DEPARTMENT OF COMPUTER APPLICATION

TKM COLLEGE OF ENGINEERING

KOLLAM – 691005

20MCA135 - DATA STRUCTURES LAB

PRACTICAL RECORD BOOK

First Semester MCA

2020-2021

Submitted by:

NAME : ARJUN V PANKAJAKSHAN

ROLL NO : MCA138

DEPARTMENT OF COMPUTER APPLICATION

TKM COLLEGE OF ENGINEERING

KOLLAM – 691005

Certificate

This is a bonafide record of the work done by ARJUN V PANKAJAKSHAN in the First Semester in Data Structures Lab Course(20MCA135) towards the partial fulfillment of the degree of Master of Computer Applications during the academic year 2020-2021.

Staff Member in-charge   Examiner

……………………….   ………………………

INDEX

PROGRAM 1 : MERGE TWO SORTED ARRAYS

AIM : Write a program to merge two sorted arrays

ALGORITHM :

ENTER (a[10],n):

1. Repeat step 2 for i = 0 to (n-1)
2. Input a[i]
3. Return

DISPLAY(c[20],p):

1. Repeat step 2 for k = 0 to p-1
2. Print c[k]
3. Return

MAIN( ):

1. Start
2. Input no. of elements in 1st & 2nd array as „n‟ & „m‟
3. Enter (a.n)
4. Enter (b,m)
5. i = j = k = 0
6. Repeat step 7 to 12 while ((i < n)&&(j < m))
7. If (a[i] >= b[j]),goto step 9
8. c[k+1] = a[i+1]
9. If a[i] = b[j] ,goto step 11
10. c[k++] = b[j++] goto step 7
11. c[k++] = a[i++] & j++
13. Repeat step 14 while (i<n)
14. c[k++] = a[i++]
15. Repeat step 16 while m > j

16. c[k++] = b[j++]

17. Display merged arrays as display(c;k) and exit.

PROGRAM CODE :

1.c	

```
#include<stdio.h>
void main()
{
int n, m, i, j, k, c[40], a[20], b[20];
printf ("Enter limit for A:");
scanf ("%d", &n);
printf ("\nEnter limit for B:");
scanf ("%d", &m);
printf ("Enter elements for A in sorted order:-\n");
for (i = 0; i < n; i++)
    {
scanf ("%d", &a[i]);
    }
printf ("Enter elements for B in sorted order:-\n");
for (j = 0; j < m; j++)
    {
scanf ("%d", &b[j]);
    }
i = j = k = 0;
while (i < n && j < m)
    {
if (a[i] < b[j])
   {
   c[k++] = a[i++];
   }
else if (a[i] > b[j])
          {
           c[k++] = b[j++];
          }
else
          {
    c[k++] = b[j++];
            i++;
            j++;
    }}
if (i < n)
{
for (int t = 0; t < n; t++)
{
c[k++] = a[i++];
}}
if (j < m)
```

```
    {
    for (int t = 0; t < m; t++)
      {
          c[k++] = b[j++];
      }}
printf ("\n");
for (k = 0; k < (m + n); k++)
  {
printf ("\t \n %d ", c[k]);
  }
printf("\n");
}
```

RESULT : The above program is successfully executed and obtained the output

OUTPUT :

```
rony@rony-HP-Laptop-14s-cr2xxx:~/Documents/DS lab$ cd "/home/rony/Documents/DS
 lab/" && gcc array_merge.c -o array_merge && "/home/rony/Documents/DS lab/"ar
ray_merge
Enter limit for A:3

Enter limit for B:2
Enter elements for A in sorted order:-
2
6
9
Enter elements for B in sorted order:-
3
5


 2
 3
 5
 6
 9
```

PROGRAM 2 : CIRCULAR QUEUE

AIM : Write a program to implement a circular queue and perform add,delete and search operation.

ALGORITHM :

ENQUEUE OPERATION:

1. check if the queue is full
2. for the first element, set value of FRONT to 0
3. circularly increase the REAR index by 1 (i.e. if the rear reaches the end, next it would be at the start of the queue)
4. add the new element in the position pointed to by REAR

DEQUEUE OPERATION:

1) check if the queue is empty
2) return the value pointed by FRONT
3) circularly increase the FRONT index by 1
4) for the last element, reset the values of FRONT and REAR to -1

PROGRAM CODE :

2.c	`#include <stdio.h>`
	`#define SIZE 5`
	`int items[SIZE];int front = -1, rear = -1;`
	`// Check if the queue is fullint isFull() {`

```
  if ((front == rear + 1) || (front == 0 && rear == SIZE - 1)) return 1;
  return 0;
}
// Check if the queue is emptyint isEmpty() {
  if (front == -1) return 1;
  return 0;
}
// Adding an elementvoid enQueue(int element) {
  if (isFull())
    printf("\n Queue is full!! \n");
  else {
    if (front == -1) front = 0;
    rear = (rear + 1) % SIZE;
    items[rear] = element;
    printf("\n Inserted -> %d", element);
  }}
// Removing an elementint
 deQueue() {
  int element;
  if (isEmpty()) {
    printf("\n Queue is empty !! \n");
    return (-1);
  } else {
    element = items[front];
    if (front == rear) {
      front = -1;
      rear = -1;
    }
    // Q has only one element, so we reset the
    // queue after dequeing it. ?
```

```
    else {
      front = (front + 1) % SIZE;
    }
    printf("\n Deleted element -> %d \n", element);
    return (element);
  }
}
// Display the queuevoid display() {
  int i;
  if (isEmpty())
    printf(" \n Empty Queue\n");
  else {
    printf("\n Front -> %d ", front);
    printf("\n Items -> ");
    for (i = front; i != rear; i = (i + 1) % SIZE) {
      printf("%d ", items[i]);
    }
    printf("%d ", items[i]);
    printf("\n Rear -> %d \n", rear);
  }}
int main() {
  // Fails because front = -1
  deQueue();
  enQueue(1);
  enQueue(2);
  enQueue(3);
  enQueue(4);
  enQueue(5);
  // Fails to enqueue because front == 0 && rear == SIZE - 1
  enQueue(6);
```

```
    display();
    deQueue();
    display();
    enQueue(7);
    display();
    // Fails to enqueue because front == rear + 1
    enQueue(8);
    return 0;
}
```

RESULT : The above program is successfully executed and obtained the output

OUTPUT :

```
rony@rony-HP-Laptop-14s-cr2xxx:~/Documents/Data-structures/doubly linked list$
ts/DS lab/" && gcc Circular_queue.c -o Circular_queue && "/home/rony/Documents/

 Queue is empty !!

 Inserted -> 1
 Inserted -> 2
 Inserted -> 3
 Inserted -> 4
 Inserted -> 5
 Queue is full!!

 Front -> 0
 Items -> 1 2 3 4 5
 Rear -> 4

 Deleted element -> 1

 Front -> 1
 Items -> 2 3 4 5
 Rear -> 4

 Inserted -> 7
 Front -> 1
 Items -> 2 3 4 5 7
 Rear -> 0

 Queue is full!!
```

PROGRAM 3 : SINGLY LINKED STACK

AIM : Write a program to implement a stack using linked list and perform push, pop and linear search.

ALGORITHM :

PUSH( ):

1. t = newnode( )
2. Enter info to be inserted
3. Read n
4. t=>info = n
5. t=>next = top
6. top = t
7. Return

POP( ):

1. If (top = NULL)
   Print " underflow"
   Return
2. x = top
3. top = top next
4. delnode(x)
5. Return

PROGRAM CODE :

stack.c

```
#include <stdio.h>
#include <stdlib.h>
void push();
void pop();
void display();
struct node
{
int val;
```

```
struct node *next;
};
struct node *head;

void main ()
{
   int choice=0;
   while(choice != 4)
   {
      printf("\n\nChose one from the below options...\n");
      printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
      printf("\n Enter your choice = ");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         {
            push();
            break;
         }
         case 2:
         {
            pop();
            break;
         }
         case 3:
         {
            display();
            break;
         }
         case 4:
         {
            printf("Exiting....");
            break;
         }
         default:
         {
            printf("Please Enter valid choice ");
         } };
}}
void push ()
{
   int val;
   struct node *ptr = (struct node*)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
      printf("not able to push the element");
   }
   else
   {
      printf("Enter the value = ");
      scanf("%d",&val);
      if(head==NULL)
      {
```

```
        ptr->val = val;
        ptr -> next = NULL;
        head=ptr;
    }
    else
    {
        ptr->val = val;
        ptr->next = head;
        head=ptr;
    }
    printf("Item pushed");
    }}
void pop()
{
    int item;
    struct node *ptr;
    if (head == NULL)
    {
        printf("Underflow");
    }
    else
    {
        item = head->val;
        ptr = head;
        head = head->next;
        free(ptr);
        printf("Item popped");
    } }
void display()
{
    int i;
    struct node *ptr;
    ptr=head;
    if(ptr == NULL)
    {
        printf("Stack is empty\n");
    }
    else
    {
        printf("Printing Stack elements \n");
        while(ptr!=NULL)
        {
            printf("%d\n",ptr->val);
            ptr = ptr->next;
        }}}
```

RESULT : The above program is successfully executed and obtained the output

OUTPUT :

```
rony@rony-HP-Laptop-14s-cr2xxx:~/Documents/DS lab$ cd "/home/rony/Documents/Data-struct
programs/" && gcc stack.c -o stack && "/home/rony/Documents/Data-structures/stack c pro


Chose one from the below options...

1.Push
2.Pop
3.Show
4.Exit
 Enter your choice = 1
Enter the value = 77
Item pushed

Chose one from the below options...

1.Push
2.Pop
3.Show
4.Exit
 Enter your choice = 1
Enter the value = 88
Item pushed

Chose one from the below options...

1.Push
2.Pop
3.Show
4.Exit
 Enter your choice = 1
Enter the value = 92
Item pushed

Chose one from the below options...

1.Push
2.Pop
3.Show
4.Exit
 Enter your choice = 2
Item popped

Chose one from the below options...

1.Push
2.Pop
3.Show
4.Exit
 Enter your choice = 3
Printing Stack elements
88
77
```

PROGRAM 4 : DOUBLY LINKED LIST

AIM: Write a program to implement a doubly linked list and perform insertion,deletion and search opeartion

ALGORITHM :

INSERTION

BEGIN:

1. If start = NULL

   start = t

2. else

   t=>next = NULL

   t=>next   prev = t

   start = t

   Return

MIDDLE:

1. Print “ enter info of the node after which you want to insert”
2. Read x
3. p = start
4. Repeat while p< > NULL

   If (p=>info = n)

     t=>next = p=> next

     p=>next = t

     t=>prev = p

     p =>next=>prev = t

     Return

   Else

     p = p=>next

5. Print x not found

t=>next = NULL

p=>next = t

DELETION :

BEGIN

1. p = start
2. p=>next=>prev = NULL
3. start = p=>next
4. start = p=>next
5. delnode(p)
6. Return

MIDDLE

1. Enter “info of the node to be deleted”
2. Read x
3. p = start
4. Repeat until p< > NULL

If(p=>info = x)

p=>prev=>next = p=>next

p=>next=>prev = p=>prev

delnode(p)

Return

Else

p = p=>next

5. Print “x not found”

LAST

1. P = start
2. Repeat while p< > NULL

If(p=>next = NULL)

Delnode(p)

3. Return

DISPLAY:

1. p = start
2. Repeat while p < > NULL

Print p=>info

P = p =>next

PROGRAM CODE :

