
place the key at the posn of 5
4 5 6 7 9 10 11 8
-> the key taken is 8
Right shift 11 by 1 posn
Right shift 10 by 1 posn
Right shift 9 by 1 posn
place the key at the posn of 9
Final Correct Output: 4 5 6 7 8 9 10 11
```

- **Selection Sort**

```
Selection sort
11 is swapped with 4
4 10 7 9 5 6 11 8
10 is swapped with 5
4 5 7 9 10 6 11 8
7 is swapped with 6
4 5 6 9 10 7 11 8
9 is swapped with 7
4 5 6 7 10 9 11 8
10 is swapped with 8
4 5 6 7 8 9 11 10
9 is swapped with 9
4 5 6 7 8 9 11 10
11 is swapped with 10
4 5 6 7 8 9 10 11
Final Correct Output: 4 5 6 7 8 9 10 11
```

**ALGO FOR INSERTION SORT**

Algorithm

To sort an array of size n in ascending order:
1: Iterate from arr[1] to arr[n] over the array.
2: Compare the current element (key) to its predecessor.
3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

**ALGO FOR BUBBLE SORT**

We assume list is an array of n elements. We further assume that swap function swaps the values of the given array elements.

```
begin BubbleSort(list)

  for all elements of list
    if list[i] > list[i+1]
      swap(list[i], list[i+1])
    end if
  end for

  return list

end BubbleSort
```

**ALGO FOR SELECTION SORT**

Step 1 – Set MIN to location 0
Step 2 – Search the minimum element in the list
Step 3 – Swap with value at location MIN
Step 4 – Increment MIN to point to next element
Step 5 – Repeat until list is sorted

Mayank Gupta 20BCE1538_DSA

# AIM Randomized Quick and Merge Sort

1. A set of children were made to stand in a line. Each child has been asked to pick a card which has a unique number (1 to n). The children have to rearrange themselves in ascending order according to the number they are holding. The arrangement has to begin by considering any of the children as a pivot element. Moreover the kids are supposed to use divide and conquer strategy for this arrangement process.

Write a program to help the kids arrange themselves.

Note: The pivot element is the element of a matrix, or an array, which is selected first by an algorithm.

**CODE;**

```
*****************************************************************************/


#include <iostream>


using namespace std;
void swap(int &x,int &y){
   int temp=y;
   y=x;
   x=temp;
   cout<<x<<" is swapped with "<<y<<endl;


}
int partition (int arr[], int low, int high)
{
   cout<<"students in focus : ";
```

```
for(int i=low;i<=high;i++){
    cout<<arr[i]<<" ";
}
cout<<endl;
int pivot = arr[high]; // pivot
cout<<"The Pivot element is (in this case last element) "<<pivot<<endl;
int i = (low - 1); // Index of smaller element and indicates the right position of pivot found so far


for (int j = low; j <= high - 1; j++)
{
    // If current element is smaller than the pivot
    if (arr[j] < pivot)
    {
        i++; // increment index of smaller element
        swap(arr[i], arr[j]);
        cout<<"Arrangement after swapping : ";
        for(int i=low;i<=high;i++){
            cout<<arr[i]<<" ";
        }
        cout<<endl;


    }
}
swap(arr[i + 1], arr[high]);
cout<<"Arrangement after swapping : ";
```

```
   for(int i=low;i<=high;i++){
      cout<<arr[i]<<" ";
   }
   cout<<endl;
   return (i + 1);
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
   if (low < high)
   {
      /* pi is partitioning index, arr[p] is now
      at right place */
      int pi = partition(arr, low, high);

      // Separately sort elements before
      // partition and after partition
      cout<<"Arrangement after left division"<<endl;
      quickSort(arr, low, pi - 1);
      cout<<"Arrangement after right division"<<endl;
      quickSort(arr, pi + 1, high);
```

```
    }
}


int main()
{
    int n;
    cin>>n;
    int arr[n];
    for(int i=0;i<n;i++){
        int x;
        cin>>x;
        if(x<=n){
            arr[i]=x;
        }
        else{
            cout<<"students till n roll no. to be entered"<<endl;
        }
    }
    cout<<"*****Quick Sort*****"<<endl;
    quickSort(arr,0,n-1);
    cout<<"Final arrangement of Students is : ";
    for(int i=0;i<n;i++){
        cout<<arr[i]<<" ";
    }
```

```
  return 0;

}
```

```
*****Quick Sort*****
students in focus : 9 8 7 6 5 4 3 2 1
The Pivot element is (in this case last element) 1
1 is swapped with 9
Arrangement after swapping : 1 8 7 6 5 4 3 2 9
Arrangement after left division
Arrangement after right division
students in focus : 8 7 6 5 4 3 2 9
The Pivot element is (in this case last element) 9
8 is swapped with 8
Arrangement after swapping : 8 7 6 5 4 3 2 9
7 is swapped with 7
Arrangement after swapping : 8 7 6 5 4 3 2 9
6 is swapped with 6
Arrangement after swapping : 8 7 6 5 4 3 2 9
5 is swapped with 5
Arrangement after swapping : 8 7 6 5 4 3 2 9
4 is swapped with 4
Arrangement after swapping : 8 7 6 5 4 3 2 9
3 is swapped with 3
Arrangement after swapping : 8 7 6 5 4 3 2 9
2 is swapped with 2
Arrangement after swapping : 8 7 6 5 4 3 2 9
9 is swapped with 9
Arrangement after swapping : 8 7 6 5 4 3 2 9
Arrangement after left division
students in focus : 8 7 6 5 4 3 2
The Pivot element is (in this case last element) 2
2 is swapped with 8
Arrangement after swapping : 2 7 6 5 4 3 8
Arrangement after left division
Arrangement after right division
students in focus : 7 6 5 4 3 8
The Pivot element is (in this case last element) 8
7 is swapped with 7
```

