**Matrix addition**

#include<stdio.h>

void main()

{

int a[10][10],b[10][10],d[10][10],r,c,i,j;

printf("enter the size of row and column ");

scanf("%d%d",&r,&c);

printf("enter the elements in first matrix");

for(i=0;i<r;i++){

for(j=0;j<c;j++){

scanf("%d",&a[i][j]);

}

}

printf("enter the elements in second matrix");

for(i=0;i<r;i++){

for(j=0;j<c;j++){

scanf("%d",&b[i][j]);

}

}

printf("matrix after addition\n");

for(i=0;i<r;i++){

for(j=0;j<c;j++){

d[i][j]=a[i][j]+b[i][j];

printf("%d\t",d[i][j]);

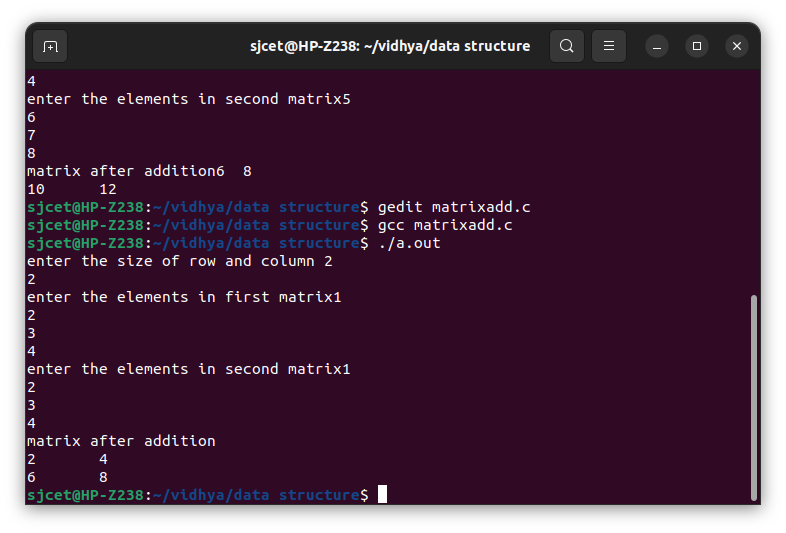
}

printf("\n");

}

}

**Output**

****

**Merging 2 sorted array**

#include <stdio.h>

void main()

{

int n1,n2,n3; //Array Size Declaration

int a[10000], b[10000], c[20000];

printf("Enter the size of first array: ");

scanf("%d",&n1);

printf("Enter the array elements: ");

for(int i = 0; i < n1; i++)

scanf("%d", &a[i]);

printf("Enter the size of second array: ");

scanf("%d",&n2);

printf("Enter the array elements: ");

for(int i = 0; i < n2; i++)

scanf("%d", &b[i]);

n3 = n1 + n2;

for(int i = 0; i < n1; i++)

c[i] = a[i];

for(int i = 0; i < n2; i++)

c[i + n1] = b[i];

printf("The merged array: ");

for(int i = 0; i < n3; i++)

printf("%d ", c[i]); //Print the merged array

printf("\nFinal array after sorting: ");

for(int i = 0; i < n3; i++){

int temp;

for(int j = i + 1; j < n3; j++) {

if(c[i] > c[j]) {

temp = c[i];

c[i] = c[j];

c[j] = temp;

}

}

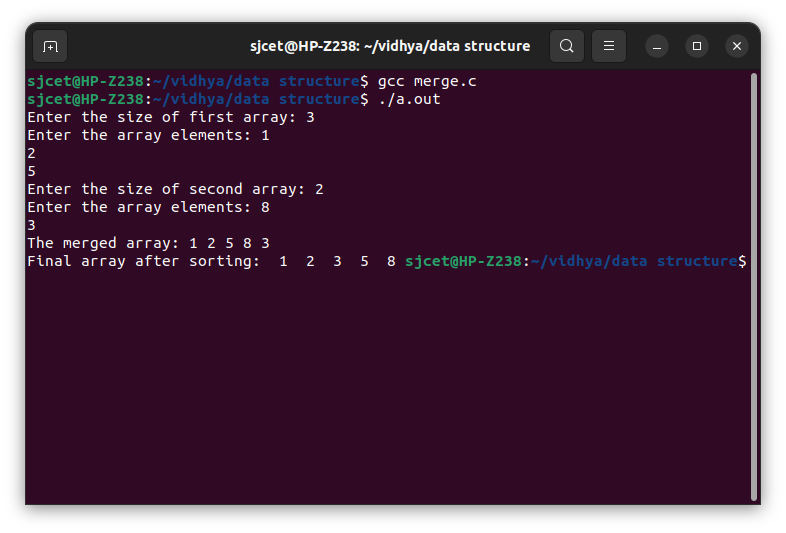
}

for(int i = 0; i < n3 ; i++) //Print the sorted Array

printf(" %d ",c[i]);

}

**Output**



**Binary search**

#include <stdio.h>

void main()

{

int i, low, high, mid, n, key, array[100];

printf("Enter number of elements");

scanf("%d",&n);

printf("Enter %d integer", n);

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

scanf("%d",&array[i]);

printf("Enter value to find");

scanf("%d", &key);

low = 0;

high = n - 1;

mid = (low+high)/2;

while (low <= high) {

if(array[mid] < key)

low = mid + 1;

else if (array[mid] == key) {

printf("%d found at location %d.n", key, mid);

break;

}

else

high = mid - 1;

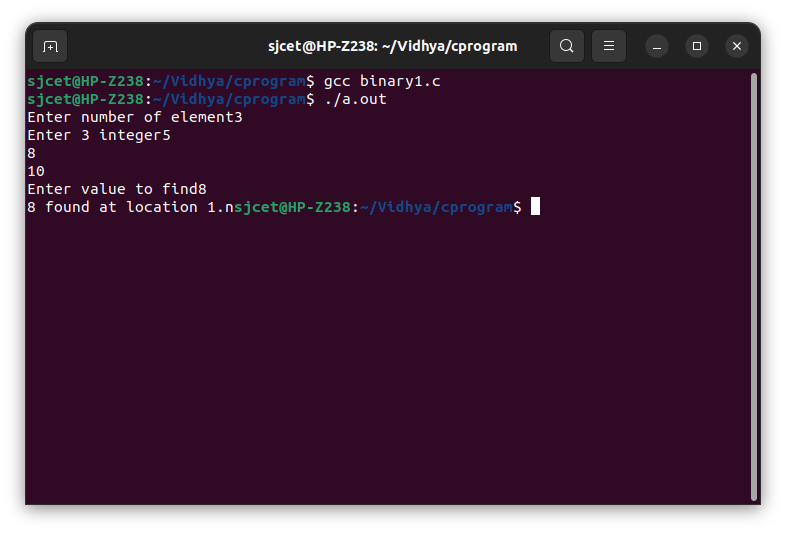
mid = (low + high)/2;

}

if(low > high)

printf("Not found! %d isn't present in the list.n", key);

}



**Linear Search**

#include<stdio.h>

void main()

{

int a[10],f=0;;

int s,n;

printf("enter the limit");

scanf("%d",&n);

printf("enter elements");

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

scanf("%d",&a[i]);

}

printf("enter the element to be searched");

scanf("%d",&s);

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

if(a[i]==s){

f=1;

printf("%d is find at %dth position",s,i);

}

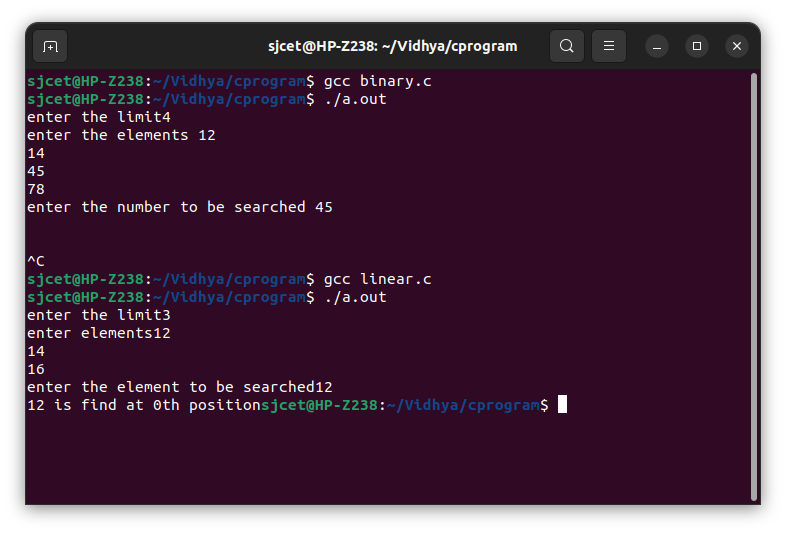
}

if(f==0)

{

printf("item not present");

**}}**

****

**Array insertion**

#include<stdio.h>

void main()

{

int a[10],n,i,pos,val;

printf("enter the limit ");

scanf("%d",&n);

printf("enter the elements ");