4.c

```
#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("\n*********Main Menu*********\n");
      printf("\nChoose one option from the following list ...\n");

printf("\n==================================================\n");
      printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random
location\n4.Delete from Beginning\n 5.Delete from last\n6.Delete the node after the
given data\n7.Search\n8.Show\n9.Exit\n");
      printf("\nEnter your choice? = ");
      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;
    } }
void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
```

```
        printf("\nEmpty List\n");
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        while (ptr!=NULL)
        {
            if(ptr->data == item)
            {
                printf("\nitem found at location %d ",i+1);
                flag=0;
                break;
            }
            else
            {
                flag=1;
            }
            i++;
            ptr = ptr -> next;
        }
        if(flag==1)
        {
            printf("\nItem not found\n");
  }
  }
}
```

RESULT : The above program is successfully executed and obtained the output

OUTPUT :

```
rony@rony-HP-Laptop-14s-cr2xxx:~$ cd "/home/rony/Documents/Data-structures/doubly linked list/" && gcc doublylinkedlist.c -o doublylinkedlist && "/home/rony/Documents/Data-stru

*********Main Menu*********
Choose one option from the following list
1.Insert in begining 2.Insert at last 3.Insert at any random location 4.Delete from Beginning 5.Delete from last 6.Delete the node after the given data 7.Search 8.Show 9.Exit
Enter your choice? = 1
Enter Item value = 2
Node inserted
*********Main Menu*********
Choose one option from the following list
1.Insert in begining 2.Insert at last 3.Insert at any random location 4.Delete from Beginning 5.Delete from last 6.Delete the node after the given data 7.Search 8.Show 9.Exit
Enter your choice? = 2
 Enter value = 3
node inserted
*********Main Menu*********
Choose one option from the following list
1.Insert in begining 2.Insert at last 3.Insert at any random location 4.Delete from Beginning 5.Delete from last 6.Delete the node after the given data 7.Search 8.Show 9.Exit
Enter your choice? = 3
Enter the location = 1
Enter value = 4

node inserted

*********Main Menu*********
Choose one option from the following list
1.Insert in begining 2.Insert at last 3.Insert at any random location 4.Delete from Beginning 5.Delete from last 6.Delete the node after the given data 7.Search 8.Show 9.Exit
Enter your choice? = 3
Enter the location = 2
Enter value = 7

node inserted

*********Main Menu*********
Choose one option from the following list
1.Insert in begining 2.Insert at last 3.Insert at any random location 4.Delete from Beginning 5.Delete from last 6.Delete the node after the given data 7.Search 8.Show 9.Exit
Enter your choice? = 8

 printing values...2
3
4
7

*********Main Menu*********
Choose one option from the following list
1.Insert in begining 2.Insert at last 3.Insert at any random location 4.Delete from Beginning 5.Delete from last 6.Delete the node after the given data 7.Search 8.Show 9.Exit
Enter your choice? = 6

 Enter the data after which the node is to be deleted : 3

node deleted
*********Main Menu*********
Choose one option from the following list
1.Insert in begining 2.Insert at last 3.Insert at any random location 4.Delete from Beginning 5.Delete from last 6.Delete the node after the given data 7.Search 8.Show 9.Exit
Enter your choice? = 4

node deleted
*********Main Menu*********
Choose one option from the following list
1.Insert in begining 2.Insert at last 3.Insert at any random location 4.Delete from Beginning 5.Delete from last 6.Delete the node after the given data 7.Search 8.Show 9.Exit
Enter your choice? = 8

 printing values...3
7

*********Main Menu*********
Choose one option from the following list
1.Insert in begining 2.Insert at last 3.Insert at any random location 4.Delete from Beginning 5.Delete from last 6.Delete the node after the given data 7.Search 8.Show 9.Exit
Enter your choice? = 9
rony@rony-HP-Laptop-14s-cr2xxx:~/Documents/Data-structures/doubly linked list$
```

PROGRAM 5 : BINARY SEARCH TREE

AIM : Write a program to implement a binary search tree and perform insertion, deletion and search operation

ALGORITHM :

INSERTION:

```
1. t = newnode
2. t=>info = n
3. t =>left = t=>right = NULL
4. If (root = NULL)
       root = t
       return
5. ptr = root
6. Repeat step 7 until ptr = NULL
7. If (ptr =>info > n)
           If (ptr =>left = NULL)
               Ptr =>left = t
               Return
           Else
               Ptr = ptr =>left
   Else
           If (ptr     right = NULL)
             Ptr =>right = t
             Return
           Else
             Ptr = ptr => right
```

DELETION:

```
1. If (root = NULL)
       Print “Empty tree “
```

Return

2. ptr = root, par = NULL

3. Repeat step 4 & 5 until (ptr =>info = n or ptr = NULL)

4. par = ptr

5. If (ptr=>info > n)

   ptr = ptr=>left

   Else

   Ptr = ptr=>right

6. If ptr = NULL

   print " no. not present"

PROGRAM CODE :

5.c	See code below

5.c

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

struct node{
        struct node *left;
        struct node *right;
        int data;
};
struct node *root;

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

struct node* insert(struct node* root,int value) {
  if(root == NULL){
        return newNode(value);
  }
  else if(value == root->data){
        printf("Same data can't be stored");
  }
  else if(value>root->data){
        root->right = insert(root->right,value);
  }
  else if(value<root->data){
```

```
        root->left = insert(root->left,value);
  }
  return root;
}

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

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

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

struct node* search(struct node* root, int key) {
  if (root == NULL)
    printf("\nNot FOUND!\n");
  else if (root->data == key)
    printf("\nFOUND!\n");
  else{
        if (root->data < key)
    return search(root->right, key);
  return search(root->left, key);
  }
}

struct node* minValueNode(struct node* node){
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    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->data)
      root->left = deleteNode(root->left, key);
   else if (key > root->data)
      root->right = deleteNode(root->right, key);
   else {
      // node with only one child or no child
      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;
      }
      // node with two children:
      // Get the inorder successor
      // (smallest in the right subtree)
      struct node* temp = minValueNode(root->right);

      // Copy the inorder
      // successor's content to this node
      root->data = temp->data;

      // Delete the inorder successor
      root->right = deleteNode(root->right, temp->data);
   }
   return root;
}

int main(){
         int opt;
         int value,searchv,key;
         do{
                  printf("\n1)Create Root Node \n2)Insert Node\n3)Search\n");
                  printf("4)inorderTraversal \n5)preorderTraversal
\n6)postorderTraversal \n7)Delete \n8)Quiet \n");
                  printf("Choose Option :: ");
                  scanf("%d",&opt);
                  switch(opt){
                           case 1:
                                    printf("\nEnter a number : ");
                                    scanf("%d",&value);
                                    root = newNode(value);
```

```
                        break;
                case 2:
                        printf("\nEnter a number : ");
                        scanf("%d",&value);
                        root = insert(root,value);
                        break;
                case 3:
                        printf("\nEnter a number : ");
                        scanf("%d",&searchv);
                        search(root,searchv);
                        break;
                case 4:
                        printf("\n.................................\n");
                        inorderTraversal(root);
                        printf("\n.................................\n");
                        break;
                case 5:
                        printf("\n.................................\n");
                        preorderTraversal(root);
                        printf("\n.................................\n");
                        break;
                case 6:
                        printf("\n.................................\n");
                        postorderTraversal(root);
                        printf("\n.................................\n");
                        break;
                case 7:
                        printf("\nEnter a number to be deleted : ");
                        scanf("%d",&key);
                        deleteNode(root,key);
                        break;
                defualt:
                        printf("Invalid option!");
            }
        }while(opt!=8);
        return 0;
}
```

RESULT : The above program is successfully executed and obtained the output

OUTPUT:

```
rony@rony-HP-Laptop-14s-cr2xxx:~/Documents/DS-lab$ ./a.out

1)Create Root Node
2)Insert Node
3)Search
4)inorderTraversal
5)preorderTraversal
6)postorderTraversal
7)Delete
8)Quiet
Choose Option :: 1

Enter a number : 5

1)Create Root Node
2)Insert Node
3)Search
4)inorderTraversal
5)preorderTraversal
6)postorderTraversal
7)Delete
8)Quiet
Choose Option :: 2

Enter a number : 8

1)Create Root Node
2)Insert Node
3)Search
4)inorderTraversal
5)preorderTraversal
6)postorderTraversal
7)Delete
8)Quiet
Choose Option :: 2

Enter a number : 3

1)Create Root Node
2)Insert Node
3)Search
4)inorderTraversal
5)preorderTraversal
6)postorderTraversal
7)Delete
8)Quiet
Choose Option :: 5

.................................
5 ->3 ->8 ->
.................................
```

```
1)Create Root Node
2)Insert Node
3)Search
4)inorderTraversal
5)preorderTraversal
6)postorderTraversal
7)Delete
8)Quiet
Choose Option :: 4

.................................
3 ->5 ->8 ->
.................................

1)Create Root Node
2)Insert Node
3)Search
4)inorderTraversal
5)preorderTraversal
6)postorderTraversal
7)Delete
8)Quiet
Choose Option :: 3

Enter a number : 5

FOUND!

1)Create Root Node
2)Insert Node
3)Search
4)inorderTraversal
5)preorderTraversal
6)postorderTraversal
7)Delete
8)Quiet
Choose Option :: 7

Enter a number to be deleted : 8
```

```
1)Create Root Node
2)Insert Node
3)Search
4)inorderTraversal
5)preorderTraversal
6)postorderTraversal
7)Delete
8)Quiet
Choose Option :: 4

.................................
3 ->5 ->
.................................

1)Create Root Node
2)Insert Node
3)Search
4)inorderTraversal
5)preorderTraversal
6)postorderTraversal
7)Delete
8)Quiet
Choose Option :: 8
rony@rony-HP-Laptop-14s-cr2xxx:~/Documents/DS-lab$
```

PROGRAM 6 : BIT STRING

AIM : Write a program to implement set data structure and set operations (Union,Intersection and Difference) using bit string.

ALGORITHM :

/* Operations covered :

1) Create() : for creating a new set with initial members of the set
2) print() : diaplays all members of the set
3) Union() : finds union of two sets, set1[] and set2[] and stores the result in set3[]
4) intersection() : finds intersection of two sets, set1[] and set2[] and stores the result in set3[]
5) difference() :finds difference of two sets, set1[] and set2[] and stores the result in set3[]
6) member() :function returns 1 or 0, depending onwhether the element x belongs or not to a set.
7) symmdiff() : Finds Symmetric difference of two sets

PROGRAM CODE :

6.c

```
#define MAX 30
#include<stdio.h>
#include<stdlib.h>
void create(int set[]);
void print(int set[]);
void Union(int set1[],int set2[],int set3[]);
void intersection(int set1[],int set2[],int set3[]);
void difference(int set1[],int set2[],int set3[]);
void symmdiff(int set1[],int set2[],int set3[]);
int member(int set[],int x);
void main()
{ int set1[MAX],set2[MAX],set3[MAX];
int x,op;
set1[0]=set2[0]=set3[0]=0;
```

```
do{
printf("\n1)Create 2)Print 3)Union 4)Intersection 5)Difference 6)Symmetric Difference
7)Quit");
printf("\nEnter Your Choice:");
scanf("%d",&op);
switch(op){
case 1: printf("---Creating First Set---");
create(set1);
printf("---Creating Second Set---");
create(set2);
break;
case 2: printf("First Set : ");
print(set1);
printf("Second Set : ");
print(set2);
break;
case 3: Union(set1,set2,set3);print(set3);break;
case 4: intersection(set1,set2,set3);print(set3);break;
case 5: difference(set1,set2,set3);print(set3);break;
case 6: symmdiff(set1,set2,set3);print(set3);break;}
printf("\nPRESS ANY KEY");
}while(op!=7);}
void create(int set[])
{ int n,i,x;
set[0]=0;/*make it a null set*/
printf("\n No. of elements in the set:");
scanf("%d",&n);
printf("enter set elements :");
for(i=1;i<=n;i++)
scanf("%d",&set[i]);
set[0]=n; }
void print(int set[])
{ int i,n;
n=set[0];/* number of elements in the set */
```