```
7 is swapped with 7
Arrangement after swapping : 7 6 5 4 3 8
6 is swapped with 6
Arrangement after swapping : 7 6 5 4 3 8
5 is swapped with 5
Arrangement after swapping : 7 6 5 4 3 8
4 is swapped with 4
Arrangement after swapping : 7 6 5 4 3 8
3 is swapped with 3
Arrangement after swapping : 7 6 5 4 3 8
8 is swapped with 8
Arrangement after swapping : 7 6 5 4 3 8
Arrangement after left division
students in focus : 7 6 5 4 3
The Pivot element is (in this case last element) 3
3 is swapped with 7
Arrangement after swapping : 3 6 5 4 7
Arrangement after left division
Arrangement after right division
students in focus : 6 5 4 7
The Pivot element is (in this case last element) 7
6 is swapped with 6
Arrangement after swapping : 6 5 4 7
5 is swapped with 5
Arrangement after swapping : 6 5 4 7
4 is swapped with 4
Arrangement after swapping : 6 5 4 7
7 is swapped with 7
Arrangement after swapping : 6 5 4 7
Arrangement after left division
students in focus : 6 5 4
The Pivot element is (in this case last element) 4
4 is swapped with 6
Arrangement after swapping : 4 5 6
Arrangement after left division
Arrangement after right division
```

```
Arrangement after swapping : 4 5 6
Arrangement after left division
Arrangement after right division
students in focus : 5 6
The Pivot element is (in this case last element) 6
5 is swapped with 5
Arrangement after swapping : 5 6
6 is swapped with 6
Arrangement after swapping : 5 6
Arrangement after left division
Arrangement after right division
Arrangement after right division
Arrangement after right division
Arrangement after right division
Final arrangement of Students is : 1 2 3 4 5 6 7 8 9
```

**ALGO**

```
/* low --> Starting index, high --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1); // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
```

2.Microsoft organized an App fest for students wherein 10 students participated and posted applications. Students were given prizes based on the number of applications posted. Arrange the students in the increasing order of applications using merge sort.

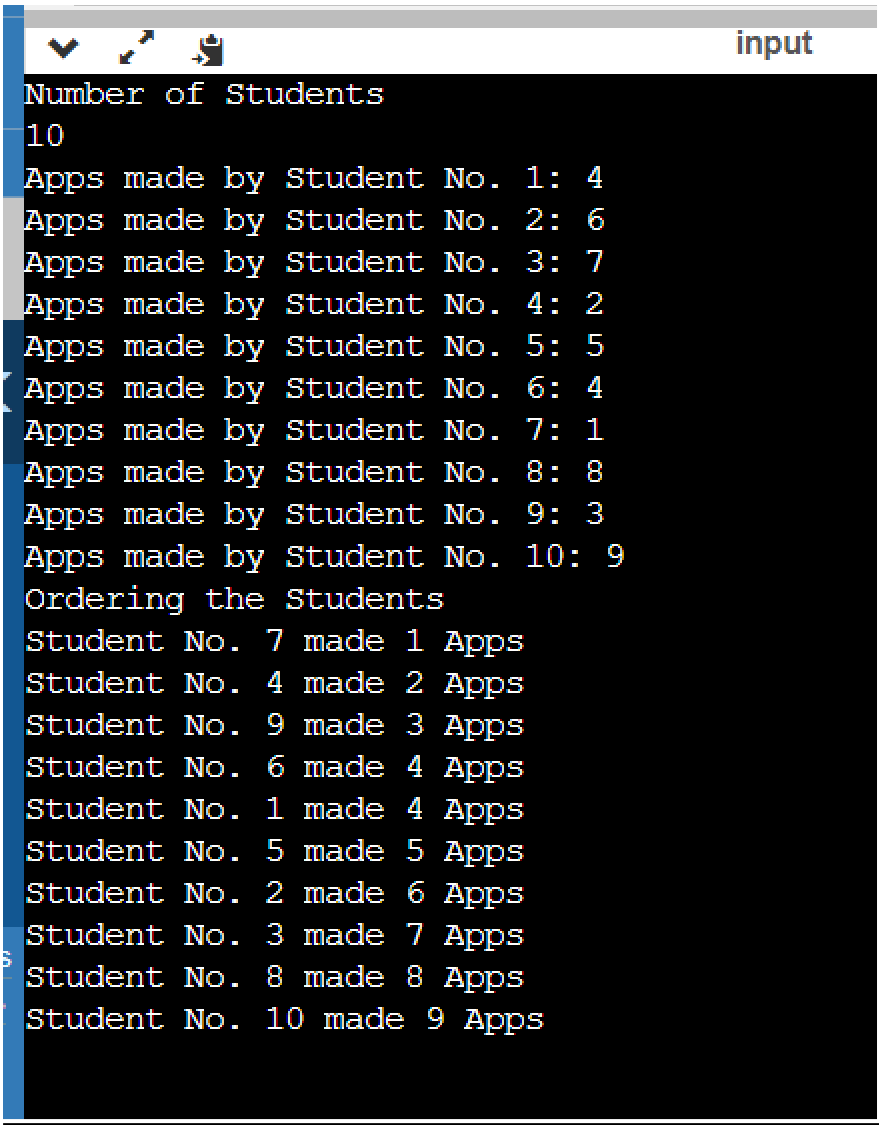
| Student Number | No.of App's posted |
|---|---|
| 1 | 4 |
| 2 | 6 |
| 3 | 7 |
| 4 | 2 |
| 5 | 5 |
| 6 | 4 |
| 7 | 1 |
| 8 | 8 |
| 9 | 3 |
| 10 | 9 |

**Code;**

```
#include<iostream>

using namespace std;

struct stud{

int n,ap;

};

void copy(struct stud &a,struct stud &b){

a.ap=b.ap;

a.n=b.n;

}

void merge(struct stud ar[],int l,int r,int mid){

int i=l,j=mid+1,k=l;
```

```
struct stud b[100];
while(i<=mid && j<=r){
if(ar[i].ap<ar[j].ap){
copy(b[k],ar[i]);
i++;k++;
}else{
copy(b[k],ar[j]);
j++;k++;
}
}if(i>mid){
while(j<=r){
copy(b[k],ar[j]);
j++;k++;
}
}else{
while(i<=mid){
copy(b[k],ar[i]);
i++;k++;
}
}
for(k=l;k<=r;++k) copy(ar[k],b[k]);
}
void mergesort(struct stud ar[],int l, int r){
if(l<r){
int mid=(l+r)/2;
```