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

{

scanf("%d",&a[i]);

}

printf("enter the position to be value inserted");

scanf("%d",&pos);

printf("enter the value ");

scanf("%d",&val);

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

{

if(a[i]==pos)

{

a[i]=val;

}

}

printf("elements are.");

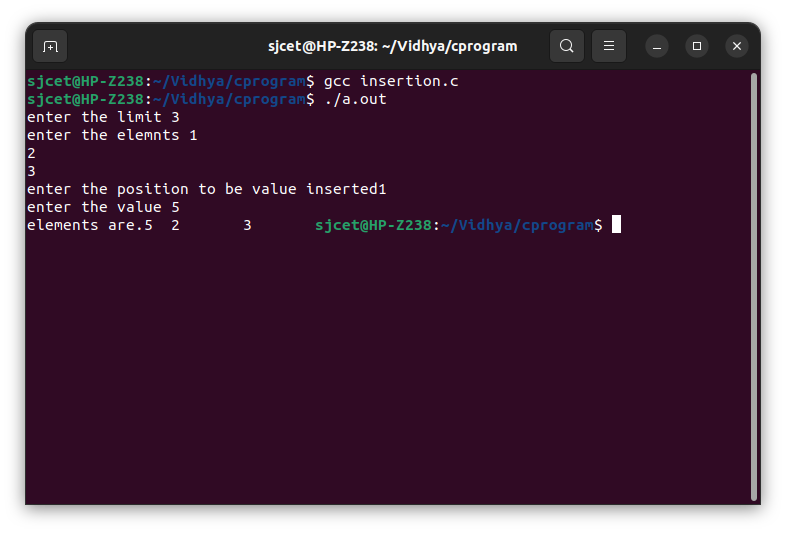
for(i=0;i<n;i++)

{

printf("%d\t",a[i]);

}

}



**Array deletion**

#include<stdio.h>

void main()

{

int a[10],n,i,pos;

printf("enter the limit ");

scanf("%d",&n);

printf("enter the elements ");

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

{

scanf("%d",&a[i]);

}

printf("enter the position of value to be deleted");

scanf("%d",&pos);

if(pos>=n+1)

printf("deletion is not possible");

else{

for(i=pos-1;i<n-1;i++)

{

a[i]=a[i+1];

printf("elements are.");

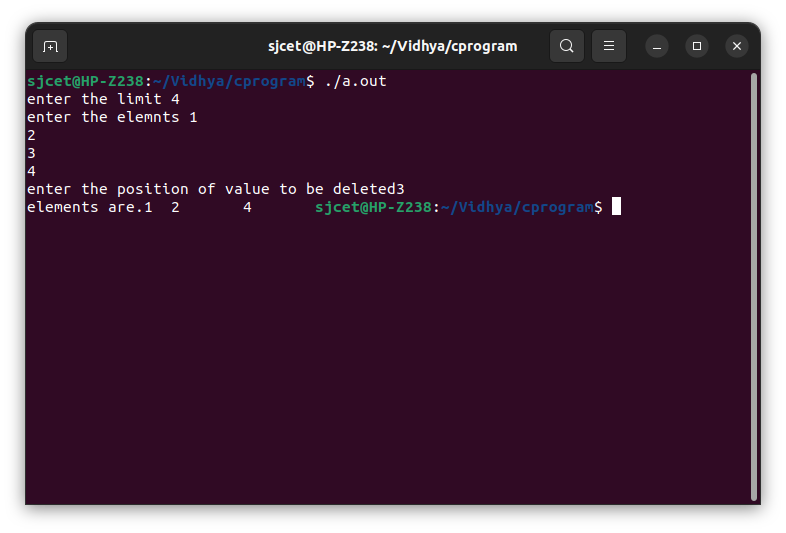
}

for(i=0;i<n-1;i++)

**printf("%d\t",a[i]);**

**}**

**}**

****

**Structure**

struct Car {

char brand[50];

char model[50];

int year;

};

#include<stdio.h>

void main() {

struct Car car1 = {"BMW", "X5", 1999};

struct Car car2 = {"Ford", "Mustang", 1969};

struct Car car3 = {"Toyota", "Corolla", 2011};

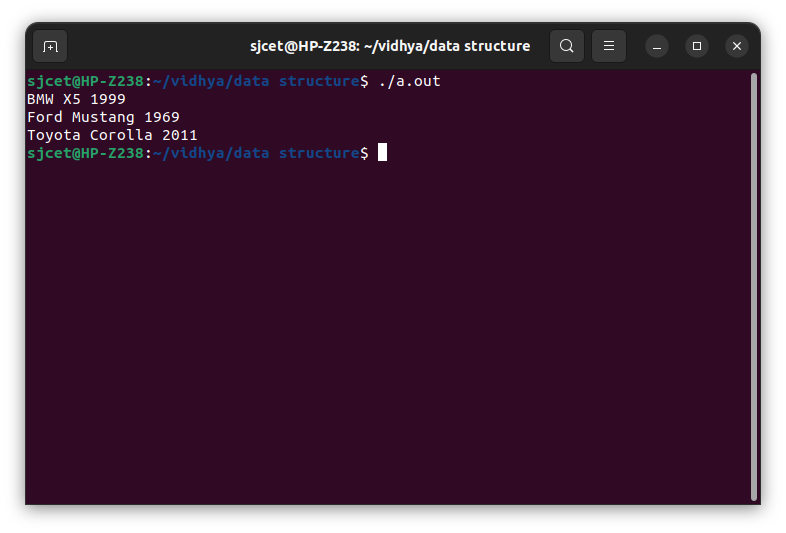
printf("%s %s %d\n", car1.brand, car1.model, car1.year);

printf("%s %s %d\n", car2.brand, car2.model, car2.year);

printf("%s %s %d\n", car3.brand, car3.model, car3.year);

}

**Output**

****

**Stack**

#include <stdio.h>

int stack[100],i,j,choice=0,n,top=-1;

void push();

void pop();

void show();

void main ()

{

printf("Enter the number of elements in the stack ");

scanf("%d",&n);

while(choice <4)

{

printf("Choose one from the below options...\n");

printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");

printf("\n Enter your choice \n");

scanf("%d",&choice);

switch(choice)

{

case 1:

push();

break;

case 2:

pop();

break;

case 3:

show();

break;

case 4:

printf("Exiting....");

break;

default:

printf("Please Enter valid choice ");

}

}

}

void push ()

{

int val;

if (top == n )

printf("\n Overflow");

else

{

printf("Enter the value?");

scanf("%d",&val);

top = top +1;

stack[top] = val;

}

}

void pop ()

{

if(top == -1)

printf("Underflow");

else

top = top -1;

}

void show()

{

for (i=top;i>=0;i--)

{

printf("%d\n",stack[i]);

}

if(top == -1)

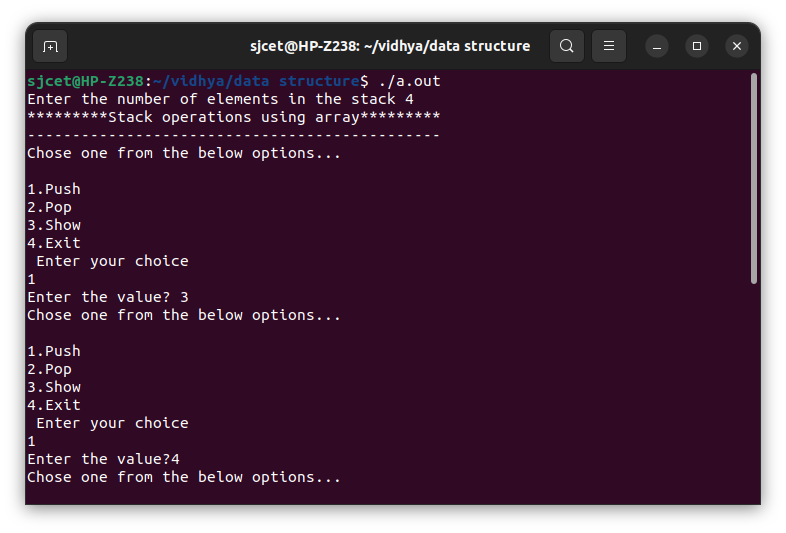
{

printf("Stack is empty");

}

}

**Output**

****

**Circular queue**

#include <stdio.h>

# define max 6

int queue[max]; // array declaration

int front=-1;

int rear=-1;

// function to insert an element in a circular queue

void enqueue(int element)

{

if(front==-1 && rear==-1) // condition to check queue is empty

{

front=0;

rear=0;

queue[rear]=element;

}

else if((rear+1)%max==front) // condition to check queue is full

{

printf("Queue is overflow..");

}

else

{

rear=(rear+1)%max; // rear is incremented

queue[rear]=element; // assigning a value to the queue at the rear position.