```
printf("\n Members of the set :-->");
for(i=1;i<=n;i++)
printf("%d ",set[i]);
}
/* union of set1[] and set2[] is stored in set3[]*/
void Union(int set1[],int set2[],int set3[])
{ int i,n;
/* copy set1[] to set3[]*/
set3[0]=0;/*make set3[] a null set */
n=set1[0];/* number of elements in the set*/
//Union of set1,set2= set1 + (set2-set1)
for(i=0;i<=n;i++)
set3[i]=set1[i];
n=set2[0];
for(i=1;i<=n;i++)
if(!member(set3,set2[i]))
set3[++set3[0]]=set2[i]; // insert and increment no. of elements
}
/*function returns 1 or 0 depending on whether x belongs
to set[] or not */
int member(int set[],int x)
{ int i,n;
n=set[0]; /* number of elements in the set*/
for(i=1;i<=n;i++)
if(x==set[i])
return(1);
return(0);
}
void intersection(int set1[],int set2[],int set3[])
{
int i,n;
set3[0]=0; /* make a NULL set*/
n=set1[0];/* number of elements in the set*/
for(i=1;i<=n;i++)
```

```
if(member(set2,set1[i])) /* all common elements are inserted in set3[]*/
set3[++set3[0]]=set1[i]; // insert and increment no. of elements
}
/*difference of set1[] and set2[] is stored in set3[]*/
void difference(int set1[],int set2[],int set3[])
{ int i,n;
n=set1[0];/* number of elements in the set*/
set3[0]=0;/*make it a null set*/
for(i=1;i<=n;i++)
if(!member(set2,set1[i]))
set3[++set3[0]]=set1[i]; // insert and increment no. of elements
}
void symmdiff(int set1[],int set2[],int set3[])
{ int i,n;
n=set1[0];/* number of elements in the set*/
set3[0]=0;/*make it a null set*/
//Calculate set1-set2
for(i=1;i<=n;i++)
if(!member(set2,set1[i]))
set3[++set3[0]]=set1[i]; // insert and increment no. of elements
//Calculate set2-set1
n=set2[0];
for(i=1;i<=n;i++)
if(!member(set1,set2[i]))
set3[++set3[0]]=set2[i]; // insert and increment no. of elements
}
```

RESULT : The above program is successfully executed and obtained the output

OUTPUT :

```
rony@rony-HP-Laptop-14s-cr2xxx:~/Documents/DS lab$ cd "/home/rony/Documents/DS lab/" &&

1)Create 2)Print 3)Union 4)Intersection 5)Difference 6)Symmetric Difference 7)Quit
Enter Your Choice:1
---Creating First Set---
 No. of elements in the set:3
enter set elements :1
7
9
---Creating Second Set---
 No. of elements in the set:4
enter set elements :1
9
7
3

PRESS ANY KEY
1)Create 2)Print 3)Union 4)Intersection 5)Difference 6)Symmetric Difference 7)Quit
Enter Your Choice:2
First Set :
 Members of the set :-->1 7 9 Second Set :
 Members of the set :-->1 9 7 3
PRESS ANY KEY
1)Create 2)Print 3)Union 4)Intersection 5)Difference 6)Symmetric Difference 7)Quit
Enter Your Choice:3

 Members of the set :-->1 7 9 3
PRESS ANY KEY
1)Create 2)Print 3)Union 4)Intersection 5)Difference 6)Symmetric Difference 7)Quit
Enter Your Choice:4

 Members of the set :-->1 7 9
PRESS ANY KEY
1)Create 2)Print 3)Union 4)Intersection 5)Difference 6)Symmetric Difference 7)Quit
Enter Your Choice:5

 Members of the set :-->
PRESS ANY KEY
1)Create 2)Print 3)Union 4)Intersection 5)Difference 6)Symmetric Difference 7)Quit
Enter Your Choice:6

 Members of the set :-->3
PRESS ANY KEY
1)Create 2)Print 3)Union 4)Intersection 5)Difference 6)Symmetric Difference 7)Quit
Enter Your Choice:7
```

PROGRAM 7 : DISJOINT SETS

AIM : Write a program to implement disjoint sets and the associated operations (create, union, find).

ALGORITHM :

1. make_set(v) - creates a new set consisting of the new element v
2. union_sets(a, b) - merges the two specified sets (the set in which the element a is located, and the set in which the element b is located)
3. find_set(v) - returns the representative (also called leader) of the set that contains the element v. This representative is an element of its corresponding set. It is selected in each set by the data structure itself (and can change over time, namely after union_sets calls). This representative can be used to check if two elements are part of the same set or not. a and b are exactly in the same set, if find_set(a) == find_set(b). Otherwise they are in different sets.

PROGRAM CODE :

7.c

```
#include<stdio.h>
#include<stdlib.h>
struct node{
   struct node *rep;
   struct node *next;
   int data;
}*heads[50],*tails[50];
static int countRoot=0;
void makeSet(int x){
        struct node *new=(struct node *)malloc(sizeof(struct node));
        new->rep=new;
        new->next=NULL;
        new->data=x;
        heads[countRoot]=new;
        tails[countRoot++]=new;
}
struct node* find(int a){
        int i;
        struct node *tmp=(struct node *)malloc(sizeof(struct node));
        for(i=0;i<countRoot;i++){
                tmp=heads[i];
                while(tmp!=NULL){
                if(tmp->data==a)
```

```
                    return tmp->rep;
                    tmp=tmp->next;
                    }
          }
          return NULL;
}
void unionSets(int a,int b){
          int i,pos,flag=0,j;
          struct node *tail2=(struct node *)malloc(sizeof(struct node));
          struct node *rep1=find(a);
          struct node *rep2=find(b);
          if(rep1==NULL||rep2==NULL){
                    printf("Element not present in the DS");
                    return;
          }
          if(rep1!=rep2){
                    for(j=0;j<countRoot;j++){
                              if(heads[j]==rep2){
                                        pos=j;
                                        flag=1;
                                        countRoot-=1;
                                        tail2=tails[j];
                                        for(i=pos;i<countRoot;i++){
                                                  heads[i]=heads[i+1];
                                                  tails[i]=tails[i+1];
                                        }}
                              if(flag==1)
                                        break;
                    }
                    for(j=0;j<countRoot;j++){
                              if(heads[j]==rep1){
                                        tails[j]->next=rep2;
                                        tails[j]=tail2;
                                        break;
                              }}
                    while(rep2!=NULL){
                    rep2->rep=rep1;
                    rep2=rep2->next;
                    }}}
int search(int x){
          int i;
          struct node *tmp=(struct node *)malloc(sizeof(struct node));
          for(i=0;i<countRoot;i++){
                    tmp=heads[i];
                    if(heads[i]->data==x)
                              return 1;
                    while(tmp!=NULL){
                              if(tmp->data==x)
                                        return 1;
                              tmp=tmp->next;
```

```
            }}
        return 0;
}
void main(){
int choice,x,i,j,y,flag=0;
        do{
            printf("\n.......MENU.......1.Make Set......2.Display set
representatives.....3.Union.....4.Find Set....5.Exit....");
            printf("\nEnter your choice :  ");
            scanf("%d",&choice);
            switch(choice){
            case 1:
                printf("Enter new element : ");
                scanf("%d",&x);
                if(search(x)==1)
                printf("Element already present in the disjoint set DS");
                else
                    makeSet(x);
                break;
            case 2:
                for(i=0;i<countRoot;i++)
                    printf("%d ",heads[i]->data);
                break;
            case 3:
                printf("Enter first element : ");
                scanf("%d",&x);
                printf("Enter second element : ");
                scanf("%d",&y);
                unionSets(x,y);
                break;
            case 4:
                printf("Enter the element");
                scanf("%d",&x);
                struct node *rep=(struct node *)malloc(sizeof(struct node));
                rep=find(x);
                if(rep==NULL)
                printf("\nElement not present in the DS");
                else
                printf("\nThe representative of %d is %d",x,rep->data);
                break;
            case 5:
                exit(0);
            default:
                printf("\nWrong choice");
                break;
            }}
        while(1);
    };
```

RESULT : The above program is successfully executed and obtained the output

OUTPUT :

```
rony@rony-HP-Laptop-14s-cr2xxx:~/Documents/DS-lab$ cd "/home/rony/Documents/DS-lab/" && gcc disjoint_set.c -o disjoint_set &
oint_set

.......MENU.......1.Make Set......2.Display set representatives.....3.Union.....4.Find Set....5.Exit....
Enter your choice :  1
Enter new element : 5

.......MENU.......1.Make Set......2.Display set representatives.....3.Union.....4.Find Set....5.Exit....
Enter your choice :  1
Enter new element : 15

.......MENU.......1.Make Set......2.Display set representatives.....3.Union.....4.Find Set....5.Exit....
Enter your choice :  1
Enter new element : 30

.......MENU.......1.Make Set......2.Display set representatives.....3.Union.....4.Find Set....5.Exit....
Enter your choice :  2
5 15 30
.......MENU.......1.Make Set......2.Display set representatives.....3.Union.....4.Find Set....5.Exit....
Enter your choice :  4
Enter the element15

The representative of 15 is 15
.......MENU.......1.Make Set......2.Display set representatives.....3.Union.....4.Find Set....5.Exit....
Enter your choice :  5
```

PROGRAM 8 : BINOMIAL HEAP

AIM : Write a program to implement a binomial heaps and operations (create, insert, delete, extract-min, decrease key)

ALGORITHM :

CREATE:

To make an empty binomial heap, the MAKE-BINOMIAL-HEAP procedure simply allocates and returns an object H, where head[H] = NIL.

BINOMIAL-HEAP-MINIMUM(H) :

1. y ← NIL
2. x ← head[H]
3. min ← ∞
4. while x = NIL
5. do if key[x] < min
6. then min ← key[x]
7. y ← x
8. x ← sibling[x]
9. return y

BINOMIAL-HEAP-UNION(H1, H2) :

1. H ← MAKE-BINOMIAL-HEAP()
2. head[H] ← BINOMIAL-HEAP-MERGE(H1, H2)
3. free the objects H1 and H2 but not the lists they point to
4. if head[H] = NIL
5. then return H
6. prev-x ← NIL
7. x ← head[H]
8. next-x ← sibling[x]

9. while next-x = NIL

10. do if (degree[x] = degree[next-x]) or (sibling[next-x] = NIL and degree[sibling[next-x]] = degree[x])

11 then prev-x ← x Cases 1 and 2

12. x ← next-x Cases 1 and 2

13. else if key[x] ≤ key[next-x]

14. then sibling[x] ← sibling[next-x] Case 3

15. BINOMIAL-LINK(next-x, x) Case 3

16. else if prev-x = NIL Case 4

17. then head[H] ← next-x Case 4

18. else sibling[prev-x] ← next-x Case 4

19. BINOMIAL-LINK(x, next-x) Case 4

20. x ← next-x Case 4

21. next-x ← sibling[x]

22. return H

BINOMIAL-HEAP-INSERT(H, x):

1. H ← MAKE-BINOMIAL-HEAP()

2. p[x] ← NIL

3. child[x] ← NIL

4. sibling[x] ← NIL

5. degree[x] ← 0

6. head[H] ← x

7. H ← BINOMIAL-HEAP-UNION(H, H)

BINOMIAL-HEAP-DECREASE-KEY(H, x, k):

1. if k > key[x]

2. then error “new key is greater than current key”

3. key[x] ← k

4. y ← x

5. z ← p[y]

6. while z = NIL and key[y] < key[z]

7. do exchange key[y] ↔ key[z]

8. If y and z have satellite fields, exchange them, too.

9. y ← z

10. z ← p[y]

BINOMIAL-HEAP-DELETE(H, x):

1. BINOMIAL-HEAP-DECREASE-KEY(H, x, −∞)

2. BINOMIAL-HEAP-EXTRACT-MIN(H)

PROGRAM CODE :

8.c