```
mergesort(ar,l,mid);

mergesort(ar,mid+1,r);

merge(ar,l,r,mid);

}

}
int main(){

int n;cout<<"Number of Students\n";

cin>>n;
struct stud apps[n];

for(int i=0;i<n;++i){

cout<<"Apps made by Student No. "<<i+1<<": ";

cin>>apps[i].ap;

apps[i].n=i+1;

}
mergesort(apps,0,n-1);

cout<<"Ordering the Students\n";

for(int i=0;i<n;++i) cout<<"Student No. "<<apps[i].n<<" made "<<apps[i].ap<<" Apps\n";

cout<<endl;

}
```




**Algo**

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = l+ (r-l)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

# AIM Binary Tree Traversals

CODE

```
// Tree traversal in C

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

struct node {
  int item;
  struct node* left;
  struct node* right;
};

// Inorder traversal
void inorderTraversal(struct node* root) {
  if (root == NULL) return;
  inorderTraversal(root->left);
  printf("%d ->", root->item);
  inorderTraversal(root->right);
}

// preorderTraversal traversal
void preorderTraversal(struct node* root) {
  if (root == NULL) return;
  printf("%d ->", root->item);
  preorderTraversal(root->left);
  preorderTraversal(root->right);
}

// postorderTraversal traversal
void postorderTraversal(struct node* root) {
  if (root == NULL) return;
  postorderTraversal(root->left);
  postorderTraversal(root->right);
  printf("%d ->", root->item);
}

// Create a new Node
struct node* createNode(value) {
```

```
  struct node* newNode = malloc(sizeof(struct node));
  newNode->item = value;
  newNode->left = NULL;
  newNode->right = NULL;

  return newNode;
}

// Insert on the left of the node
struct node* insertLeft(struct node* root, int value) {
  root->left = createNode(value);
  return root->left;
}

// Insert on the right of the node
struct node* insertRight(struct node* root, int value) {
  root->right = createNode(value);
  return root->right;
}

int main() {
  struct node* root = createNode(1);
  insertLeft(root, 12);
  insertRight(root, 9);

  insertLeft(root->left, 5);
  insertRight(root->left, 6);

  printf("Inorder traversal \n");
  inorderTraversal(root);

  printf("\nPreorder traversal \n");
  preorderTraversal(root);

  printf("\nPostorder traversal \n");
  postorderTraversal(root);
}
```

**ALGORITHM**

**Inorder traversal**

First, visit all the nodes in the left subtree

Then the root node

Visit all the nodes in the right subtree

**Preorder traversal**

Visit root node

Visit all the nodes in the left subtree

Visit all the nodes in the right subtree

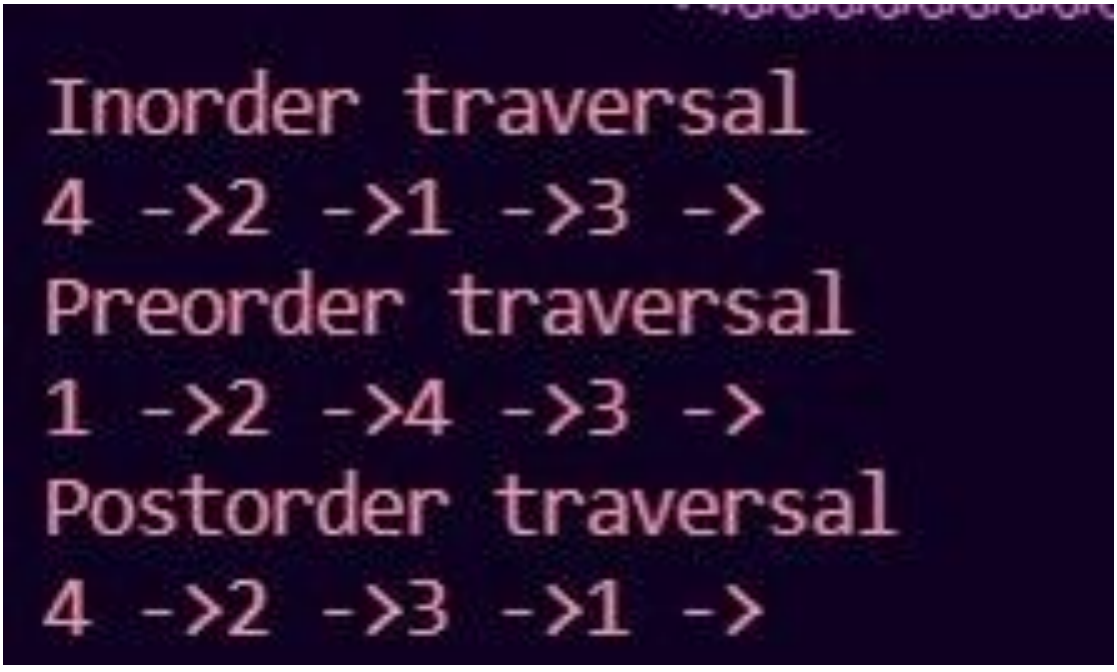
**Postorder traversal**

Visit all the nodes in the left subtree

Visit all the nodes in the right subtree

Visit the root node

## Output



# AIM Binary Search Tree

**QUEST** Assume that there are twenty students enrolled for a spoken English class in British council. The students are roll numbered from 1 to 20. As per the registry in the entrance of the British Council, every day the entry of the students for the class is in random order. Implement a program to construct a BST (using the student Roll No) from the order in which the students are entering in to council for the class. If any student leaves the council before the end of the class, remove them from the BST. Display BST on every operation performed.

**CODE**

```
#include <bits/stdc++.h>
using namespace std;
class Node{
public:

   int data;

   Node *left;

   Node *right;



 public:

   Node(int x){

   data = x;

     left = NULL;

     right = NULL;