}

}

// function to delete the element from the queue

int dequeue()

{

if((front==-1) && (rear==-1)) // condition to check queue is empty

{

printf("\nQueue is underflow..");

}

else if(front==rear)

{

printf("\nThe dequeued element is %d", queue[front]);

front=-1;

rear=-1;

}

else

{

printf("\nThe dequeued element is %d", queue[front]);

front=(front+1)%max;

}

}

// function to display the elements of a queue

void display()

{

int i=front;

if(front==-1 && rear==-1)

{

printf("\n Queue is empty..");

}

else

{

printf("\nElements in a Queue are :");

while(i<=rear)

{

printf("%d,", queue[i]);

i=(i+1)%max;

}

}

}

void main()

{

int choice=1,x; // variables declaration

while(choice<4 && choice!=0) // while loop

{

printf("\n Press 1: Insert an element");

printf("\nPress 2: Delete an element");

printf("\nPress 3: Display the element");

printf("\nEnter your choice");

scanf("%d", &choice);

switch(choice)

{

case 1:

printf("Enter the element which is to be inserted");

scanf("%d", &x);

enqueue(x);

break;

case 2:

dequeue();

break;

case 3:

display();

}

}

}

**output**



**Queue implementation**

#include<stdio.h>

#include<stdlib.h>

#define maxsize 5

void insert();

void delete();

void display();

int front = -1, rear = -1;

int queue[maxsize];

void main ()

{

int choice;

while(choice != 4)

{

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("\n=================================================================\n");

printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");

printf("\nEnter your choice ?");

scanf("%d",&choice);

switch(choice)

{

case 1:

insert();

break;

case 2:

delete();

break;

case 3:

display();

break;

case 4:

exit(0);

break;

default:

printf("\nEnter valid choice??\n");

}

}

}

void insert()

{

int item;

printf("\nEnter the element\n");

scanf("\n%d",&item);

if(rear == maxsize-1)

{

printf("\nOVERFLOW\n");

return;

}

if(front == -1 && rear == -1)

{

front = 0;

rear = 0;

}

else

{

rear = rear+1;

}

queue[rear] = item;

printf("\nValue inserted ");

}

void delete()

{

int item;

if (front == -1 || front > rear)

{

printf("\nUNDERFLOW\n");

return;

}

else

{

item = queue[front];

if(front == rear)

{

front = -1;

rear = -1 ;

}

else

{

front = front + 1;

}

printf("\nvalue deleted ");

}

}

void display()

{

int i;

if(rear == -1)

{

printf("\nEmpty queue\n");

}

else

{ printf("\nprinting values .....\n");

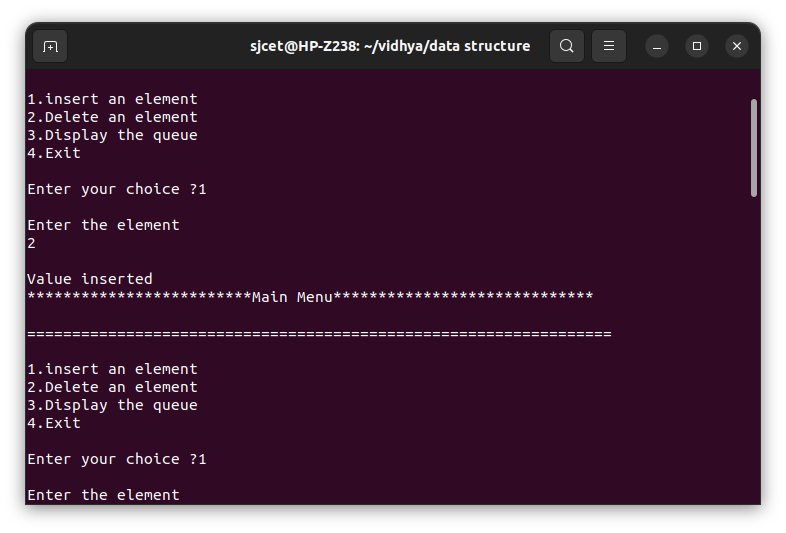
for(i=front;i<=rear;i++)

{

printf("\n%d\n",queue[i]);

}

**Output**



**Singly linked list**

**Code**

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

struct node {

int value;

struct node \*next;

};

void insert();

void display();

void delete();

int count();

typedef struct node DATA\_NODE;

DATA\_NODE \*head\_node, \*first\_node, \*temp\_node = 0, \*prev\_node, next\_node;

int data;

int main()

{

int option = 0;

printf("Singly Linked List :\n");

while (option < 5)

{

printf("\nOptions\n");

printf("1 : Insert into Linked List \n");

printf("2 : Delete from Linked List \n");

printf("3 : Display Linked List\n");

printf("4 : Count Linked List\n");

printf("5: Exit\n");

printf("Enter your option:");

scanf("%d", &option);

switch (option) {

case 1:insert();

break;

case 2:delete();

break;

case 3:display();

break;

case 4:count();

break;

case 5:exit(1);

break;

default:printf("Incorrect Choice.\n");

break;

}

}

return 0;

}

void insert() {

printf("\n Enter Element for Inserting in Linked List : \n");

scanf("%d", &data);

temp\_node = (DATA\_NODE \*) malloc(sizeof (DATA\_NODE));

temp\_node->value = data;

if (first\_node == 0) {

first\_node = temp\_node;

} else {

head\_node->next = temp\_node;

}

temp\_node->next = 0;

head\_node = temp\_node;

}

void delete() {

int countvalue, pos, i = 0;

countvalue = count();

temp\_node = first\_node;

printf("\n Enter Position for Delete Element : \n");

scanf("%d", &pos);

if (pos > 0 && pos <= countvalue) {

if (pos == 1) {

temp\_node = temp\_node -> next;

first\_node = temp\_node;

printf("\n Deleted Successfully \n\n");

} else {

while (temp\_node != 0) {

if (i == (pos - 1)) {

prev\_node->next = temp\_node->next;

if(i == (countvalue - 1))

{

head\_node = prev\_node;

}

printf("\n Deleted Successfully \n\n");

break;

} else {j

i++;

prev\_node = temp\_node;

temp\_node = temp\_node -> next;

}

}

}

} else

printf("\n Invalid Position \n\n");

}

void display() {

int count = 0;

temp\_node = first\_node;

printf("\n Display Linked List : \n");

while (temp\_node != 0) {

printf(" %d ", temp\_node->value);

count++;

temp\_node = temp\_node -> next;

}

}

int count() {

int count = 0;

temp\_node = first\_node;

while (temp\_node != 0) {

count++;

temp\_node = temp\_node -> next;