```
#include<stdio.h>
#include<stdlib.h>
struct node {
 int n;
 int degree;
 struct node* parent;
 struct node* child;
 struct node* sibling;
};
struct node* MAKE_bin_HEAP();
int bin_LINK(struct node*, struct node*);
struct node* CREATE_NODE(int);
struct node* bin_HEAP_UNION(struct node*, struct node*);
struct node* bin_HEAP_INSERT(struct node*, struct node*);
struct node* bin_HEAP_MERGE(struct node*, struct node*);
struct node* bin_HEAP_EXTRACT_MIN(struct node*);
int REVERT_LIST(struct node*);
int DISPLAY(struct node*);
struct node* FIND_NODE(struct node*, int);
int bin_HEAP_DECREASE_KEY(struct node*, int, int);
int bin_HEAP_DELETE(struct node*, int);
int count = 1;
struct node* MAKE_bin_HEAP() {
 struct node* np;
 np = NULL;
 return np;
}
struct node * H = NULL;
struct node *Hr = NULL;
int bin_LINK(struct node* y, struct node* z) {
 y->parent = z;
 y->sibling = z->child;
 z->child = y;
 z->degree = z->degree + 1;
}
struct node* CREATE_NODE(int k) {
 struct node* p;//new node;
 p = (struct node*) malloc(sizeof(struct node));
 p->n = k;
```

```
return p;
}
struct node* bin_HEAP_UNION(struct node* H1, struct node* H2) {
struct node* prev_x;
struct node* next_x;
struct node* x;
struct node* H = MAKE_bin_HEAP();
H = bin_HEAP_MERGE(H1, H2);
if (H == NULL)
return H;
prev_x = NULL;
x = H;
next_x = x->sibling;
while (next_x != NULL) {
if ((x->degree != next_x->degree) || ((next_x->sibling != NULL)
&& (next_x->sibling)->degree == x->degree)) {
prev_x = x;
x = next_x;
} else {
if (x->n <= next_x->n) {
x->sibling = next_x->sibling;
bin_LINK(next_x, x);
} else {
if (prev_x == NULL)
H = next_x;
else
prev_x->sibling = next_x;
bin_LINK(x, next_x);
x = next_x;
}
}
next_x = x->sibling;
}
return H;
}
struct node* bin_HEAP_INSERT(struct node* H, struct node* x) {
struct node* H1 = MAKE_bin_HEAP();
x->parent = NULL;
x->child = NULL;
x->sibling = NULL;
x->degree = 0;
H1 = x;
H = bin_HEAP_UNION(H, H1);
return H;
}
struct node* bin_HEAP_MERGE(struct node* H1, struct node* H2) {
struct node* H = MAKE_bin_HEAP();
struct node* y;
struct node* z;
struct node* a;
```

```
struct node* b;
y = H1;
z = H2;
if (y != NULL) {
if (z != NULL && y->degree <= z->degree)
H = y;
else if (z != NULL && y->degree > z->degree)
H = z;
else
H = y;
} else
H = z;
while (y != NULL && z != NULL) {
if (y->degree < z->degree) {
y = y->sibling;
} else if (y->degree == z->degree) {
a = y->sibling;
y->sibling = z;
y = a;
} else {
b = z->sibling;
z->sibling = y;
z = b;
}
}
return H;
}
int DISPLAY(struct node* H) {
struct node* p;
if (H == NULL) {
printf("\nHEAP EMPTY");
return 0;
}
printf("\nTHE ROOT NODES ARE:-");
p = H;
while (p != NULL) {
printf("%d", p->n);
if (p->sibling != NULL)
printf("-->");
p = p->sibling;
}
printf("\n");
}
struct node* bin_HEAP_EXTRACT_MIN(struct node* H1) {
int min;
struct node* t = NULL;
struct node* x = H1;
struct node *Hr;
struct node* p;
Hr = NULL;
```

```
 if (x == NULL) {
 printf("\nNOTHING TO EXTRACT");
 return x;
 }
 // int min=x->n;
 p = x;
 while (p->sibling != NULL) {
 if ((p->sibling)->n < min) {
 min = (p->sibling)->n;
 t = p;
 x = p->sibling;
 }
 p = p->sibling;
 }
 if (t == NULL && x->sibling == NULL)
 H1 = NULL;
 else if (t == NULL)
 H1 = x->sibling;
 else if (t->sibling == NULL)
 t = NULL;
 else
 t->sibling = x->sibling;
 if (x->child != NULL) {
 REVERT_LIST(x->child);
 (x->child)->sibling = NULL;
 }
 H = bin_HEAP_UNION(H1, Hr);
 return x;
}
int REVERT_LIST(struct node* y) {
 if (y->sibling != NULL) {
 REVERT_LIST(y->sibling);
 (y->sibling)->sibling = y;
 } else {
 Hr = y;
 }
}
struct node* FIND_NODE(struct node* H, int k) {
 struct node* x = H;
 struct node* p = NULL;
 if (x->n == k) {
 p = x;
 return p;
 }
 if (x->child != NULL && p == NULL) {
 p = FIND_NODE(x->child, k);
 }
 if (x->sibling != NULL && p == NULL) {
 p = FIND_NODE(x->sibling, k);
 }
```

```
 return p;
}
int bin_HEAP_DECREASE_KEY(struct node* H, int i, int k) {
 int temp;
 struct node* p;
 struct node* y;
 struct node* z;
 p = FIND_NODE(H, i);
 if (p == NULL) {
 printf("\nINVALID CHOICE OF KEY TO BE REDUCED");
 return 0;
 }
 if (k > p->n) {
 printf("\nSORY!THE NEW KEY IS GREATER THAN CURRENT ONE");
 return 0;
 }
 p->n = k;
 y = p;
 z = p->parent;
 while (z != NULL && y->n < z->n) {
 temp = y->n;
 y->n = z->n;
 z->n = temp;
 y = z;
 z = z->parent;
 }
 printf("KEY REDUCED SUCCESSFULLY!");
}
int bin_HEAP_DELETE(struct node* H, int k) {
 struct node* np;
 if (H == NULL) {
 printf("\nHEAP EMPTY");
 return 0;
 }
 bin_HEAP_DECREASE_KEY(H, k, -1000);
 np = bin_HEAP_EXTRACT_MIN(H);
 if (np != NULL)
 printf("NODE DELETED SUCCESSFULLY");
}
int main() {
 int i, n, m, l;
 struct node* p;
 struct node* np;
 char ch;
 printf("\nENTER THE NUMBER OF ELEMENTS:");
 scanf("%d", &n);
 printf("\nENTER THE ELEMENTS:\n");
 for (i = 1; i <= n; i++) {
 scanf("%d", &m);
 np = CREATE_NODE(m);
```

```
H = bin_HEAP_INSERT(H, np);
}
DISPLAY(H);
do {
printf("\nMENU:-\n");
printf("1)INSERT AN ELEMENT......2)EXTRACT THE MINIMUM KEY
NODE...3)DECREASE A NODE KEY... 4)DELETE A NODE...5)QUIT\n");
scanf("%d", &l);
switch (l) {
case 1:
do {
printf("ENTER THE ELEMENT TO BE INSERTED:");
scanf("%d", &m);
p = CREATE_NODE(m);
H = bin_HEAP_INSERT(H, p);
printf("NOW THE HEAP IS:");
DISPLAY(H);
printf("INSERT MORE(y/Y)= ");
fflush(stdin);
scanf("%c", &ch);
} while (ch == 'Y' || ch == 'y');
break;
case 2:
do {
printf("EXTRACTING THE MINIMUM KEY NODE");
p = bin_HEAP_EXTRACT_MIN(H);
if (p != NULL)
printf("THE EXTRACTED NODE IS %d", p->n);
printf("NOW THE HEAP IS:");
DISPLAY(H);
printf("EXTRACT MORE(y/Y)");
fflush(stdin);
scanf("%c", &ch);
} while (ch == 'Y' || ch == 'y');
break;
case 3:
do {
printf("ENTER THE KEY OF THE NODE TO BE DECREASED:");
scanf("%d", &m);
printf("ENTER THE NEW KEY : ");
scanf("%d", &l);
bin_HEAP_DECREASE_KEY(H, m, l);
printf("NOW THE HEAP IS:");
DISPLAY(H);
printf("DECREASE MORE(y/Y)");
fflush(stdin);
scanf("%c", &ch);
} while (ch == 'Y' || ch == 'y');
break;
case 4:
```

```
do {
printf("ENTER THE KEY TO BE DELETED: ");
scanf("%d", &m);
bin_HEAP_DELETE(H, m);
printf("DELETE MORE(y/Y)");
fflush(stdin);
scanf("%c", &ch);
} while (ch == 'y' || ch == 'Y');
break;
case 5:
break;
default:
printf("\nINVALID ENTRY...TRY AGAIN....\n");
}
} while (l != 5);
}
```

RESULT : The above program is successfully executed and obtained the output

OUTPUT :

```
rony@rony-HP-Laptop-14s-cr2xxx:~/Documents/DS-lab$ cd "/home/rony/Documents/DS-lab/" && gcc Bheap.c -o Bheap && "/home/rony/Documents/DS-lab/"Bheap

ENTER THE NUMBER OF ELEMENTS:1

ENTER THE ELEMENTS:
6

THE ROOT NODES ARE:-6

MENU:-
1)INSERT AN ELEMENT......2)EXTRACT THE MINIMUM KEY NODE...3)DECREASE A NODE KEY... 4)DELETE A NODE...5)QUIT
1
ENTER THE ELEMENT TO BE INSERTED:2
NOW THE HEAP IS:
THE ROOT NODES ARE:-2
INSERT MORE(y/Y)=
MENU:-
1)INSERT AN ELEMENT......2)EXTRACT THE MINIMUM KEY NODE...3)DECREASE A NODE KEY... 4)DELETE A NODE...5)QUIT
1
ENTER THE ELEMENT TO BE INSERTED:3
NOW THE HEAP IS:
THE ROOT NODES ARE:-3-->2
INSERT MORE(y/Y)=
MENU:-
1)INSERT AN ELEMENT......2)EXTRACT THE MINIMUM KEY NODE...3)DECREASE A NODE KEY... 4)DELETE A NODE...5)QUIT
2
EXTRACTING THE MINIMUM KEY NODETHE EXTRACTED NODE IS 2NOW THE HEAP IS:
THE ROOT NODES ARE:-3
EXTRACT MORE(y/Y)
MENU:-
1)INSERT AN ELEMENT......2)EXTRACT THE MINIMUM KEY NODE...3)DECREASE A NODE KEY... 4)DELETE A NODE...5)QUIT
1
ENTER THE ELEMENT TO BE INSERTED:1
NOW THE HEAP IS:
THE ROOT NODES ARE:-1
INSERT MORE(y/Y)=
MENU:-
1)INSERT AN ELEMENT......2)EXTRACT THE MINIMUM KEY NODE...3)DECREASE A NODE KEY... 4)DELETE A NODE...5)QUIT
3
ENTER THE KEY OF THE NODE TO BE DECREASED:3
ENTER THE NEW KEY : 2
KEY REDUCED SUCCESSFULLY!NOW THE HEAP IS:
THE ROOT NODES ARE:-1
DECREASE MORE(y/Y)
MENU:-
1)INSERT AN ELEMENT......2)EXTRACT THE MINIMUM KEY NODE...3)DECREASE A NODE KEY... 4)DELETE A NODE...5)QUIT
4
ENTER THE KEY TO BE DELETED: 1
KEY REDUCED SUCCESSFULLY!NODE DELETED SUCCESSFULLYDELETE MORE(y/Y)
MENU:-
1)INSERT AN ELEMENT......2)EXTRACT THE MINIMUM KEY NODE...3)DECREASE A NODE KEY... 4)DELETE A NODE...5)QUIT
5
rony@rony-HP-Laptop-14s-cr2xxx:~/Documents/DS-lab$
```

PROGRAM 9: B TREES

AIM : Write a program to implement B trees and its operations.

ALGORITHM :