   }

};
```

```
Node *insert_BST(Node *tree, int val)

{

 if(tree == NULL){

   cout<<val <<" is inserted into BST successfully"<<endl;

   return (new Node(val));

 }

 if(tree->data >= val)

   tree->left = insert_BST(tree->left, val);

 else

   tree->right = insert_BST(tree->right, val);

 return tree;

}

bool searchInBST(Node *root , int k) {
            // Write your code here
   if(root==NULL){
      return false;
   }
   if(root->data==k){
      return true;
   }
   else if(root->data>k){
            return searchInBST(root->left ,k);
   }
   else if(root->data<k){
      return searchInBST(root->right ,k);
   }

}
Node* delete_BST(Node *tree, int val)
```

```
{
 if(tree == NULL){
   cout<<"value is not present in BST"<<endl;
   return tree;
 }
 if(tree->data > val)
   tree->left = delete_BST(tree->left, val);
 else if(tree->data < val)
   tree->right = delete_BST(tree->right, val);
 else{
  if(tree->left == NULL){
    Node *temp = tree->right;
    free(tree);
    tree= temp;
  }

  else if(tree->right == NULL){
    Node *temp = tree->left;
    tree = temp;
    free(temp);
  }
```

```
   else{
     Node *temp = tree->left;
     while(temp->right->right!=NULL){
       temp = temp->right;
     }
     tree->data = temp->right->data;
     Node *temp2 = temp->right->left;
     free(temp->right);
     temp->right = temp2;
   }
  }
  return tree;
}
void inorder(Node *tree)
{
   if (tree != NULL)
   {
      inorder(tree->left);
      cout<<tree->data<<" ";
      inorder(tree->right);
   }
```

```
}
void printLevelATNewLine(Node *root) {
   if (root == NULL) {
      return;
   }
   queue<Node *> q;
   q.push(root);
   q.push(NULL);
   while (!q.empty()) {
      Node *first = q.front();
      q.pop();
      if (first == NULL) {
         if (q.empty()) {
            break;
         }
         cout << endl;
         q.push(NULL);
         continue;
      }
      cout << first->data << " ";
      if (first->left != NULL) {
         q.push(first->left);
      }
      if (first->right != NULL) {
         q.push(first->right);
      }
   }
}

int main(){
   cout<<"BRITISH COUNCIL CLASS"<<endl;
   // cout<<"Root of the BST i.e. the first child ";
   // int x;
   // cin>>x;
   // cout<<x<<endl;
   Node *root = NULL;
   //takeInput(root);

   int z;
   while(z!=6){
   cout<<"*****************"<<endl;
```

```
cout<<"Enter 1: To Insert student"<<endl;
cout<<"Enter 2: To Search for any student"<<endl;
cout<<"Enter 3: if any student exits"<<endl;
cout<<"Enter 4: To print the tree in order"<<endl;
cout<<"Enter 5: To print level wise"<<endl;
cout<<"Enter 6: To Exit"<<endl;
cout<<"******************"<<endl;

cin >> z;
switch(z){
   case 1:
   cout<<"Enter the student roll no. as they enter the class "<<endl;
   int id;
   cin>>id;
   root=insert_BST(root, id);
   inorder(root);
   break;
   case 2:
   int k;
   cout<<"Enter the student Roll No. for search "<<endl;
   cin>>k;
   if(searchInBST(root,k)==false){
      cout<<"Student is NOT PRESENT in the class"<<endl;
   }
   else{
      cout<<"Student is PRESENT in the class"<<endl;
   }
   break;
   case 3:
   int m;
   cout<<"Enter the student's Roll No. who exited "<<endl;
   cin>>m;
   delete_BST(root, m);
   //printLevelATNewLine(root);
   break;
   case 4:
   inorder(root);
   break;
   case 5:
   printLevelATNewLine(root);
   break;
   case 6:
```

```
      z=6;
      break;
    }

    }
    // cout << ((searchInBST(root, k)) ? "true" : "false");
    cout<<"*******EXIT********"<<endl;
    delete root;