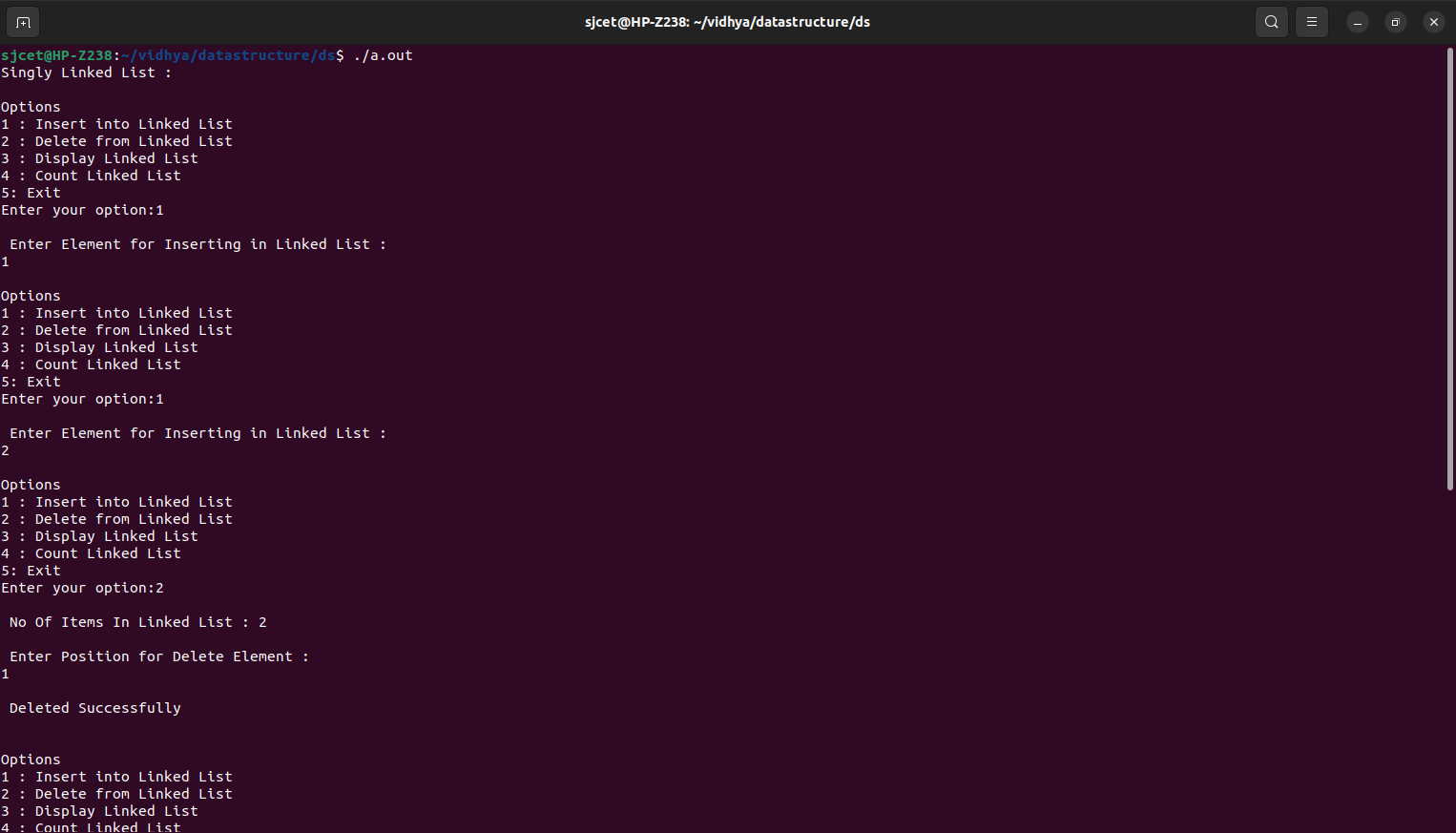
}

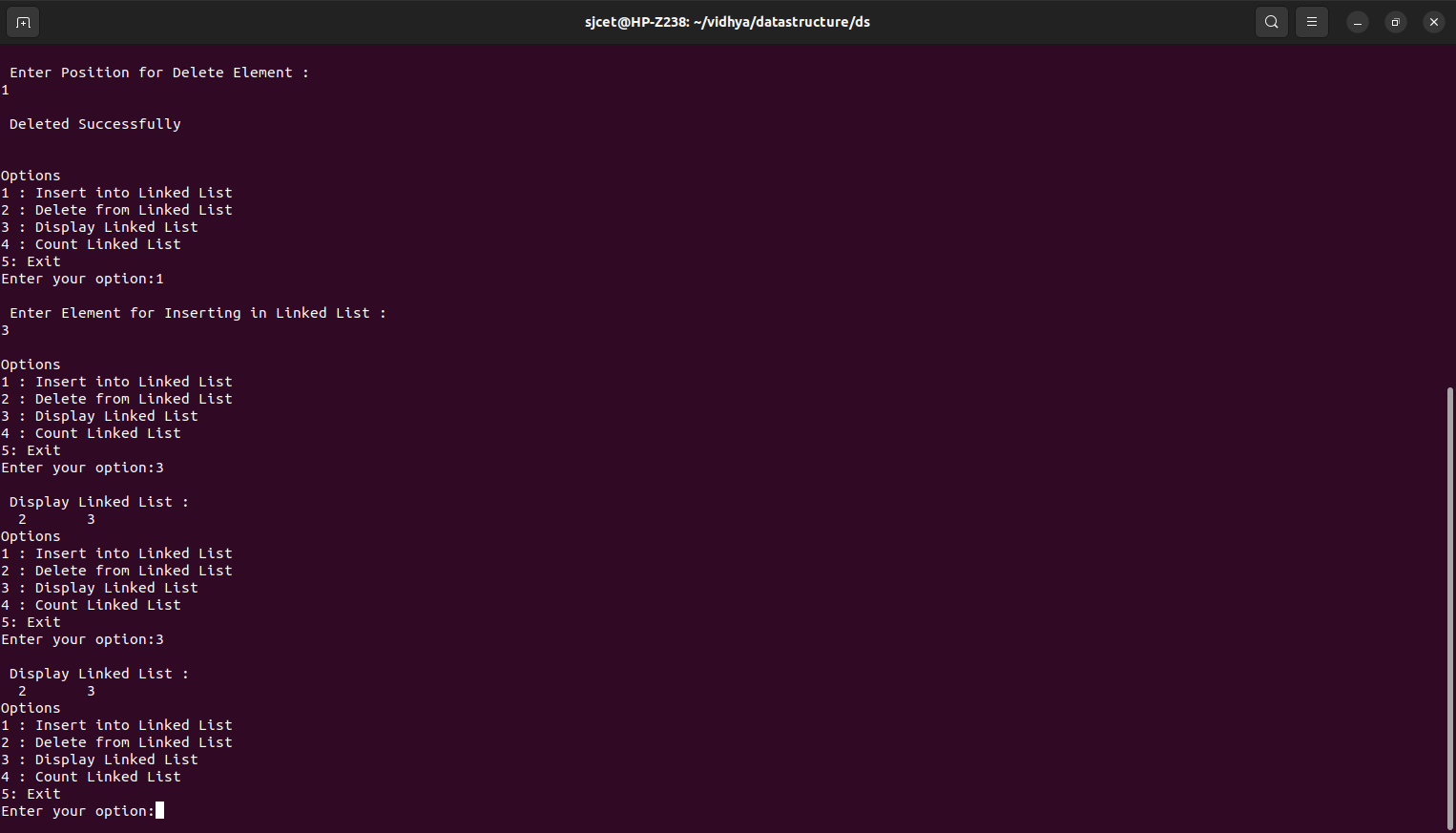
printf("\n No Of Items In Linked List : %d\n", count);

return count;

}

**output**

****

****

**DOUBLY LINKED LIST**

CODE

#include<stdio.h>

#include<stdlib.h>

struct node

{

struct node \*prev;

struct node \*next;

int data;

};

struct node \*head;

void insertion\_beginning();

void insertion\_last();

void insertion\_specified();

void deletion\_beginning();

void deletion\_last();

void deletion\_specified();

void display();

void search();

void main ()

{

int choice =0;

while(choice != 9)

{

printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete from Beginning\n5.Delete from last\n6.Delete the node after the given data\n7.Search\n8.Show\n9.Exit\n");

printf("\nEnter your choice?\n");

scanf("\n%d",&choice);

switch(choice)

{

case 1:

insertion\_beginning();

break;

case 2:

insertion\_last();

break;

case 3:

insertion\_specified();

break;

case 4:

deletion\_beginning();

break;

case 5:

deletion\_last();

break;

case 6:

deletion\_specified();

break;

case 7:

search();

break;

case 8:

display();

break;

case 9:

exit(0);

break;

default:

printf("Please enter valid choice..");

}

}

}

void insertion\_beginning()

{

struct node \*ptr;

int item;

ptr = (struct node \*)malloc(sizeof(struct node));

if(ptr == NULL)

{

printf("\nOVERFLOW");

}

else

{

printf("\nEnter Item value");

scanf("%d",&item);

if(head==NULL)

{

ptr->next = NULL;

ptr->prev=NULL;

ptr->data=item;

head=ptr;

}

else

{

ptr->data=item;

ptr->prev=NULL;

ptr->next = head;

head->prev=ptr;

head=ptr;

}

printf("\nNode inserted\n");

}

}

void insertion\_last()

{

struct node \*ptr,\*temp;

int item;

ptr = (struct node \*) malloc(sizeof(struct node));

if(ptr == NULL)

{

printf("\nOVERFLOW");

}

else

{

printf("\nEnter value");

scanf("%d",&item);

ptr->data=item;

if(head == NULL)

{

ptr->next = NULL;

ptr->prev = NULL;

head = ptr;

}

else

{

temp = head;

while(temp->next!=NULL)

{

temp = temp->next;

}

temp->next = ptr;

ptr ->prev=temp;

ptr->next = NULL;

}

}

printf("\nnode inserted\n");

}

void insertion\_specified()

{

struct node \*ptr,\*temp;

int item,loc,i;

ptr = (struct node \*)malloc(sizeof(struct node));

if(ptr == NULL)

{

printf("\n OVERFLOW");

}

else

{

temp=head;

printf("Enter the location");

scanf("%d",&loc);

for(i=0;i<loc;i++)

{

temp = temp->next;

if(temp == NULL)

{

printf("\n There are less than %d elements", loc);

return;

}

}

printf("Enter value");

scanf("%d",&item);

ptr->data = item;

ptr->next = temp->next;

ptr -> prev = temp;

temp->next = ptr;

temp->next->prev=ptr;

printf("\nnode inserted\n");

}

}

void deletion\_beginning()

{

struct node \*ptr;

if(head == NULL)

{

printf("\n UNDERFLOW");

}

else if(head->next == NULL)

{

head = NULL;

free(head);

printf("\nnode deleted\n");

}

else

{

ptr = head;

head = head -> next;

head -> prev = NULL;

free(ptr);

printf("\nnode deleted\n");

}

}

void deletion\_last()

{

struct node \*ptr;

if(head == NULL)

{

printf("\n UNDERFLOW");

}

else if(head->next == NULL)

{

head = NULL;

free(head);

printf("\nnode deleted\n");

}

else

{

ptr = head;

if(ptr->next != NULL)

{

ptr = ptr -> next;

}

ptr -> prev -> next = NULL;

free(ptr);

printf("\nnode deleted\n");