```
BreeInsertion(T, k)
r root[T]
if n[r] = 2t - 1
        s = AllocateNode()
        root[T] = s
        leaf[s] = FALSE
        n[s] <- 0
        c1[s] <- r
        BtreeSplitChild(s, 1, r)
        BtreeInsertNonFull(s, k)
else BtreeInsertNonFull(r, k)
BtreeInsertNonFull(x, k)
i = n[x]
if leaf[x]
        while i ≥ 1 and k < keyi*x+
                keyi+1 [x] = keyi[x]
                i = i - 1
        keyi+1[x] = k
        n[x] = n[x] + 1
else while i ≥ 1 and k < keyi*x+
 i = i - 1
 i = i + 1
 if n[ci[x]] == 2t - 1
        BtreeSplitChild(x, i, ci[x])
```

```
        if k &rt; keyi[x]
              i = i + 1
 BtreeInsertNonFull(ci[x], k)
BtreeSplitChild(x, i)
BtreeSplitChild(x, i, y)
z = AllocateNode()
leaf[z] = leaf[y]
n[z] = t - 1
for j = 1 to t - 1
        keyj[z] = keyj+t[y]
if not leaf [y]
        for j = 1 to t
           cj[z] = cj + t[y]
n[y] = t - 1
for j = n[x] + 1 to i + 1
    cj+1[x] = cj[x]
ci+1[x] = z
for j = n[x] to i
    keyj+1[x] = keyj[x]
keyi[x] = keyt[y]
n[x] = n[x] + 1
```

PROGRAM CODE :

9.c

```
#include<stdio.h>
#include<stdlib.h>
#define M 5
struct node{
int n; /* n < M No. of keys in node will always less than order of B
tree */
int keys[M-1]; /*array of keys*/
struct node *p[M]; /* (n+1 pointers will be in use) */
}*root=NULL;
enum KeyStatus { Duplicate,SearchFailure,Success,InsertIt,LessKeys };
void insert(int key);
void display(struct node *root,int);
```

```
void DelNode(int x);
void search(int x);
enum KeyStatus ins(struct node *r, int x, int* y, struct node** u);
int searchPos(int x,int *key_arr, int n);
enum KeyStatus del(struct node *r, int x);
int main()
{
int key;
int choice;
printf("Creation of B tree for node %d\n",M);
while(1)
{
printf("1.Insert\n");
printf("2.Delete\n");
printf("3.Search\n");
printf("4.Display\n");
printf("5.Quit\n");
printf("Enter your choice : ");
scanf("%d",&choice);
switch(choice)
{
case 1:
printf("Enter the key : ");
scanf("%d",&key);
insert(key);
break;
case 2:
printf("Enter the key : ");
scanf("%d",&key);
DelNode(key);
break;
case 3:
printf("Enter the key : ");
scanf("%d",&key);
search(key);
break;
case 4:
printf("Btree is :\n");
display(root,0);
break;
case 5:
exit(1);
default:
printf("Wrong choice\n");
break;
}/*End of switch*/
}/*End of while*/
return 0;
}/*End of main()*/
void insert(int key)
```

```
{
struct node *newnode;
int upKey;
enum KeyStatus value;
value = ins(root, key, &upKey, &newnode);
if (value == Duplicate)
printf("Key already available\n");
if (value == InsertIt)
{
struct node *uproot = root;
root=malloc(sizeof(struct node));
root->n = 1;
root->keys[0] = upKey;
root->p[0] = uproot;
root->p[1] = newnode;
}/*End of if */
}/*End of insert()*/
enum KeyStatus ins(struct node *ptr, int key, int *upKey,struct node**newnode)
{
struct node *newPtr, *lastPtr;
int pos, i, n,splitPos;
int newKey, lastKey;
enum KeyStatus value;
if (ptr == NULL)
{
*newnode = NULL;
*upKey = key;
return InsertIt;
}
n = ptr->n;
pos = searchPos(key, ptr->keys, n);
if (pos < n && key == ptr->keys[pos])
return Duplicate;
value = ins(ptr->p[pos], key, &newKey, &newPtr);
if (value != InsertIt)
return value;
/*If keys in node is less than M-1 where M is order of B tree*/
if (n < M - 1)
{
pos = searchPos(newKey, ptr->keys, n);
/*Shifting the key and pointer right for inserting the new key*/
for (i=n; i>pos; i--)
{
ptr->keys[i] = ptr->keys[i-1];
ptr->p[i+1] = ptr->p[i];
}
/*Key is inserted at exact location*/
ptr->keys[pos] = newKey;
ptr->p[pos+1] = newPtr;
++ptr->n; /*incrementing the number of keys in node*/
```

```
return Success;
}
if (pos == M - 1)
{
lastKey = newKey;
lastPtr = newPtr;
}
else /*If keys in node are maximum and position of node to be inserted
is not last*/
{
lastKey = ptr->keys[M-2];
lastPtr = ptr->p[M-1];
for (i=M-2; i>pos; i--)
{
ptr->keys[i] = ptr->keys[i-1];
ptr->p[i+1] = ptr->p[i];
}
ptr->keys[pos] = newKey;
ptr->p[pos+1] = newPtr;
}
splitPos = (M - 1)/2;
(*upKey) = ptr->keys[splitPos];

(*newnode)=malloc(sizeof(struct node));/*Right node after split*/
ptr->n = splitPos; /*No. of keys for left splitted node*/
(*newnode)->n = M-1-splitPos;/*No. of keys for right splitted node*/
for (i=0; i < (*newnode)->n; i++)
{
(*newnode)->p[i] = ptr->p[i + splitPos + 1];
if(i < (*newnode)->n - 1)
(*newnode)->keys[i] = ptr->keys[i + splitPos + 1];
else
(*newnode)->keys[i] = lastKey;
}
(*newnode)->p[(*newnode)->n] = lastPtr;
return InsertIt;
}/*End of ins()*/

void display(struct node *ptr, int blanks)
{
if (ptr)
{
int i;
for(i=1;i<=blanks;i++)
printf(" ");
for (i=0; i < ptr->n; i++)
printf("%d ",ptr->keys[i]);
printf("\n");
for (i=0; i <= ptr->n; i++)
display(ptr->p[i], blanks+10);
```

```
}/*End of if*/
}/*End of display()*/
void search(int key)
{
int pos, i, n;
struct node *ptr = root;
printf("Search path:\n");
while (ptr)
{
n = ptr->n;
for (i=0; i < ptr->n; i++)
printf(" %d",ptr->keys[i]);
printf("\n");
pos = searchPos(key, ptr->keys, n);
if (pos < n && key == ptr->keys[pos])
{
printf("Key %d found in position %d of last dispalyed node\n",key,i);
return;
}
ptr = ptr->p[pos];
}
printf("Key %d is not available\n",key);
}/*End of search()*/
int searchPos(int key, int *key_arr, int n)
{
int pos=0;
while (pos < n && key > key_arr[pos])
pos++;
return pos;
}/*End of searchPos()*/
void DelNode(int key)
{
struct node *uproot;
enum KeyStatus value;
value = del(root,key);
switch (value)
{
case SearchFailure:
printf("Key %d is not available\n",key);
break;
case LessKeys:
uproot = root;
root = root->p[0];
free(uproot);
break;
}/*End of switch*/
}/*End of delnode()*/
enum KeyStatus del(struct node *ptr, int key)
{
int pos, i, pivot, n ,min;
```

```
int *key_arr;
enum KeyStatus value;
struct node **p,*lptr,*rptr;
if (ptr == NULL)
return SearchFailure;
/*Assigns values of node*/
n=ptr->n;
key_arr = ptr->keys;
p = ptr->p;
min = (M - 1)/2;/*Minimum number of keys*/
pos = searchPos(key, key_arr, n);
if (p[0] == NULL)
{
if (pos == n || key < key_arr[pos])
return SearchFailure;
/*Shift keys and pointers left*/
for (i=pos+1; i < n; i++)
{
key_arr[i-1] = key_arr[i];
p[i] = p[i+1];
}
return --ptr->n >= (ptr==root ? 1 : min) ? Success : LessKeys;
}/*End of if */
if (pos < n && key == key_arr[pos])
{
struct node *qp = p[pos], *qp1;
int nkey;
while(1)
{
nkey = qp->n;
qp1 = qp->p[nkey];
if (qp1 == NULL)
break;
qp = qp1;
}/*End of while*/
key_arr[pos] = qp->keys[nkey-1];
qp->keys[nkey - 1] = key;
}/*End of if */
value = del(p[pos], key);
if (value != LessKeys)
return value;
if (pos > 0 && p[pos-1]->n > min)
{
pivot = pos - 1; /*pivot for left and right node*/
lptr = p[pivot];
rptr = p[pos];
rptr->p[rptr->n + 1] = rptr->p[rptr->n];
for (i=rptr->n; i>0; i--)
{
rptr->keys[i] = rptr->keys[i-1];
```

```
rptr->p[i] = rptr->p[i-1];
}
rptr->n++;
rptr->keys[0] = key_arr[pivot];
rptr->p[0] = lptr->p[lptr->n];
key_arr[pivot] = lptr->keys[--lptr->n];
return Success;
}
if (pos > min)
{
pivot = pos; /*pivot for left and right node*/
lptr = p[pivot];
rptr = p[pivot+1];
lptr->keys[lptr->n] = key_arr[pivot];
lptr->p[lptr->n + 1] = rptr->p[0];
key_arr[pivot] = rptr->keys[0];
lptr->n++;
rptr->n--;
for (i=0; i < rptr->n; i++)
{
rptr->keys[i] = rptr->keys[i+1];
rptr->p[i] = rptr->p[i+1];
}
rptr->p[rptr->n] = rptr->p[rptr->n + 1];
return Success;
}/*End of if */
if(pos == n)
pivot = pos-1;
else
pivot = pos;
lptr = p[pivot];
rptr = p[pivot+1];
lptr->keys[lptr->n] = key_arr[pivot];
lptr->p[lptr->n + 1] = rptr->p[0];
for (i=0; i < rptr->n; i++)
{
lptr->keys[lptr->n + 1 + i] = rptr->keys[i];
lptr->p[lptr->n + 2 + i] = rptr->p[i+1];
}
lptr->n = lptr->n + rptr->n +1;
free(rptr); /*Remove right node*/
for (i=pos+1; i < n; i++)
{
key_arr[i-1] = key_arr[i];
p[i] = p[i+1];
}
return --ptr->n >= (ptr == root ? 1 : min) ? Success : LessKeys;
}
```

RESULT : The above program is successfully executed and obtained the output

OUTPUT :