}
```

**PSEUDO CODE**

**INSERTION**

Step1 If tree is empty (no root), create a node holding key k as root; done.

Step2. If the val if less than the root we will traverse the left side.

Step3. If the val if greater than the root we will traverse the right side.
Step4. If k< CurrNode.key ... /* key must go in left subtree */
If CurrNode.left == NULL, create a node holding k as left child of CurrNode; done.
else set CurrNode = CurrNode.left, and go to 3.

Step5. else ... /* key must go in right subtree */
If CurrNode.right == NULL, create a node holding k as left child of CurrNode; done.
else set CurrNode = CurrNode.right, and go to 3.

**SEARCH**

Step1 Base CASE-If root is empty return empty;

Step2 If root is equal to the given value return true;

Step3 if root is greater than the val traverse left

Step4 if root is less than the val traverse right

**DELETE**

Step1 Base CASE-If tree==NULL,print “not present” and return tree;

Step2 If root is equal to the given value, delete the root node and connect everything;

Step3 if root is greater than the val traverse left

Step4 if root is less than the val traverse right

**INORDER**

Step1 check if tree not equal to null;

Step2 start printing from the left most node then the root then the right;

PRINT LEVEL WISE

Step1 if root is equal to NULL return;

Step2 define a queue “q”;

Step3 push the first element;

Step4 start the loop till queue is empty

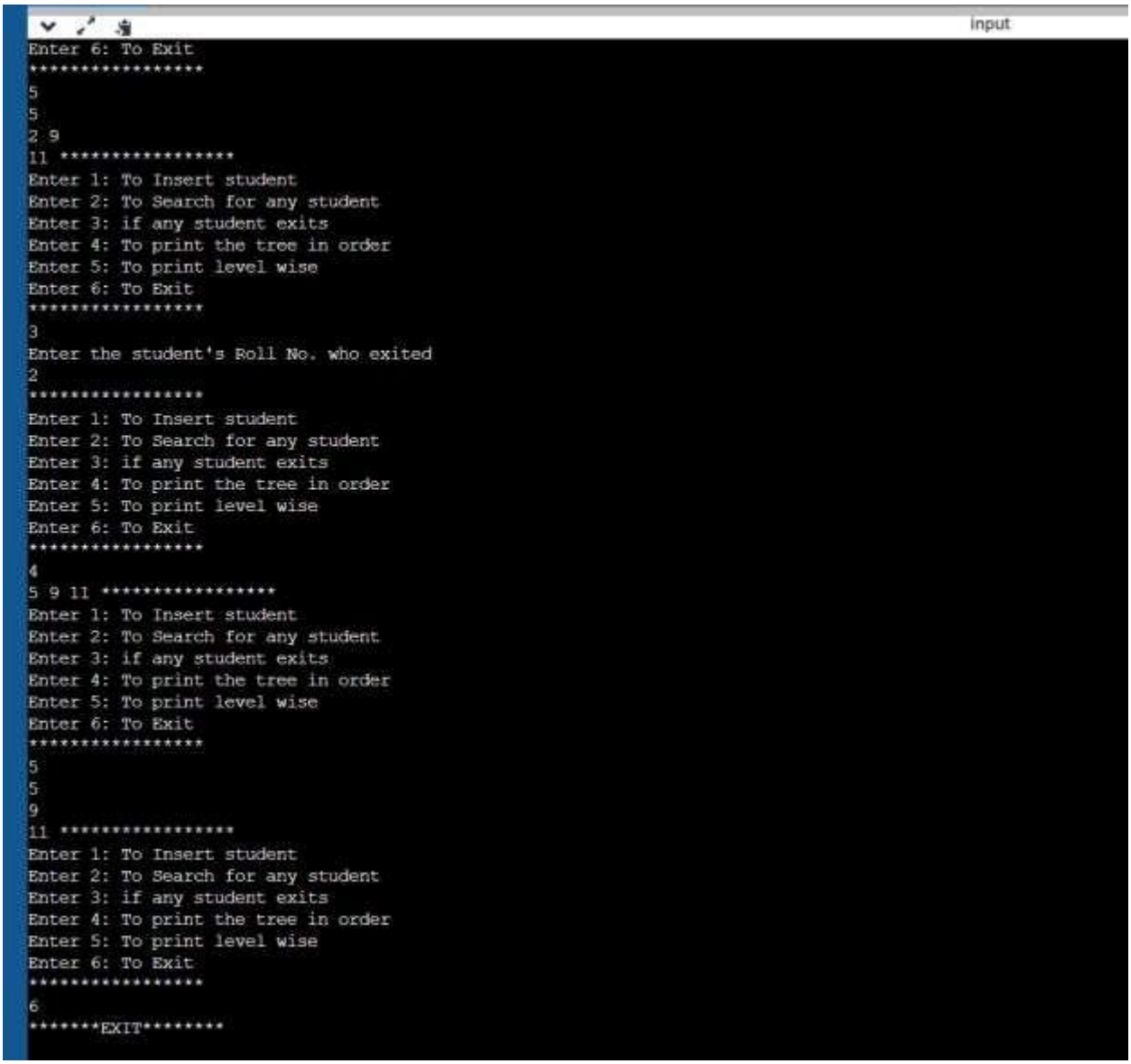
Step5 store the front node in “first” and pop

Step 6 END

## OUTPUT

```
BRITISH COUNCIL CLASS
*****************
Enter 1: To Insert student
Enter 2: To Search for any student
Enter 3: if any student exits
Enter 4: To print the tree in order
Enter 5: To print level wise
Enter 6: To Exit
*****************
1
Enter the student roll no. as they enter the class
5
5 is inserted into BST successfully
5 *****************
Enter 1: To Insert student
Enter 2: To Search for any student
Enter 3: if any student exits
Enter 4: To print the tree in order
Enter 5: To print level wise
Enter 6: To Exit
*****************
1
Enter the student roll no. as they enter the class
9
9 is inserted into BST successfully
5 9 *****************
Enter 1: To Insert student
Enter 2: To Search for any student
Enter 3: if any student exits
Enter 4: To print the tree in order
Enter 5: To print level wise
Enter 6: To Exit
*****************
1
Enter the student roll no. as they enter the class
11
11 is inserted into BST successfully
5 9 11 *****************
Enter 1: To Insert student
Enter 2: To Search for any student
Enter 3: if any student exits
Enter 4: To print the tree in order
Enter 5: To print level wise
Enter 6: To Exit
```

```
Enter 3: if any student exits
Enter 4: To print the tree in order
Enter 5: To print level wise
Enter 6: To Exit
*****************
1
Enter the student roll no. as they enter the class
2
2 is inserted into BST successfully
2 5 9 11 *****************
Enter 1: To Insert student
Enter 2: To Search for any student
Enter 3: if any student exits
Enter 4: To print the tree in order
Enter 5: To print level wise
Enter 6: To Exit
*****************
2
Enter the student Roll No. for search
11
Student is PRESENT in the class
*****************
Enter 1: To Insert student
Enter 2: To Search for any student
Enter 3: if any student exits
Enter 4: To print the tree in order
Enter 5: To print level wise
Enter 6: To Exit
*****************
2
Enter the student Roll No. for search
14
Student is NOT PRESENT in the class
*****************
Enter 1: To Insert student
Enter 2: To Search for any student
Enter 3: if any student exits
Enter 4: To print the tree in order
Enter 5: To print level wise
Enter 6: To Exit
*****************
4
2 5 9 11 *****************
Enter 1: To Insert student
Enter 2: To Search for any student
Enter 3: if any student exits
Enter 4: To print the tree in order
```


input
Enter 6: To Exit
**********
5
5
2 9
11 **********
Enter 1: To Insert student
Enter 2: To Search for any student
Enter 3: if any student exits
Enter 4: To print the tree in order
Enter 5: To print level wise
Enter 6: To Exit
**********
3
Enter the student's Roll No. who exited
2
**********
Enter 1: To Insert student
Enter 2: To Search for any student
Enter 3: if any student exits
Enter 4: To print the tree in order
Enter 5: To print level wise
Enter 6: To Exit
**********
4
5 9 11 **********
Enter 1: To Insert student
Enter 2: To Search for any student
Enter 3: if any student exits
Enter 4: To print the tree in order
Enter 5: To print level wise
Enter 6: To Exit
**********
5
5
9
11 **********
Enter 1: To Insert student
Enter 2: To Search for any student
Enter 3: if any student exits
Enter 4: To print the tree in order
Enter 5: To print level wise
Enter 6: To Exit
**********
6
*******EXIT********

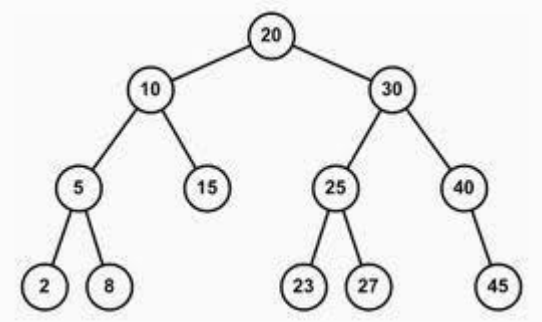
# MAYANK GUPTA 20BCE1538

| | | | | | |
|---|---|---|---|---|---|
| Programme | : | **B.Tech** | Semester | : | **Fall 2021-22** |
| Course | : | **Data structures and Algorithms** | Code | : | **CSE2003** |
| Faculty | : | **Dr.V.Vani** | Slot | : | **L27+L28** |

# AIM Depth First Search and Breadth First Search

## AIM

1. Consider the below tree with 12 nodes, start from the root node and explore all the nodes in each level and reach the node 45.