}

}

void deletion\_specified()

{

struct node \*ptr, \*temp;

int val;

printf("\n Enter the data after which the node is to be deleted : ");

scanf("%d", &val);

ptr = head;

while(ptr -> data != val)

ptr = ptr -> next;

if(ptr -> next == NULL)

{

printf("\nCan't delete\n");

}

else if(ptr -> next -> next == NULL)

{

ptr ->next = NULL;

}

else

{

temp = ptr -> next;

ptr -> next = temp -> next;

temp -> next -> prev = ptr;

free(temp);

printf("\nnode deleted\n");

}

}

void display()

{

struct node \*ptr;

printf("\n printing values...\n");

ptr = head;

while(ptr != NULL)

{

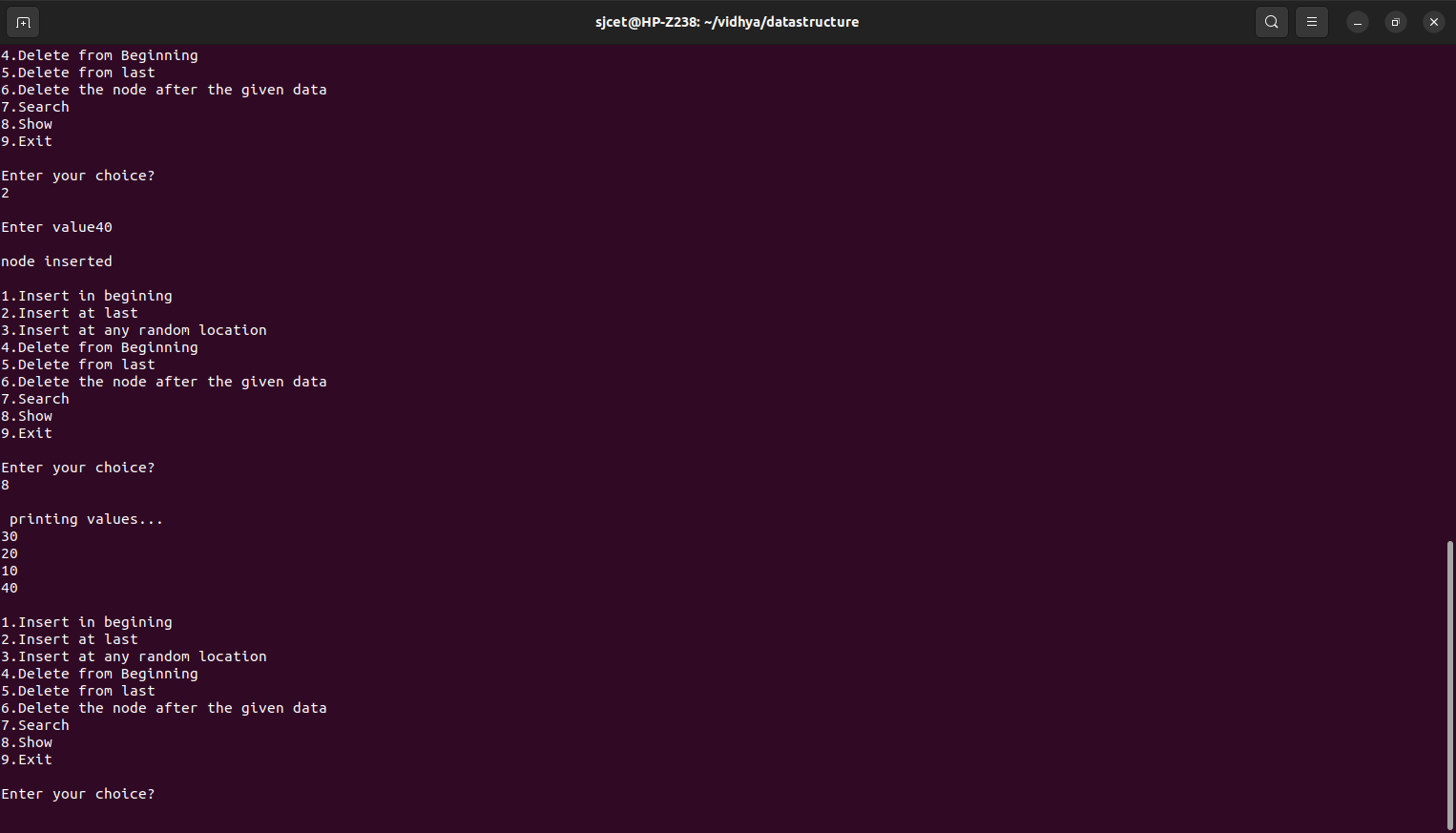
printf("%d\n",ptr->data);

ptr=ptr->next;

}

}

**Output**

**￼￼￼**

**BINARY SEARCH TREE**

**CODE**

#include<stdio.h>

#include<stdlib.h>

struct node{

int key;

struct node \*left,\*right;

};

struct node \*newnode(int item){

struct node \*temp=(struct node\*)malloc(sizeof(struct node));

temp->key=item;

temp->left=temp->right=NULL;

return temp;

}

void inorder(struct node \*root){

if(root!=NULL)

{

inorder(root->left);

printf("%d->",root->key);

inorder(root->right);

}

}

struct node \*insert(struct node \*node,int key)

{

if(node==NULL)

return newnode(key);

if(key<node->key)

node->left=insert(node->left,key);

else

node->right=insert(node->right,key);

return node;

}

struct node \*minvaluenode(struct node \*node)

{

struct node \*current=node;

while(current&&current->left!=NULL)

current=current->left;

return current;

}

struct node \*deletenode(struct node \*root,int key)

{

if(root==NULL)

return root;

if(key<root->key)

root->left=deletenode(root->left,key);

else if(key>root->key)

root->right=deletenode(root->right,key);

else

{

if(root->left==NULL)

{

struct node \*temp=root->right;

free(root);

return temp;

}

else if(root->right==NULL)

{

struct node \*temp=root->left;

free(root);

return temp;

}

struct node \*temp=minvaluenode(root->right);

root->key=temp->key;

root->right=deletenode(root->right,temp->key);

}

return root;

}

void main()

{

struct node \*root=NULL;

int choice,n;

while(1)

{

printf("\n1.insert");

printf("\n2.deletion");

printf("\n3.traversal");

printf("\n4.exit");

printf("\n enter your choice");

scanf("%d",&choice);

switch(choice)

{

case 1:printf("enter the elements to be inserted :");

scanf("%d",&n);

root=insert(root,n);

break;

case 2:printf("enter the elememnt to be deleted ");

scanf("%d",&n);

root=deletenode(root,n);

break;

case 3:printf("inorder traversal ");

inorder(root);

break;

case 4:exit(0);

break;

default:

printf("\n wrong choice:n");

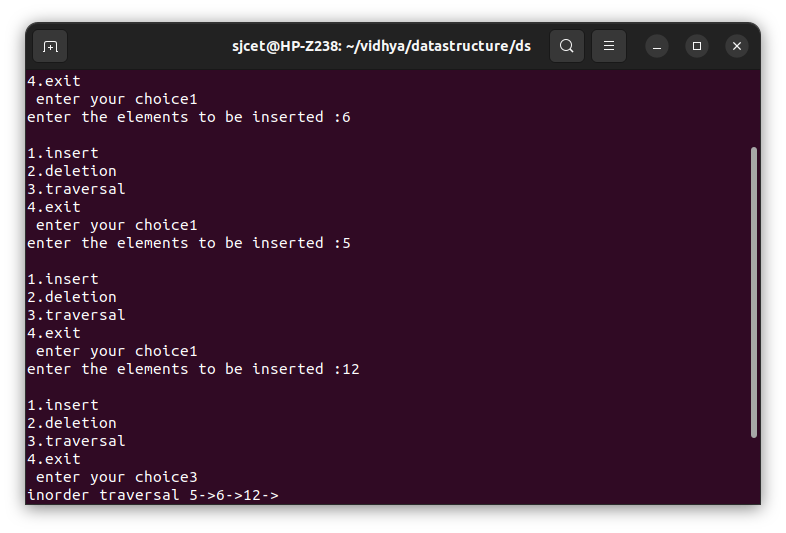
break;

}

}

}

**OUTPUT**



**SET OPERATION**

**CODE**

#include<stdio.h>

#include<conio.h>

main()

{

int i,j,k,p,ch,n1,n2,set1[10],set2[10], set3[20],flag;

int wish;

printf("\n Enter the size of sets1 \n");

scanf("%d",&n1);

printf("\n Enter the element of set1 \n" );

for(i=0;i<n1;i++)

scanf("%d",&set1[i]);

printf("\n Enter the size of sets2 \n");

scanf("%d",&n2);

printf("\n Enter the elements of set2 \n" );

for(i=0;i<n2;i++)

scanf("%d",&set2[i]);

do

{

printf("\n Menu for set operations");

printf("\n Press 1 for UNION");

printf("\n press 2 for INTERSECTION");

printf("\n press 3 for DIFFERENCE");

printf("\n Enter your Choice: ");

scanf("%d",&ch);

switch(ch)