```
rony@rony-HP-Laptop-14s-cr2xxx:~/Documents/DS-lab$ cd "/home/rony/Documents/DS-lab/" &&
Creation of B tree for node 5
1.Insert
2.Delete
3.Search
4.Display
5.Quit
Enter your choice : 1
Enter the key : 2
1.Insert
2.Delete
3.Search
4.Display
5.Quit
Enter your choice : 1
Enter the key : 5
1.Insert
2.Delete
3.Search
4.Display
5.Quit
Enter your choice : 1
Enter the key : 9
1.Insert
2.Delete
3.Search
4.Display
5.Quit
Enter your choice : 4
Btree is :
2 5 9
1.Insert
2.Delete
3.Search
4.Display
5.Quit
Enter your choice : 3
Enter the key : 5
Search path:
 2 5 9
Key 5 found in position 3 of last dispalyed node
1.Insert
2.Delete
3.Search
4.Display
5.Quit
Enter your choice : 2
Enter the key : 9
1.Insert
2.Delete
3.Search
4.Display
5.Quit
Enter your choice : 4
Btree is :
2 5
1.Insert
2.Delete
3.Search
4.Display
5.Quit
Enter your choice : 5
```

PROGRAM 10 : RED-BLACK TREE

AIM : Write a program to implement red black tree and its operations.

ALGORITHM :

Following steps are followed for inserting a new element into a red-black tree:

1. Let y be the leaf (ie. NIL) and x be the root of the tree.
2. Check if the tree is empty (ie. whether x is NIL). If yes, insert newNode as a root node and color it black.
3. Else, repeat steps following steps until leaf (NIL) is reached.
   a. Compare newKey with rootKey.
   b. If newKey is greater than rootKey, traverse through the right subtree.
   c. Else traverse through the left subtree.
4. Assign the parent of the leaf as a parent of newNode.
5. If leafKey is greater than newKey, make newNode as rightChild.
6. Else, make newNode as leftChild.
7. Assign NULL to the left and rightChild of newNode.
8. Assign RED color to newNode.
9. Call InsertFix-algorithm to maintain the property of red-black tree if violated.

Algorithm to maintain red-black property after insertion

This algorithm is used for maintaining the property of a red-black tree if the insertion of a newNode violates this property.

1. Do the following while the parent of newNode p is RED.
2. If p is the left child of grandParent gP of z, do the following.

Case-I:

a. If the color of the right child of gP of z is RED, set the color of both the children of gP as BLACK and the color of gP as RED.

b. Assign gP to newNode.

Case-II:

c. Else if newNode is the right child of p then, assign p to newNode.

d. Left-Rotate newNode.

Case-III:

e. Set color of p as BLACK and color of gP as RED.

f. Right-Rotate gP.

3. Else, do the following.

a. If the color of the left child of gP of z is RED, set the color of both the children of gP as BLACK and the color of gP as RED.

b. Assign gP to newNode.

c. Else if newNode is the left child of p then, assign p to newNode and RightRotate newNode.

d. Set color of p as BLACK and color of gP as RED.

e. Left-Rotate gP.

4. Set the root of the tree as BLACK.

Deleting an element from a Red-Black Tree

Algorithm to delete a node

1. Save the color of nodeToBeDeleted in origrinalColor.

2. If the left child of nodeToBeDeleted is NULL

a. Assign the right child of nodeToBeDeleted to x.

b. Transplant nodeToBeDeleted with x.

3. Else if the right child of nodeToBeDeleted is NULL

a. Assign the left child of nodeToBeDeleted into x.

b. Transplant nodeToBeDeleted with x.

4. Else

a. Assign the minimum of right subtree of noteToBeDeleted into y.

b. Save the color of y in originalColor.

c. Assign the rightChild of y into x.

d. If y is a child of nodeToBeDeleted, then set the parent of x as y.

e. Else, transplant y with rightChild of y.

f. Transplant nodeToBeDeleted with y.

g. Set the color of y with originalColor.

5. If the originalColor is BLACK, call DeleteFix(x).

Algorithm to maintain Red-Black property after deletion

1. It reaches the root node.

2. If x points to a red-black node. In this case, x is colored black.

3. Suitable rotations and recoloring are performed.

The following algorithm retains the properties of a red-black tree.

1. Do the following until the x is not the root of the tree and the color of x is BLACK

2. If x is the left child of its parent then,

   a. Assign w to the sibling of x.

   b. If the right child of parent of x is RED,

Case-I:

a. Set the color of the right child of the parent of x as BLACK.

b. Set the color of the parent of x as RED.

c. Left-Rotate the parent of x.

d. Assign the rightChild of the parent of x to w.

e. If the color of both the right and the leftChild of w is BLACK,

Case-II:

a. Set the color of w as RED

b. Assign the parent of x to x.

c. Else if the color of the rightChild of w is BLACK

Case-III:

a. Set the color of the leftChild of w as BLACK

b. Set the color of w as RED

c. Right-Rotate w.

d. Assign the rightChild of the parent of x to w.

e. If any of the above cases do not occur, then do the following.

Case-IV:

a. Set the color of w as the color of the parent of x.

b. Set the color of the parent of x as BLACK.

c. Set the color of the right child of w as BLACK.

d. Left-Rotate the parent of x.

e. Set x as the root of the tree.

3. Else the same as above with right changed to left and vice versa.

4. Set the color of x as BLACK.

PROGRAM CODE :

10.c

INSERTION:

```
#include <stdio.h>
#include <stdlib.h>
enum COLOR {Red, Black};
typedef struct tree_node {
int data;
struct tree_node *right;
struct tree_node *left;
struct tree_node *parent;
enum COLOR color;
}tree_node;
typedef struct red_black_tree {
tree_node *root;
tree_node *NIL;
}red_black_tree;
tree_node* new_tree_node(int data) {
tree_node* n = malloc(sizeof(tree_node));
n->left = NULL;
n->right = NULL;
n->parent = NULL;
n->data = data;
n->color = Red;
return n;
}
```

```
red_black_tree* new_red_black_tree() {
red_black_tree *t = malloc(sizeof(red_black_tree));
tree_node *nil_node = malloc(sizeof(tree_node));
nil_node->left = NULL;
nil_node->right = NULL;
nil_node->parent = NULL;
nil_node->color = Black;
nil_node->data = 0;
t->NIL = nil_node;
t->root = t->NIL;
return t;
}
void left_rotate(red_black_tree *t, tree_node *x) {
tree_node *y = x->right;
x->right = y->left;
if(y->left != t->NIL) {
y->left->parent = x;
}
y->parent = x->parent;
if(x->parent == t->NIL) { //x is root
t->root = y;
}
else if(x == x->parent->left) { //x is left child
x->parent->left = y;
}
else { //x is right child
x->parent->right = y;
}
y->left = x;
x->parent = y;
```

```
}
void right_rotate(red_black_tree *t, tree_node *x) {
tree_node *y = x->left;
x->left = y->right;
if(y->right != t->NIL) {
y->right->parent = x;
}
y->parent = x->parent;
if(x->parent == t->NIL) { //x is root
t->root = y;
}
else if(x == x->parent->right) { //x is left child
x->parent->right = y;
}
else { //x is right child
x->parent->left = y;
}
y->right = x;
x->parent = y;
}
void insertion_fixup(red_black_tree *t, tree_node *z) {
while(z->parent->color == Red) {
if(z->parent == z->parent->parent->left) { //z.parent is the left child
tree_node *y = z->parent->parent->right; //uncle of z
if(y->color == Red) { //case 1
z->parent->color = Black;
y->color = Black;
z->parent->parent->color = Red;
z = z->parent->parent;
}
```

```
else { //case2 or case3
if(z == z->parent->right) { //case2
z = z->parent; //marked z.parent as new z
left_rotate(t, z);
}
//case3
z->parent->color = Black; //made parent black
z->parent->parent->color = Red; //made parent red
right_rotate(t, z->parent->parent);
}
}
else { //z.parent is the right child
tree_node *y = z->parent->parent->left; //uncle of z
if(y->color == Red) {
z->parent->color = Black;
y->color = Black;
z->parent->parent->color = Red;
z = z->parent->parent;
}
else {
if(z == z->parent->left) {
z = z->parent; //marked z.parent as new z
right_rotate(t, z);
}
z->parent->color = Black; //made parent black
z->parent->parent->color = Red; //made parent red
left_rotate(t, z->parent->parent);
}
}
}
```

```
t->root->color = Black;
}
void insert(red_black_tree *t, tree_node *z) {
tree_node* y = t->NIL; //variable for the parent of the added node
tree_node* temp = t->root;
while(temp != t->NIL) {
y = temp;
if(z->data < temp->data)
temp = temp->left;
else
temp = temp->right;
}
z->parent = y;
if(y == t->NIL) { //newly added node is root
t->root = z;
}
else if(z->data < y->data) //data of child is less than its parent, left child
y->left = z;
else
y->right = z;
z->right = t->NIL;
z->left = t->NIL;
insertion_fixup(t, z);
}
void inorder(red_black_tree *t, tree_node *n) {
if(n != t->NIL) {
inorder(t, n->left);
printf("%d\n", n->data);
inorder(t, n->right);
}
```

```
}
int main() {
red_black_tree *t = new_red_black_tree();
tree_node *a, *b, *c, *d, *e, *f, *g, *h, *i, *j, *k, *l, *m;
a = new_tree_node(10);
b = new_tree_node(20);
c = new_tree_node(30);
d = new_tree_node(100);
e = new_tree_node(90);
f = new_tree_node(40);
g = new_tree_node(50);
h = new_tree_node(60);
i = new_tree_node(70);
j = new_tree_node(80);
k = new_tree_node(150);
l = new_tree_node(110);
m = new_tree_node(120);
insert(t, a);
insert(t, b);
insert(t, c);
insert(t, d);
insert(t, e);
insert(t, f);
insert(t, g);
insert(t, h);
insert(t, i);
insert(t, j);
insert(t, k);
insert(t, l);
insert(t, m);
```

```
inorder(t, t->root);
return 0;
}
```

DELETION:

```
#include <stdio.h>
#include <stdlib.h>
enum COLOR {Red, Black};
typedef struct tree_node {
int data;
struct tree_node *right;
struct tree_node *left;
struct tree_node *parent;
enum COLOR color;
}tree_node;
typedef struct red_black_tree {
tree_node *root;
tree_node *NIL;
}red_black_tree;
tree_node* new_tree_node(int data) {
tree_node* n = malloc(sizeof(tree_node));
n->left = NULL;
n->right = NULL;
n->parent = NULL;
n->data = data;
n->color = Red;
return n;
}
red_black_tree* new_red_black_tree() {
red_black_tree *t = malloc(sizeof(red_black_tree));
tree_node *nil_node = malloc(sizeof(tree_node));
```

```
nil_node->left = NULL;
nil_node->right = NULL;
nil_node->parent = NULL;
nil_node->color = Black;
nil_node->data = 0;
t->NIL = nil_node;
t->root = t->NIL;
return t;
}
void left_rotate(red_black_tree *t, tree_node *x) {
tree_node *y = x->right;
x->right = y->left;
if(y->left != t->NIL) {
y->left->parent = x;
}
y->parent = x->parent;
if(x->parent == t->NIL) { //x is root
t->root = y;
}
else if(x == x->parent->left) { //x is left child
x->parent->left = y;
}
else { //x is right child
x->parent->right = y;
}
y->left = x;
x->parent = y;
}
void right_rotate(red_black_tree *t, tree_node *x) {
tree_node *y = x->left;
```