```
#include <iostream>
#include <queue>

template <typename T>
class BinaryTreeNode {
  public:
   T  data;
   BinaryTreeNode<T> *left;
   BinaryTreeNode<T> *right;

   BinaryTreeNode(T data) {
      this->data = data;
      left = NULL;
      right = NULL;
   }
   BinaryTreeNode() {
      if (left) delete left;
      if (right) delete right;
```

```
    }
};

using namespace std;
bool searchInBST(BinaryTreeNode<int> *root , int k) {
        // Write your code here
    if(root==NULL){
        return false;
    }
    if(root->data==k){
        return true;
    }
    else if(root->data>k){
        return searchInBST(root->left ,k);
    }
    else if(root->data<k){
        return searchInBST(root->right ,k);
    }

}

BinaryTreeNode<int> *takeInput() {
    int rootData;
    cin >> rootData;
    if (rootData == -1) {
        return NULL;
    }
    BinaryTreeNode<int> *root = new BinaryTreeNode<int>(rootData);
    queue<BinaryTreeNode<int> *> q;
    q.push(root);
    while (!q.empty()) {
        BinaryTreeNode<int> *currentNode = q.front();
        q.pop();
        int leftChild, rightChild;
        cin >> leftChild;
        if (leftChild != -1) {
            BinaryTreeNode<int> *leftNode = new BinaryTreeNode<int>(leftChild);
            currentNode->left = leftNode;
            q.push(leftNode);
        }
        cin >> rightChild;
        if (rightChild != -1) {
            BinaryTreeNode<int> *rightNode =
                new BinaryTreeNode<int>(rightChild);
            currentNode->right = rightNode;
            q.push(rightNode);
        }
    }
```

```
    return root;
}
void printLevelWise(BinaryTreeNode<int> *root) {
        // Write your code here
    queue<BinaryTreeNode<int>*> pendingNode;
    pendingNode.push(root);
    while(!pendingNode.empty()){
        BinaryTreeNode<int> *front=pendingNode.front();
            pendingNode.pop();
        cout<<front->data<<":";
        if(front->left!=NULL){
            cout<<"L:"<<front->left->data;
            pendingNode.push(front->left);
        }
        else{
            cout<<"L:"<<-1;
        }
        cout<<",";
        if(front->right!=NULL){
            cout<<"R:"<<front->right->data;
            pendingNode.push(front->right);
        }
        else{
            cout<<"R:"<<-1;
        }
        cout<<endl;
    }
}
int main() {
    BinaryTreeNode<int> *root = takeInput();
    int k;
    cin >> k;
    cout << ((searchInBST(root, k)) ? "true" : "false")<<endl;
    printLevelWise(root);
    if(searchInBST(root,k)){
        printLevelWise(root);
    }
    else{
        cout<<"Element not present";
    }
    delete root;
}
```

## BFS algorithm

A standard BFS implementation puts each vertex of the graph into one of two categories:

- Visited
- Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

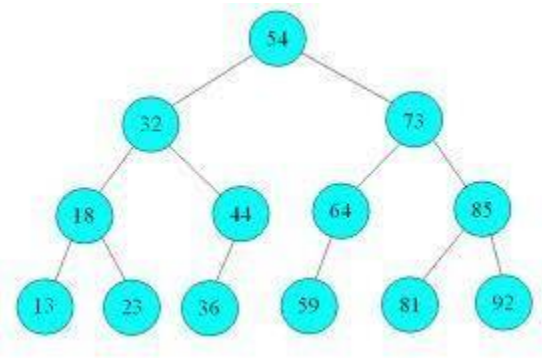
The algorithm works as follows:

- Start by putting any one of the graph's vertices at the back of a queue.
- Take the front item of the queue and add it to the visited list.
- Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
- Keep repeating steps 2 and 3 until the queue is empty.
- The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

```
true
20:L:10,R:30
10:L:5,R:15
30:L:25,R:40
5:L:2,R:8
15:L:-1,R:-1
25:L:23,R:27
40:L:-1,R:45
2:L:-1,R:-1
8:L:-1,R:-1
23:L:-1,R:-1
27:L:-1,R:-1
45:L:-1,R:-1
```

**AIM**

2. Consider the below tree with 13 nodes, start from the root node and explore all the nodes and reach the node 92 using DFS.