{

case 1:

k=0;

for(i=0;i<n1;i++)

{

set3[k]=set1[i];

k++;

}

for(i=0;i<n2;i++)

{

flag=1;

for(j=0;j<n1;j++)

{

if(set2[i]==set1[j])

{

flag=0;

break;

}

}

if(flag==1)

{

set3[k]=set2[i];

k++;

}

}

p=k;

for(k=0;k <p;k++)

{

printf(" %d",set3[k]);

}

break;

case 2:

k=0;

for(i=0;i<n2;i++)

{

flag=1;

for(j=0;j<n1;j++)

{

if(set2[i]==set1[j])

{

flag=0;

break;

}

}

if(flag==0)

{

set3[k]=set2[i];

k++;

}

}

p=k;

for(k=0;k<p;k++)

{

printf("\t%d",set3[k]);

}

break;

case 3:

k=0;

for(i=0;i<n1;i++)

{

flag=1;

for(j=0;j<n2;j++)

{

if(set1[i]==set2[j])

{

flag=0;

break;

}

}

if(flag==1)

{

set3[k]=set1[i];

k++;

}

}

p=k;

for(k=0;k <p;k++)

{

printf(" %d ",set3[k]);

}

break;

}

printf("\n Do you want to continue(0/1)? ");

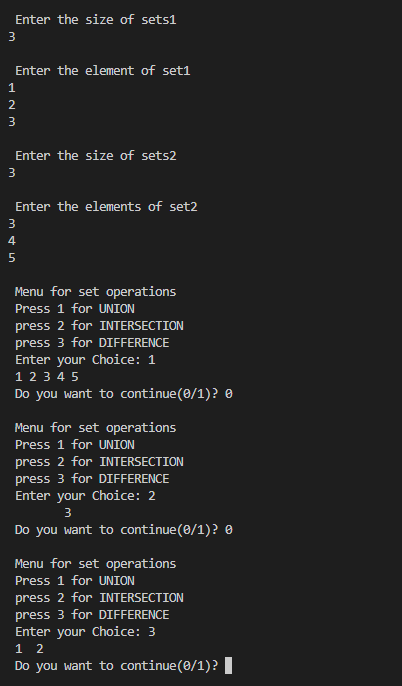
scanf("%d",&wish);

}

while(wish==0);

}

**OUTPUT**



**DISJOINT SET**

**CODE:**

#include<stdio.h>

struct disjointSet {

int parent[10];

int rank[10];

int n;

}

dis;

void makeset()

{

int i;

for(i=0;i<dis.n;i++)

dis.parent[i]=i;

dis.rank[i]=0;

}

void displayset()

{

int i;

printf("\nparent array\n");

for(i=0;i<dis.n;i++)

{

printf("%d",dis.parent[i]);

}

printf("\nrank of array\n");

for(i=0;i<dis.n;i++)

{

printf("%d",dis.rank[i]);

}

printf("\n");

}

int find(int x)

{

if(dis.parent[x]!=x)

{

dis.parent[x]=find(dis.parent[x]);

}

return dis.parent[x];

}

void Union(int x,int y)

{

int xset=find(x) , yset=find(y);

if(xset==yset)

return;

if(dis.rank[xset]<dis.rank[yset])

{

dis.parent[xset]=yset;

dis.rank[xset]=-1;

}

else if(dis.rank[xset]>dis.rank[yset])

{

dis.parent[yset]=xset;

dis.rank[yset]=-1;

}

else

{

dis.parent[yset]=xset;

dis.rank[xset]=dis.rank[xset]+1;

dis.rank[yset]=-1;

}

}

int main()

{

int x,y,n;

printf("\nenter number of elements :\n");

scanf("%d",&dis.n);

makeset();

int ch,w;

do{

printf("\n1.UNION\n2.FIND \n3.DISPLAY");

printf("\nenter choice :");

scanf("%d",&ch);

switch(ch)

{

case 1:

printf("\nenter elements to perform union :");

scanf("%d%d",&x,&y);

Union(x,y);

break;

case 2:

printf("\nenter elements to check if connected components :");

scanf("%d%d",&x, &y);

if(find(x)==find(y))

printf("\n connected components !");

else

printf("\n no connected components !");

break;

case 3:

displayset();

break;

}

printf("\n do you want to continue ?(1/0)");

scanf("%d",&w);

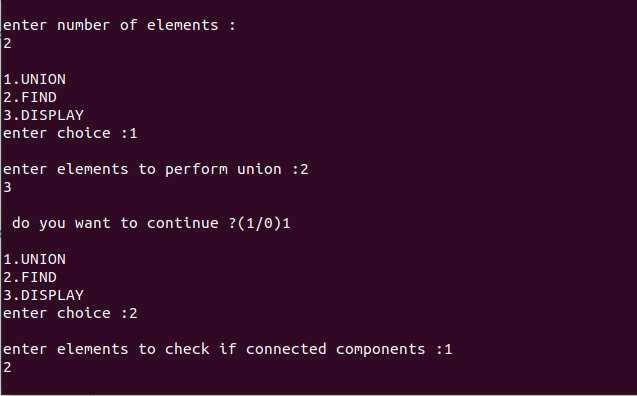
}

while(w==1);

return 0;

}

**OUTPUT**

****

**BALANCED BINARY SEARCH TREE**

**CODE:**

#include <stdio.h>

#include <stdlib.h>

#define bool int

struct node {

int item;

struct node \*left;

struct node \*right;

};

struct node \*newNode(int item) {

struct node \*node = (struct node \*)malloc(sizeof(struct node));

node->item = item;

node->left = NULL;

node->right = NULL;

return (node);

}

bool checkHeightBalance(struct node \*root, int \*height) {

int leftHeight = 0, rightHeight = 0;

int l = 0, r = 0;

if (root == NULL) {

\*height = 0;

return 1;

}

l = checkHeightBalance(root->left,&leftHeight);

r = checkHeightBalance(root->right,&rightHeight);

\*height = (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;

if ((leftHeight - rightHeight >= 2) || (rightHeight - leftHeight >= 2))

return 0;

else

return l && r;

}

int main()

{

int height = 0;

struct node \*root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

if (checkHeightBalance(root,&height))

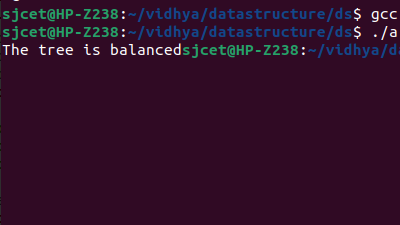
printf("The tree is balanced");

else

printf("The tree is not balanced");

}

**OUTPUT**

****

**Max heap Implementation**

**code**

#include<stdio.h>

int size = 0;

void swap(int \*a, int \*b)

{

int temp = \*b;

\*b = \*a;

\*a = temp;

}

void heapify(int array[], int size, int i)

{

if (size == 1)

{

printf("Single element in the heap");

}

else

{

int largest = i;

int l = 2 \* i + 1;

int r = 2 \* i + 2;

if (l < size && array[l] > array[largest])

largest = l;

if (r < size && array[r] > array[largest])

largest = r;

if (largest != i)

{

swap(&array[i], &array[largest]);

heapify(array, size, largest);

}

}

}

void insert(int array[], int newNum)

{

if (size == 0)

{

array[0] = newNum;

size += 1;

}

else

{

array[size] = newNum;

size += 1;

for (int i = size / 2 - 1; i>= 0; i--)

{

heapify(array, size, i);

}

}

}

void deleteRoot(int array[], int num)

{

int i;

for (i = 0; i<size; i++)

{

if (num == array[i])

break;

}

swap(&array[i], &array[size - 1]);

size -= 1;

for (int i = size / 2 - 1; i>= 0; i--)

{

heapify(array, size, i);

}

}

void printArray(int array[], int size)

{

for (int i = 0; i<size; ++i)

printf("%d", array[i]);

printf("\n");

}

int main()

{

int array[10];

insert(array, 3);

insert(array, 4);

insert(array, 9);

insert(array, 5);

insert(array, 2);

printf("Max-Heap array:");

printArray(array, size);

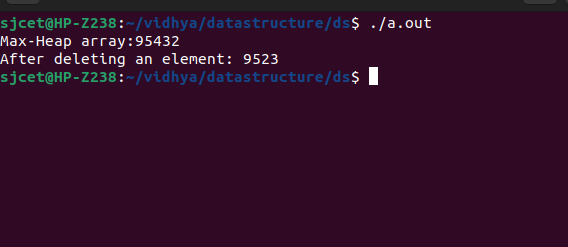
deleteRoot(array, 4);

printf("After deleting an element: ");

printArray(array, size);

}

**OUTPUT**

****

**Minheap Implementation**

**Code**

#include<stdio.h>

#include<limits.h>

int heap[1000000], heapSize;

void Init() {

heapSize = 0;

heap[0] = -INT\_MAX;