```
x->left = y->right;
if(y->right != t->NIL) {
y->right->parent = x;
}
y->parent = x->parent;
if(x->parent == t->NIL) { //x is root
t->root = y;
}
else if(x == x->parent->right) { //x is left child
x->parent->right = y;
}
else { //x is right child
x->parent->left = y;
}
y->right = x;
x->parent = y;
}
void insertion_fixup(red_black_tree *t, tree_node *z) {
while(z->parent->color == Red) {
if(z->parent == z->parent->parent->left) { //z.parent is the left child
tree_node *y = z->parent->parent->right; //uncle of z
if(y->color == Red) { //case 1
z->parent->color = Black;
y->color = Black;
z->parent->parent->color = Red;
z = z->parent->parent;
}
else { //case2 or case3
if(z == z->parent->right) { //case2
z = z->parent; //marked z.parent as new z
```

```
left_rotate(t, z);
}
//case3
z->parent->color = Black; //made parent black
z->parent->parent->color = Red; //made parent red
right_rotate(t, z->parent->parent);
}}
else { //z.parent is the right child
tree_node *y = z->parent->parent->left; //uncle of z
if(y->color == Red) {
z->parent->color = Black;
y->color = Black;
z->parent->parent->color = Red;
z = z->parent->parent;
}
else {
if(z == z->parent->left) {
z = z->parent; //marked z.parent as new z
right_rotate(t, z);
}
z->parent->color = Black; //made parent black
z->parent->parent->color = Red; //made parent red
left_rotate(t, z->parent->parent);
}}}
t->root->color = Black;
}
void insert(red_black_tree *t, tree_node *z) {
tree_node* y = t->NIL; //variable for the parent of the added node
tree_node* temp = t->root;
while(temp != t->NIL) {
```

```
y = temp;
if(z->data < temp->data)
temp = temp->left;
else
temp = temp->right;
}
z->parent = y;
if(y == t->NIL) { //newly added node is root
t->root = z;
}
else if(z->data < y->data) //data of child is less than its parent, left child
y->left = z;
else
y->right = z;
z->right = t->NIL;
z->left = t->NIL;
insertion_fixup(t, z);
}
void rb_transplant(red_black_tree *t, tree_node *u, tree_node *v) {
if(u->parent == t->NIL)
t->root = v;
else if(u == u->parent->left)
u->parent->left = v;
else
u->parent->right = v;
v->parent = u->parent;
}
tree_node* minimum(red_black_tree *t, tree_node *x) {
while(x->left != t->NIL)
x = x->left;
```

```
return x;
}
void rb_delete_fixup(red_black_tree *t, tree_node *x) {
while(x != t->root && x->color == Black) {
if(x == x->parent->left) {
tree_node *w = x->parent->right;
if(w->color == Red) {
w->color = Black;
x->parent->color = Red;
left_rotate(t, x->parent);
w = x->parent->right;
}
if(w->left->color == Black && w->right->color == Black) {
w->color = Red;
x = x->parent;
}
else {
if(w->right->color == Black) {
w->left->color = Black;
w->color = Red;
right_rotate(t, w);
w = x->parent->right;
}
w->color = x->parent->color;
x->parent->color = Black;
w->right->color = Black;
left_rotate(t, x->parent);
x = t->root;
}}
else {
```

```
tree_node *w = x->parent->left;
if(w->color == Red) {
w->color = Black;
x->parent->color = Red;
right_rotate(t, x->parent);
w = x->parent->left;
}
if(w->right->color == Black && w->left->color == Black) {
w->color = Red;
x = x->parent;
}
else {
if(w->left->color == Black) {
w->right->color = Black;
w->color = Red;
left_rotate(t, w);
w = x->parent->left;
}
w->color = x->parent->color;
x->parent->color = Black;
w->left->color = Black;
right_rotate(t, x->parent);
x = t->root;
}}}
x->color = Black;
}
void rb_delete(red_black_tree *t, tree_node *z) {
tree_node *y = z;
tree_node *x;
enum COLOR y_orignal_color = y->color;
```

```
if(z->left == t->NIL) {
x = z->right;
rb_transplant(t, z, z->right);
}
else if(z->right == t->NIL) {
x = z->left;
rb_transplant(t, z, z->left);
}
else {
y = minimum(t, z->right);
y_orignal_color = y->color;
x = y->right;
if(y->parent == z) {
x->parent = z;
}
else {
rb_transplant(t, y, y->right);
y->right = z->right;
y->right->parent = y;
}
rb_transplant(t, z, y);
y->left = z->left;
y->left->parent = y;
y->color = z->color;
}
if(y_orignal_color == Black)
rb_delete_fixup(t, x);
}
void inorder(red_black_tree *t, tree_node *n) {
if(n != t->NIL) {
```

```
inorder(t, n->left);
printf("%d\n", n->data);
inorder(t, n->right);
}}
int main() {
red_black_tree *t = new_red_black_tree();
tree_node *a, *b, *c, *d, *e, *f, *g, *h, *i, *j, *k, *l, *m;
a = new_tree_node(10);
b = new_tree_node(20);
c = new_tree_node(30);
d = new_tree_node(100);
e = new_tree_node(90);
f = new_tree_node(40);
g = new_tree_node(50);
h = new_tree_node(60);
i = new_tree_node(70);
j = new_tree_node(80);
k = new_tree_node(150);
l = new_tree_node(110);
m = new_tree_node(120);
insert(t, a);
insert(t, b);
insert(t, c);
insert(t, d);
insert(t, e);
insert(t, f);
insert(t, g);
insert(t, h);
insert(t, i);
insert(t, j);
```

```
insert(t, k);
insert(t, l);
insert(t, m);
rb_delete(t, a);
rb_delete(t, m);
inorder(t, t->root);
return 0;}
```

RESULT : The above program is successfully executed and obtained the output

OUTPUT :

INSERTION:

```
rony@rony-HP-Laptop-14s-cr2xxx:~/Documents/DS-lab$ cd "/home/rony/
10
20
30
40
50
60
70
80
90
100
110
120
150
```

DELETION:

```
rony@rony-HP-Laptop-14s-cr2xxx:~/Documents/DS-lab$ cd "/home/rony/Docume
20
30
40
50
60
70
80
90
100
110
150
```

PROGRAM 11 : DFS

AIM : Write a program to implement a DFS.

ALGORITHM :

DFS(G, u)

u.visited = true

for each v ∈ G.Adj[u]

if v.visited == false

DFS(G,v)

init()

{

For each u ∈ G

u.visited = false

For each u ∈ G

DFS(G, u)

}

PROGRAM CODE :

11.c	```#include <stdio.h>```

11.c

```
#include <stdio.h>
#include <stdlib.h>
struct node {
int vertex;
struct node* next;
};
struct node* createNode(int v);
struct Graph {
int numVertices;
int* visited;
// We need int** to store a two dimensional array.
// Similary, we need struct node** to store an array of Linked lists
struct node** adjLists;
};
// DFS algo
void DFS(struct Graph* graph, int vertex) {
```

	struct node* adjList = graph->adjLists[vertex];
	struct node* temp = adjList;
	graph->visited[vertex] = 1;
	printf("Visited %d \n", vertex);
	while (temp != NULL) {
	int connectedVertex = temp->vertex;
	if (graph->visited[connectedVertex] == 0) {
	DFS(graph, connectedVertex);
	}
	temp = temp->next;
	}}
	// Create a node
	struct node* createNode(int v) {
	struct node* newNode = malloc(sizeof(struct node));
	newNode->vertex = v;
	newNode->next = NULL;
	return newNode;
	}
	// Create graph
	struct Graph* createGraph(int vertices) {
	struct Graph* graph = malloc(sizeof(struct Graph));
	graph->numVertices = vertices;
	graph->adjLists = malloc(vertices * sizeof(struct node*));
	graph->visited = malloc(vertices * sizeof(int));
	int i;
	for (i = 0; i < vertices; i++) {
	graph->adjLists[i] = NULL;
	graph->visited[i] = 0;
	}
	return graph;
	}
	// Add edge
	void addEdge(struct Graph* graph, int src, int dest) {
	// Add edge from src to dest
	struct node* newNode = createNode(dest);
	newNode->next = graph->adjLists[src];
	graph->adjLists[src] = newNode;
	// Add edge from dest to src
	newNode = createNode(src);
	newNode->next = graph->adjLists[dest];
	graph->adjLists[dest] = newNode;
	}
	// Print the graph
	void printGraph(struct Graph* graph) {
	int v;
	for (v = 0; v < graph->numVertices; v++) {
	struct node* temp = graph->adjLists[v];
	printf("\n Adjacency list of vertex %d\n ", v);
	while (temp) {
	printf("%d -> ", temp->vertex);

```
temp = temp->next;
}
printf("\n");
}}
int main() {
struct Graph* graph = createGraph(4);
addEdge(graph, 0, 1);
addEdge(graph, 0, 2);
addEdge(graph, 1, 2);
addEdge(graph, 2, 3);
printGraph(graph);
DFS(graph, 2);
return 0;
}
```

RESULT : The above program is successfully executed and obtained the output

OUTPUT :

```
rony@rony-HP-Laptop-14s-cr2xxx:~/Documents/DS-lab$ cd "/home/rony/Documents/DS-lab/" && gcc DFS.c -o DFS && "/home/rony/Documents/DS-lab/"DFS

 Adjacency list of vertex 0
 2 -> 1 ->

 Adjacency list of vertex 1
 2 -> 0 ->

 Adjacency list of vertex 2
 3 -> 1 -> 0 ->

 Adjacency list of vertex 3
 2 ->
Visited 2
Visited 3
Visited 1
Visited 0
```

PROGRAM 12 : BFS

AIM : Write a program to implement BFS.

ALGORITHM :

1. Create a queue Q .
2. Mark v as visited and put v into Q .
3. While Q is non-empty .

   Remove the head u of Q .

   Mark and enqueue all (unvisited) neighbours of u.

PROGRAM CODE :

12.c	`#include <stdio.h>`
	`#include <stdlib.h>`
	`#define SIZE 40`
	`struct queue {`
	` int items[SIZE];`
	` int front;`
	` int rear;`
	`};`
	`struct queue* createQueue();`
	`void enqueue(struct queue* q, int);`
	`int dequeue(struct queue* q);`
	`void display(struct queue* q);`
	`int isEmpty(struct queue* q);`
	`void printQueue(struct queue* q);`
	`struct node {`
	` int vertex;`
	` struct node* next;`
	`};`
	`struct node* createNode(int);`
	`struct Graph {`
	` int numVertices;`
	` struct node** adjLists;`
	` int* visited;`
	`};`
	`// BFS algorithm`

```
void bfs(struct Graph* graph, int startVertex) {
 struct queue* q = createQueue();
 graph->visited[startVertex] = 1;
 enqueue(q, startVertex);
 while (!isEmpty(q)) {
 printQueue(q);
 int currentVertex = dequeue(q);
 printf("Visited %d\n", currentVertex);
 struct node* temp = graph->adjLists[currentVertex];
 while (temp) {
 int adjVertex = temp->vertex;
 if (graph->visited[adjVertex] == 0) {
 graph->visited[adjVertex] = 1;
 enqueue(q, adjVertex);
 }
 temp = temp->next;
 }
 }
}
// Creating a node
struct node* createNode(int v) {
 struct node* newNode = malloc(sizeof(struct node));
 newNode->vertex = v;
 newNode->next = NULL;
 return newNode;
}
// Creating a graph
struct Graph* createGraph(int vertices) {
 struct Graph* graph = malloc(sizeof(struct Graph));
 graph->numVertices = vertices;
 graph->adjLists = malloc(vertices * sizeof(struct node*));
 graph->visited = malloc(vertices * sizeof(int));
 int i;
 for (i = 0; i < vertices; i++) {
 graph->adjLists[i] = NULL;
 graph->visited[i] = 0;
 }
 return graph;
}
// Add edge
void addEdge(struct Graph* graph, int src, int dest) {
 // Add edge from src to dest
 struct node* newNode = createNode(dest);
 newNode->next = graph->adjLists[src];
 graph->adjLists[src] = newNode;
 // Add edge from dest to src
 newNode = createNode(src);
 newNode->next = graph->adjLists[dest];
 graph->adjLists[dest] = newNode;
}
```