```
#include <iostream>
#include <queue>

template <typename T>
class BinaryTreeNode {
  public:
   T  data;
   BinaryTreeNode<T> *left;
   BinaryTreeNode<T> *right;
```

```
    BinaryTreeNode(T data) {
        this->data = data;
        left = NULL;
        right = NULL;
    }
    BinaryTreeNode() {
        if (left) delete left;
        if (right) delete right;
    }
};

using namespace std;
bool searchInBST(BinaryTreeNode<int> *root , int k) {
            // Write your code here
    if(root==NULL){
        return false;
    }
    if(root->data==k){
        return true;
    }
    else if(root->data>k){
            return searchInBST(root->left ,k);
    }
    else if(root->data<k){
        return searchInBST(root->right ,k);
    }

}

BinaryTreeNode<int> *takeInput() {
    int rootData;
    cin >> rootData;
    if (rootData == -1) {
        return NULL;
    }
    BinaryTreeNode<int> *root = new BinaryTreeNode<int>(rootData);
    queue<BinaryTreeNode<int> *> q;
    q.push(root);
    while (!q.empty()) {
        BinaryTreeNode<int> *currentNode = q.front();
        q.pop();
        int leftChild, rightChild;
        cin >> leftChild;
        if (leftChild != -1) {
            BinaryTreeNode<int> *leftNode = new BinaryTreeNode<int>(leftChild);
            currentNode->left = leftNode;
            q.push(leftNode);
        }
```

```
        cin >> rightChild;
        if (rightChild != -1) {
            BinaryTreeNode<int> *rightNode =
                new BinaryTreeNode<int>(rightChild);
            currentNode->right = rightNode;
            q.push(rightNode);
        }
    }
    return root;
}
void inorder(BinaryTreeNode<int> *tree)
{
    if (tree != NULL)
    {
        inorder(tree->left);
        cout << tree->data << " ";
        inorder(tree->right);
    }
}
int main() {
    BinaryTreeNode<int> *root = takeInput();
    int k;
    cin >> k;
    cout << ((searchInBST(root, k)) ? "true" : "false")<<endl;
    if(searchInBST(root,k)){
        inorder(root);
    }
    else{
        cout<<"Element not present";
    }
    delete root;
}
```

```
true
13 18 23 32 36 44 54 59 64 73 81 85 92
```

## Depth First Search Algorithm

A standard DFS implementation puts each vertex of the graph into one of two categories:

- Visited
- Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

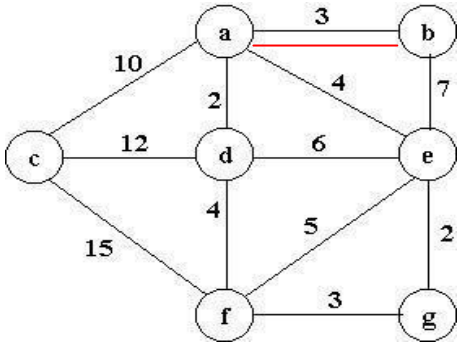
- Start by putting any one of the graph's vertices on top of a stack.
- Take the top item of the stack and add it to the visited list.

- Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
- Keep repeating steps 2 and 3 until the stack is empty.

**AIM** ***Minimum Spanning Tree Algorithm: Prim's and Kruskal***

## Assessment

1. Given a graph with weighted edges, obtain a minimal spanning tree using Prim's algorithm




Algorithm:

- **a.** Choose a starting vertex
- **b.** Start a loop till when all the nodes are reached
- **c.** Selec the edge connecting to the next node that has minimum weight which is not visited
- **d.** Add the selected edge and the vertex to the minimum spanning tree t
- **e.** Exit

Code:

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

#define INF 9999999

#define V  7

int G[V][V] = {
    {0, 3, 10, 2, 4, 0, 0},
    {3, 0, 0, 0, 7, 0, 0},
    {10, 0, 0, 12, 0, 15, 0},
```

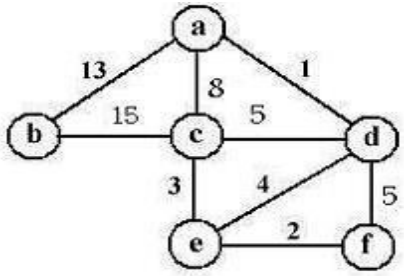
```
    {2, 0, 12, 0, 6, 4, 0},
    {4, 7, 0, 6, 0, 5, 2},
    {0, 0, 15, 4, 5, 0, 3},
    {0, 0, 0, 0, 2, 3, 0}};

int main() {
  int no_edge;
  int selected[V];
  memset(selected, false, sizeof(selected));
  no_edge = 0;
  selected[0] = true;

  int x;
  int y;

  printf("Edge : Weight\n");

  while (no_edge < V - 1) {

    int min = INF;
    x = 0;
    y = 0;

    for (int i = 0; i < V; i++)
    {
      if (selected[i])
      {
        for (int j = 0; j < V; j++)
        {
          if (!selected[j] && G[i][j])
          {
            if (min > G[i][j])
            {
              min = G[i][j];
              x = i;
              y = j;
            }
          }
        }
      }
    }
    printf("%d - %d : %d\n", x, y, G[x][y]);
    selected[y] = true;
    no_edge++;
  }

  return 0;
```

```
}
```

Output:

```
Edge : Weight
0 - 3 : 2
0 - 1 : 3
0 - 4 : 4
4 - 6 : 2
6 - 5 : 3
0 - 2 : 10
```

2. **The State Water department is planning to implement a new drinking water supply system that should connect its 6 districts. In the following graph, the nodes represent the districts and the values associated with the edges represent the cost of laying the water channel between the two districts. The graph shows all the possible connections among the nodes.Use Kruskal's algorithm to find the cost optimal solution for the given graph.**