}

void Insert(int element) {

heapSize++;

heap[heapSize] = element;

int now = heapSize;

while (heap[now / 2] > element) {

heap[now] = heap[now / 2];

now /= 2;

}

heap[now] = element;

}

int DeleteMin() {

int minElement, lastElement, child, now;

minElement = heap[1];

lastElement = heap[heapSize--];

for (now = 1; now \* 2 <= heapSize; now = child) {

child = now \* 2;

if (child != heapSize && heap[child + 1] < heap[child]) {

child++;

}

if (lastElement > heap[child]) {

heap[now] = heap[child];

} else

{

break;

}

}

heap[now] = lastElement;

return minElement;

}

int main() {

int number\_of\_elements;

printf("Program to demonstrate Heap:\nEnter the number of elements: ");

scanf("%d", &number\_of\_elements);

int iter, element;

Init();

printf("Enter the elements: ");

for (iter = 0; iter < number\_of\_elements; iter++) {

scanf("%d", &element);

Insert(element);

}

for (iter = 0; iter < number\_of\_elements; iter++) {

printf("%d ", DeleteMin());

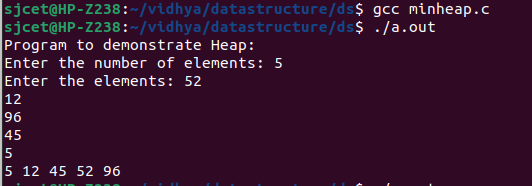
}

printf("\n");

return 0;

}

**OUTPUT**

****

**B-TREE IMPLEMENTATION**

**CODE:**

#include <stdio.h>

#include <stdlib.h>

#define MAX 3

#define MIN 2

struct BTreeNode {

int val[MAX + 1], count;

struct BTreeNode \*link[MAX + 1];

};

struct BTreeNode \*root;

// Create a node

struct BTreeNode \*createNode(int val, struct BTreeNode \*child) {

struct BTreeNode \*newNode;

newNode = (struct BTreeNode \*)malloc(sizeof(struct BTreeNode));

newNode->val[1] = val;

newNode->count = 1;

newNode->link[0] = root;

newNode->link[1] = child;

return newNode;

}

// Insert node

void insertNode(int val, int pos, struct BTreeNode \*node,struct BTreeNode \*child)

{

int j = node->count;

while (j > pos) {

node->val[j + 1] = node->val[j];

node->link[j + 1] = node->link[j];

j--;

}

node->val[j + 1] = val;

node->link[j + 1] = child;

node->count++;

}

// Split node

void splitNode(int val, int \*pval, int pos, struct BTreeNode \*node,

struct BTreeNode \*child, struct BTreeNode \*\*newNode) {

int median, j;

if (pos > MIN)

median = MIN + 1;

else

median = MIN;

\*newNode = (struct BTreeNode \*)malloc(sizeof(struct BTreeNode));

j = median + 1;

while (j <= MAX) {

(\*newNode)->val[j - median] = node->val[j];

(\*newNode)->link[j - median] = node->link[j];

j++;

}

node->count = median;

(\*newNode)->count = MAX - median;

if (pos <= MIN) {

insertNode(val, pos, node, child);

} else {

insertNode(val, pos - median, \*newNode, child);

}

\*pval = node->val[node->count];

(\*newNode)->link[0] = node->link[node->count];

node->count--;

}

// Set the value

int setValue(int val, int \*pval,

struct BTreeNode \*node, struct BTreeNode \*\*child) {

int pos;

if (!node) {

\*pval = val;

\*child = NULL;

return 1;

}

if (val < node->val[1]) {

pos = 0;

} else {

for (pos = node->count;

(val < node->val[pos] && pos > 1); pos--)

;

if (val == node->val[pos]) {

printf("Duplicates are not permitted\n");

return 0;

}

}

if (setValue(val, pval, node->link[pos], child)) {

if (node->count < MAX) {

insertNode(\*pval, pos, node, \*child);

} else {

splitNode(\*pval, pval, pos, node, \*child, child);

return 1;

}

}

return 0;

}

// Insert the value

void insert(int val) {

int flag, i;

struct BTreeNode \*child;

flag = setValue(val, &i, root, &child);

if (flag)

root = createNode(i, child);

}

// Search node

void search(int val, int \*pos, struct BTreeNode \*myNode) {

if (!myNode) {

return;

}

if (val < myNode->val[1]) {

\*pos = 0;

} else {

for (\*pos = myNode->count;

(val < myNode->val[\*pos] && \*pos > 1); (\*pos)--)

;

if (val == myNode->val[\*pos]) {

printf("%d is found", val);

return;

}

}

search(val, pos, myNode->link[\*pos]);

return;

}

// Traverse then nodes

void traversal(struct BTreeNode \*myNode) {

int i;

if (myNode) {

for (i = 0; i < myNode->count; i++) {

traversal(myNode->link[i]);

printf("%d ", myNode->val[i + 1]);

}

traversal(myNode->link[i]);

}

}

int main() {

int val, ch;

insert(8);

insert(9);

insert(10);

insert(11);

insert(15);

insert(16);

insert(17);

insert(18);

insert(20);

insert(23);

traversal(root);

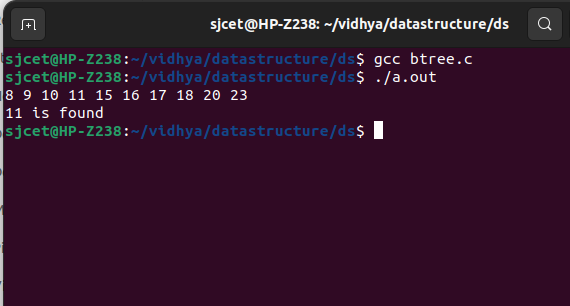
printf("\n");

search(11, &ch, root);

printf("\n");

}

**OUTPUT**



**RED BLACK TREE**

**CODE:**

#include<stdio.h>

#include<stdlib.h>

enum nodeColor {

RED,

BLACK

};

struct rbNode {

int data, color;

struct rbNode \*link[2];

};

struct rbNode \*root = NULL;

// Create a red-black tree

struct rbNode \*createNode(int data) {

struct rbNode \*newnode;

newnode = (struct rbNode \*)malloc(sizeof(struct rbNode));

newnode->data = data;

newnode->color = RED;

newnode->link[0] = newnode->link[1] = NULL;

return newnode;

}

// Insert an node

void insertion(int data) {

struct rbNode \*stack[98], \*ptr, \*newnode, \*xPtr, \*yPtr;

int dir[98], ht = 0, index;

ptr = root;

if (!root) {

root = createNode(data);

return;

}

stack[ht] = root;

dir[ht++] = 0;

while (ptr != NULL) {

if (ptr->data == data) {

printf("Duplicates Not Allowed!!\n");

return;

}

index = (data - ptr->data)> 0 ? 1 : 0;

stack[ht] = ptr;

ptr = ptr->link[index];

dir[ht++] = index;

}

stack[ht - 1]->link[index] = newnode = createNode(data);

while ((ht >= 3) && (stack[ht - 1]->color == RED)) {

if (dir[ht - 2] == 0) {

yPtr = stack[ht - 2]->link[1];

if (yPtr != NULL && yPtr->color == RED) {

stack[ht - 2]->color = RED;

stack[ht - 1]->color = yPtr->color = BLACK;

ht = ht - 2;

} else {

if (dir[ht - 1] == 0) {

yPtr = stack[ht - 1];

} else {

xPtr = stack[ht - 1];

yPtr = xPtr->link[1];

xPtr->link[1] = yPtr->link[0];

yPtr->link[0] = xPtr;

stack[ht - 2]->link[0] = yPtr;

}

xPtr = stack[ht - 2];

xPtr->color = RED;

yPtr->color = BLACK;

xPtr->link[0] = yPtr->link[1];

yPtr->link[1] = xPtr;

if (xPtr == root) {

root = yPtr;

} else {

stack[ht - 3]->link[dir[ht - 3]] = yPtr;

}

break;

}

} else {

yPtr = stack[ht - 2]->link[0];

if ((yPtr != NULL) &&(yPtr->color == RED)) {

stack[ht - 2]->color = RED;

stack[ht - 1]->color = yPtr->color = BLACK;

ht = ht - 2;

} else {

if (dir[ht - 1] == 1) {

yPtr = stack[ht - 1];

} else {

xPtr = stack[ht - 1];

yPtr = xPtr->link[0];

xPtr->link[0] = yPtr->link[1];

yPtr->link[1] = xPtr;

stack[ht - 2]->link[1] = yPtr;

}

xPtr = stack[ht - 2];

yPtr->color = BLACK;

xPtr->color = RED;

xPtr->link[1] = yPtr->link[0];

yPtr->link[0] = xPtr;

if (xPtr == root) {

root = yPtr;

} else {

stack[ht - 3]->link[dir[ht - 3]] = yPtr;

}

break;

}

}

}

root->color = BLACK;