**Code:**

```
#include <stdio.h>


#define V 6
int parent[V];
#define inf 99999


int find(int i)
{
  while (parent[i] != i)
    i = parent[i];
  return i;
}
```

```
void union1(int i, int j)
{
  int a = find(i);
  int b = find(j);
  parent[a] = b;
}


void kruskalMST(int cost[][V])
{
  int mincost = 0;


  for (int i = 0; i < V; i++)
    parent[i] = i;


  int edge_count = 0;
  while (edge_count < V - 1)
  {
    int min = inf, a = -1, b = -1;
    for (int i = 0; i < V; i++)
    {
      for (int j = 0; j < V; j++)
      {
        if (find(i) != find(j) && cost[i][j] < min)
        {
          min = cost[i][j];
          a = i;
          b = j;
        }
      }
    }


    union1(a, b);
    printf("Edge %d:(%d, %d) cost:%d \n",
           edge_count++, a, b, min);
    mincost += min;
  }
  printf("\n Minimum cost= %d \n", mincost);
}


int main()
```

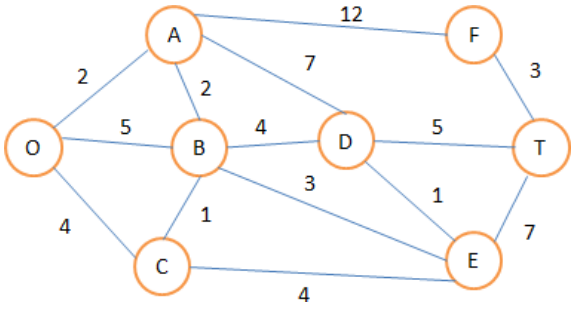
```
{
  /*the nodes in the question can be interpreted as:
  a=0;
  b=1;
  c=2;
  d=3;
  e=4;
  f=5;
  */
  int cost[][V] = {
      {inf, 13, 8, 1, inf, inf},
      {13, inf, 15, inf, inf, inf},
      {8, 15, inf, 5, 3, inf},
      {1, inf, 5, inf, 4, 5},
      {inf, inf, 3, 4, inf, 2},
      {inf, inf, inf, 5, 2, inf}};


  kruskalMST(cost);


  return 0;
}

Output:
Minimum Cost : 23
```

**Algorithm:**

a. **Sort all the edges from low weight to high**
b. **Take the edge with the lowest weight and add it to the spanning tree.**
c. **If adding the edge created a cycle, then reject the edge.**
d. **Keep adding the edge until all the vertices are visited**

## ***AIM*** *Single Source Shortest Path Algorithm*

3. Assume that, the management of a school (located in place O) has decided to operate the school buses for the convenience of their staff and students. They wanted to cover all the places that are interconnected as shown in the figure below. Suggest a suitable algorithm to find the shortest route from the school to all other locations for helping the management to plan their bus service. Illustrate the step-by-step procedure for calculation of the shortest path (source O destination F)


A
F
O
B
D
T
C
E
12
7
2
3
2
5
4
5
3
1
4
1
7
4

Algorithm:

1. Create a shortest path tree set that keeps track of vertices included
2. Assign all distances values as infinite
3. While shortest path tree doesn't contain all the vertex
    a. Pick a vertex which is not included and calculate the minimum distance
    b. Include that vertex int the tree
    c. Update the values of the distances as calcultated.

Code:

```
#include <limits.h>
#include <stdio.h>
#include <stdbool.h>


#define V 8


int minDistance(int dist[], bool sptSet[])
{
    int min = INT_MAX, min_index;


    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;


    return min_index;
}


void printSolution(int dist[])
{
    printf("Vertex \t\t Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t\t %d\n", i, dist[i]);
}


void dijkstra(int graph[V][V], int src)
{
    int dist[V];
    bool sptSet[V];
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;


    dist[src] = 0;
```

```
    for (int count = 0; count < V - 1; count++)
    {
        int u = minDistance(dist, sptSet);


        sptSet[u] = true;


        for (int v = 0; v < V; v++)


            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] +
graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }


    printSolution(dist);
}


int main()
{
    int graph[V][V] = {{0, 2, 5, 4, 0, 0, 0, 0},
                       {2, 0, 2, 0, 7, 0, 0, 12},
                       {5, 2, 0, 1, 4, 3, 0, 0},
                       {4, 0, 1, 0, 0, 4, 0, 0},
                       {0, 7, 4, 0, 0, 1, 5, 0},
                       {0, 0, 3, 4, 1, 0, 7, 0},
                       {0, 0, 0, 0, 5, 7, 0, 3},
                       {0, 12, 0, 0, 0, 0, 3, 0}};


    dijkstra(graph, 0);
```

```
    return 0;
}
```

Output:

# AIM Hashing

*Mayank Gupta*

*20BCE1538*

**AIM-The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table oflength 10 using open addressing with hash function h(k) = k mod 10 and linear probing. What is the resultant hash table.**

Explanation,

Open addressing, or closed hashing, is a method of collision resolution in hash tables. With this method a hash collision is resolved by probing, or searching through alternate locations in the array (the probe sequence) until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table. Well known probe sequences include:

*linear probing* in which the interval between probes is fixed–often at 1.

- 12 mod 10 = 2 //adding 12
- 18 mod 10 = 8 //adding 18
- 13 mod 10 = 3 //adding 13
- 2 mod 10 = 2 collision //adding 2
  (2 + 1) mod 10 = 3 again collision
  (using linear probing)
  ➔ (3 + 1) mod 10 = 4
- 3 mod 10 = 3 collision //adding 3
  (3 + 1) mod 10 = 4 again collision
  (using linear probing)
  ➔ (4 + 1) mod 10 = 5
- 23 mod 10 = 3 collision //adding 23
  (3 + 1) mod 10 = 4 collision
  (4 + 1) mod 10 = 5 again collision
  (using linear probing)
  ➔ (5 + 1) mod 10 = 6
- 5 mod 10 = 5 collision //adding 5
  (5 + 1) mod 10 = 6 again collision

➔ (6 + 1) mod 10 = 7

- 15 mod 10 = 5 collision //adding 15
  (5 + 1) mod 10 = 6 collision
  (6 + 1) mod 10 = 7 collision
  (7 + 1) mod 10 = 8 collision
  ➔ (8 + 1) mod 10 = 9 collision

So, resulting hash table

Final Table

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | 12 |
| 3 | 13 |
| 4 | 2 |
| 5 | 3 |
| 6 | 23 |
| 7 | 5 |
| 8 | 18 |
| 9 | 15 |