}

// Delete a node

void deletion(int data) {

struct rbNode \*stack[98], \*ptr, \*xPtr, \*yPtr;

struct rbNode \*pPtr, \*qPtr, \*rPtr;

int dir[98], ht = 0, diff, i;

enum nodeColor color;

if (!root) {

printf("Tree not available\n");

return;

}

ptr = root;

while (ptr != NULL) {

if ((data - ptr->data) == 0)

break;

diff = (data - ptr->data) > 0 ? 1 : 0;

stack[ht] = ptr;

dir[ht++] = diff;

ptr = ptr->link[diff];

}

if (ptr->link[1] == NULL) {

if ((ptr == root) && (ptr->link[0] == NULL)) {

free(ptr);

root = NULL;

} else if (ptr == root) {

root = ptr->link[0];

free(ptr);

} else {

stack[ht - 1]->link[dir[ht - 1]] = ptr->link[0];

}

} else {

xPtr = ptr->link[1];

if (xPtr->link[0] == NULL) {

xPtr->link[0] = ptr->link[0];

color = xPtr->color;

xPtr->color = ptr->color;

ptr->color = color;

if (ptr == root) {

root = xPtr;

} else {

stack[ht - 1]->link[dir[ht - 1]] = xPtr;

}

dir[ht] = 1;

stack[ht++] = xPtr;

} else {

i = ht++;

while (1) {

dir[ht] = 0;

stack[ht++] = xPtr;

yPtr = xPtr->link[0];

if (!yPtr->link[0])

break;

xPtr = yPtr;

}

dir[i] = 1;

stack[i] = yPtr;

if (i > 0)

stack[i - 1]->link[dir[i - 1]] = yPtr;

yPtr->link[0] = ptr->link[0];

xPtr->link[0] = yPtr->link[1];

yPtr->link[1] = ptr->link[1];

if (ptr == root) {

root = yPtr;

}

color = yPtr->color;

yPtr->color = ptr->color;

ptr->color = color;

}

}

if (ht<1)

return;

if (ptr->color == BLACK) {

while (1) {

pPtr = stack[ht - 1]->link[dir[ht - 1]];

if (pPtr && pPtr->color == RED) {

pPtr->color = BLACK;

break;

}

if (ht< 2)

break;

if (dir[ht - 2] == 0) {

rPtr = stack[ht - 1]->link[1];

if (!rPtr)

break;

if (rPtr->color == RED) {

stack[ht - 1]->color = RED;

rPtr->color = BLACK;

stack[ht - 1]->link[1] = rPtr->link[0];

rPtr->link[0] = stack[ht - 1];

if (stack[ht - 1] == root) {

root = rPtr;

} else {

stack[ht - 2]->link[dir[ht - 2]] = rPtr;

}

dir[ht] = 0;

stack[ht] = stack[ht - 1];

stack[ht - 1] = rPtr;

ht++;

rPtr = stack[ht - 1]->link[1];

}

if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&

(!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {

rPtr->color = RED;

} else {

if (!rPtr->link[1] || rPtr->link[1]->color == BLACK) {

qPtr = rPtr->link[0];

rPtr->color = RED;

qPtr->color = BLACK;

rPtr->link[0] = qPtr->link[1];

qPtr->link[1] = rPtr;

rPtr = stack[ht - 1]->link[1] = qPtr;

}

rPtr->color = stack[ht - 1]->color;

stack[ht - 1]->color = BLACK;

rPtr->link[1]->color = BLACK;

stack[ht - 1]->link[1] = rPtr->link[0];

rPtr->link[0] = stack[ht - 1];

if (stack[ht - 1] == root) {

root = rPtr;

} else {

stack[ht - 2]->link[dir[ht - 2]] = rPtr;

}

break;

}

} else {

rPtr = stack[ht - 1]->link[0];

if (!rPtr)

break;

if (rPtr->color == RED) {

stack[ht - 1]->color = RED;

rPtr->color = BLACK;

stack[ht - 1]->link[0] = rPtr->link[1];

rPtr->link[1] = stack[ht - 1];

if (stack[ht - 1] == root) {

root = rPtr;

} else {

stack[ht - 2]->link[dir[ht - 2]] = rPtr;

}

dir[ht] = 1;

stack[ht] = stack[ht - 1];

stack[ht - 1] = rPtr;

ht++;

rPtr = stack[ht - 1]->link[0];

}

if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&

(!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {

rPtr->color = RED;

} else {

if (!rPtr->link[0] || rPtr->link[0]->color == BLACK) {

qPtr = rPtr->link[1];

rPtr->color = RED;

qPtr->color = BLACK;

rPtr->link[1] = qPtr->link[0];

qPtr->link[0] = rPtr;

rPtr = stack[ht - 1]->link[0] = qPtr;

}

rPtr->color = stack[ht - 1]->color;

stack[ht - 1]->color = BLACK;

rPtr->link[0]->color = BLACK;

stack[ht - 1]->link[0] = rPtr->link[1];

rPtr->link[1] = stack[ht - 1];

if (stack[ht - 1] == root) {

root = rPtr;

} else {

stack[ht - 2]->link[dir[ht - 2]] = rPtr;

}

break;

}

}

ht--;

}

}

}

// Print the inorder traversal of the tree

void inorderTraversal(struct rbNode \*node) {

if (node) {

inorderTraversal(node->link[0]);

printf("%d ", node->data);

inorderTraversal(node->link[1]);

}

return;

}

// Driver code

int main() {

int ch, data;

printf("1. Insertion\t2. Deletion\n");

printf("3. Traverse\t4. Exit");

while (1) {

printf("\nEnter your choice:");

scanf("%d", &ch);

switch (ch)

{

case 1:

printf("Enter the element to insert:");

scanf("%d", &data);

insertion(data);

break;

case 2:

printf("Enter the element to delete:");

scanf("%d", &data);

deletion(data);

break;

case 3:

inorderTraversal(root);

printf("\n");

break;

case 4:

exit(0);

default:

printf("Not available\n");

break;

}

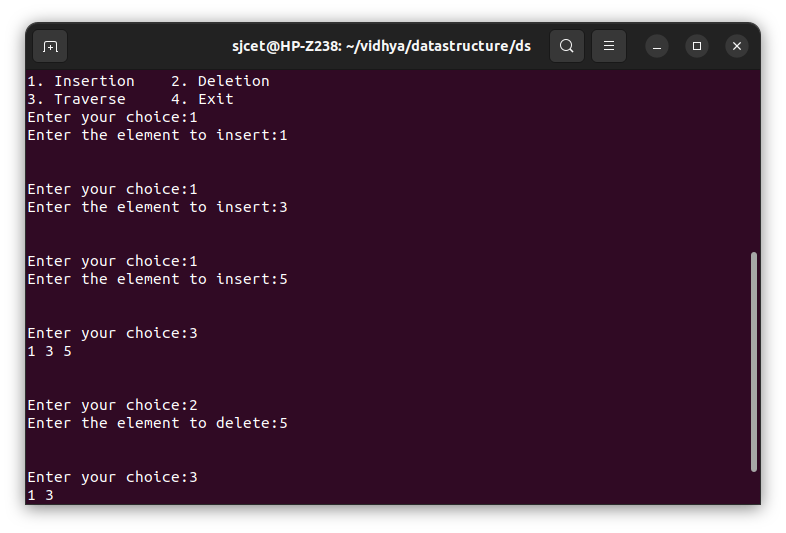
printf("\n");

}

return 0;

}

**OUTPUT**

****

**PRIM’S ALGORITHM**

#include <limits.h>

#include <stdbool.h>

#include <stdio.h>

#define V 5

int minKey(int key[], bool mstSet[])

{

int min = INT\_MAX, min\_index;

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

if (mstSet[v] == false && key[v] < min)

min = key[v], min\_index = v;

return min\_index;

}

int printMST(int parent[], int graph[V][V])

{

printf("Edge \tWeight\n");

for (int i = 1; i < V; i++)

printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);

}

void primMST(int graph[V][V])

{

int parent[V];

int key[V];

bool mstSet[V];

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

key[i] = INT\_MAX, mstSet[i] = false;

key[0] = 0;

parent[0] = -1;

for (int count = 0; count < V - 1; count++) {

int u = minKey(key, mstSet);

mstSet[u] = true;

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

if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])

parent[v] = u, key[v] = graph[u][v];

}

printMST(parent, graph);

}

int main()

{

int graph[V][V] = { { 0, 2, 0, 6, 0 },

{ 2, 0, 3, 8, 5 },

{ 0, 3, 0, 0, 7 },

{ 6, 8, 0, 0, 9 },

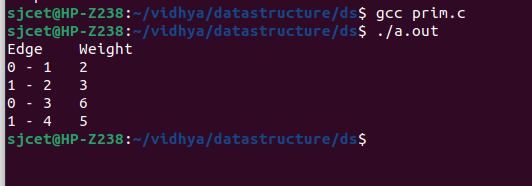
{ 0, 5, 7, 9, 0 } };

primMST(graph);

return 0;

}

**OUTPUT**

****