```
// Create a queue
struct queue* createQueue() {
 struct queue* q = malloc(sizeof(struct queue));
 q->front = -1;
 q->rear = -1;
 return q;
}
// Check if the queue is empty
int isEmpty(struct queue* q) {
 if (q->rear == -1)
 return 1;
 else
 return 0;
}
// Adding elements into queue
void enqueue(struct queue* q, int value) {
 if (q->rear == SIZE - 1)
 printf("\nQueue is Full!!");
 else {
 if (q->front == -1)
 q->front = 0;
 q->rear++;
 q->items[q->rear] = value;
 }
}
// Removing elements from queue
int dequeue(struct queue* q) {
 int item;
 if (isEmpty(q)) {
 printf("Queue is empty");
 item = -1;
 } else {
 item = q->items[q->front];
 q->front++;
 if (q->front > q->rear) {
 printf("Resetting queue ");
 q->front = q->rear = -1;
 }
 }
 return item;
}
// Print the queue
void printQueue(struct queue* q) {
 int i = q->front;
 if (isEmpty(q)) {
 printf("Queue is empty");
 } else {
 printf("\nQueue contains \n");
 for (i = q->front; i < q->rear + 1; i++) {
 printf("%d ", q->items[i]);
```

```
 }
 }
}
int main() {
 struct Graph* graph = createGraph(6);
 addEdge(graph, 0, 1);
 addEdge(graph, 0, 2);
 addEdge(graph, 1, 2);
 addEdge(graph, 1, 4);
 addEdge(graph, 1, 3);
 addEdge(graph, 2, 4);
 addEdge(graph, 3, 4);
 bfs(graph, 0);
 return 0;
}
```

RESULT : The above program is successfully executed and obtained the output

OUTPUT :

```
rony@rony-HP-Laptop-14s-cr2xxx:~/Documents/DS-lab$ cd "/home/rony/Documents/DS-lab/" && gcc BFS.c -o BFS && "/home/rony/Documents/DS-lab/"BFS

Queue contains
0 Resetting queue Visited 0

Queue contains
2 1 Visited 2

Queue contains
1 4 Visited 1

Queue contains
4 3 Visited 4

Queue contains
3 Resetting queue Visited 3
```

PROGRAM 13 : TOPOLOGICAL SORTING

AIM : Write a program to implement a topological sorting.

ALGORITHM :

1) Identify a node with no incoming edges.
2) Add that node to the ordering.
3) Remove it from the graph.
4) Repeat.

PROGRAM CODE :

13.c

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

#define MAX 100

int n; /*Number of vertices in the graph*/
int adj[MAX][MAX]; /*Adjacency Matrix*/
void create_graph();

int queue[MAX], front = -1,rear = -1;
void insert_queue(int v);
int delete_queue();
int isEmpty_queue();

int indegree(int v);

int main()
{
int i,v,count,topo_order[MAX],indeg[MAX];

create_graph();

/*Find the indegree of each vertex*/
for(i=0;i<n;i++)
{
indeg[i] = indegree(i);
if( indeg[i] == 0 )
```

```
insert_queue(i);
}
count = 0;
while( !isEmpty_queue( ) && count < n )
{
v = delete_queue();
topo_order[++count] = v; /*Add vertex v to topo_order array*/
/*Delete all edges going fron vertex v */
for(i=0; i<n; i++)
{
if(adj[v][i] == 1)
{
adj[v][i] = 0;
indeg[i] = indeg[i]-1;
if(indeg[i] == 0)
insert_queue(i);
}
}
}

if( count < n )
{
printf("\nNo topological ordering possible, graph contains cycle\n");
exit(1);
}
printf("\nVertices in topological order are :\n");
for(i=1; i<=count; i++)
printf( "%d ",topo_order[i] );
printf("\n");

return 0;
}/*End of main()*/

void insert_queue(int vertex)
{
if (rear == MAX-1)
printf("\nQueue Overflow\n");
else
{
if (front == -1) /*If queue is initially empty */
front = 0;
rear = rear+1;
queue[rear] = vertex ;
}
}/*End of insert_queue()*/

int isEmpty_queue()
{
if(front == -1 || front > rear )
return 1;
```

```
else
return 0;
}/*End of isEmpty_queue()*/

int delete_queue()
{
int del_item;
if (front == -1 || front > rear)
{
printf("\nQueue Underflow\n");
exit(1);
}
else
{
del_item = queue[front];
front = front+1;
return del_item;
}
}/*End of delete_queue() */

int indegree(int v)
{
int i,in_deg = 0;
for(i=0; i<n; i++)
if(adj[i][v] == 1)
in_deg++;
return in_deg;
}/*End of indegree() */

void create_graph()
{
int i,max_edges,origin,destin;
printf("\nEnter number of vertices : ");
scanf("%d",&n);
max_edges = n*(n-1);
for(i=1; i<=max_edges; i++)
{
printf("\nEnter edge %d(-1 -1 to quit): ",i);
scanf("%d %d",&origin,&destin);
if((origin == -1) && (destin == -1))
break;
if( origin >= n || destin >= n || origin<0 || destin<0)
{
printf("\nInvalid edge!\n");
i--;
}
else
adj[origin][destin] = 1;
}}
```

RESULT : The above program is successfully executed and obtained the output

OUTPUT :

```
rony@rony-HP-Laptop-14s-cr2xxx:~/Documents/DS-lab$ cd "/home/rony/Documen

Enter number of vertices : 4

Enter edge 1(-1 -1 to quit): 0
1

Enter edge 2(-1 -1 to quit): 0
2

Enter edge 3(-1 -1 to quit): 2
3

Enter edge 4(-1 -1 to quit): 1
2

Enter edge 5(-1 -1 to quit): 1
3

Enter edge 6(-1 -1 to quit): -1
-1

Vertices in topological order are :
0 1 2 3
```

PROGRAM 14 : STRONGLY CONNECTED COMPONENT

AIM : Write a program to find the strongly connected components in a directed graph.

ALGORITHM :

1. Perform a depth first search on the whole graph.
2. Reverse the original graph.
3. Perform depth-first search on the reversed graph.

PROGRAM CODE :

14.c

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_DEGREE 5
#define MAX_NUM_VERTICES 20
struct vertices_s {
    int visited;
    int deg;
    int adj[MAX_DEGREE]; /* < 0 if incoming edge */
} vertices[] = {
    {0, 3, {2, -3, 4}},
    {0, 2, {-1, 3}},
    {0, 3, {1, -2, 7}},
    {0, 3, {-1, -5, 6}},
    {0, 2, {4, -7}},
    {0, 3, {-4, 7, -8}},
    {0, 4, {-3, 5, -6, -12}},
    {0, 3, {6, -9, 11}},
    {0, 2, {8, -10}},
    {0, 3, {9, -11, -12}},
    {0, 3, {-8, 10, 12}},
    {0, 3, {7, 10, -11}}
};
int num_vertices = sizeof(vertices) / sizeof(vertices[0]);
struct stack_s {
    int top;
    int items[MAX_NUM_VERTICES];
} stack = {-1, {}};
void stack_push(int v) {
    stack.top++;
```

```
    if (stack.top < MAX_NUM_VERTICES)
        stack.items[stack.top] = v;
    else {
        printf("Stack is full!\n");
        exit(1);
    }}
int stack_pop() {
    return stack.top < 0 ? -1 : stack.items[stack.top--];
}
void dfs(int v, int transpose) {
    int i, c, n;
    vertices[v].visited = 1;
    for (i = 0, c = vertices[v].deg; i < c; ++i) {
        n = vertices[v].adj[i] * transpose;
        if (n > 0)
            /* n - 1 because vertex indexing begins at 0 */
            if (!vertices[n - 1].visited)
                dfs(n - 1, transpose);
    }
    if (transpose < 0)
        stack_push(v);
    else
        printf("%d ", v + 1);
}
void reset_visited() {
    int i;
    for (i = 0; i < num_vertices; ++i)
        vertices[i].visited = 0;
}
void order_pass() {
    int i;
    for (i = 0; i < num_vertices; ++i)
        if (!vertices[i].visited)
            dfs(i, -1);
}
void scc_pass() {
    int i = 0, v;
    while((v = stack_pop()) != -1) {
        if (!vertices[v].visited) {
            printf("scc %d: ", ++i);
            dfs(v, 1);
            printf("\n");
        } }}
int main(void) {
    order_pass();
    reset_visited();
    scc_pass();
    return 0;
}
```

RESULT : The above program is successfully executed and obtained the output

OUTPUT :

```
rony@rony-HP-Laptop-14s-cr2xxx:~/Documents/DS-lab$ cd "/home/rony/Documents/DS-l
ab/" && gcc scc.c -o scc && "/home/rony/Documents/DS-lab/"scc
scc 1: 5 7 6 4
scc 2: 9 10 12 11 8
scc 3: 3 2 1
```

PROGRAM 15 : PRIM'S ALGORITHM

AIM : Write a program to implement prim's algorithm for finding the minimum cost spanning tree

ALGORITHM :

1) Create a set mstSet that keeps track of vertices already included in MST.
2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
3) While mstSet doesn't include all vertices

   a) Pick a vertex u which is not there in mstSet and has minimum key value.
   b) Include u to mstSet.
   c) Update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v.The idea of using key values is to pick the minimum weight edge from cut. The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

PROGRAM CODE :

15.c

```
#include<stdio.h>
#include<stdbool.h>
#include<string.h>
#define INF 9999999
#define V 5
int G[V][V] = {
{0, 9, 75, 0, 0},
{9, 0, 95, 19, 42},
{75, 95, 0, 51, 66},
{0, 19, 51, 0, 31},
{0, 42, 66, 31, 0}};
int main() {
int no_edge;
int selected[V];
memset(selected, false, sizeof(selected));
no_edge = 0;
```

```
selected[0] = true;
int x; // row number
int y; // col number
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;
}
```

RESULT : The above program is successfully executed and obtained the output

OUTPUT :

```
rony@rony-HP-Laptop-14s-cr2xxx:~/Documents/DS-lab$ cd "/home/rony/Documents/DS-lab/" && gcc prims.c
Edge : Weight
0 - 1 : 9
1 - 3 : 19
3 - 4 : 31
3 - 2 : 51
```

PROGRAM 16 : KRUSKAL'S ALGORITHM

AIM : Write a program to implement kruskal's algorithm using the disjoint set data structure

ALGORITHM :

KRUSKAL(G):

1. A = ∅
2. For each vertex v ∈ G.V:

   MAKE-SET(v)
3. For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):

   if FIND-SET(u) ≠ FIND-SET(v):

      A = A ∪ {(u, v)}

      UNION(u, v)
4. return A

PROGRAM CODE :

16.c

```c
#include <stdio.h>
#define MAX 30
typedef struct edge {
 int u, v, w;
} edge;
typedef struct edge_list {
 edge data[MAX];
 int n;
} edge_list;
edge_list elist;
int Graph[MAX][MAX], n;
edge_list spanlist;
void kruskalAlgo();
int find(int belongs[], int vertexno);
void applyUnion(int belongs[], int c1, int c2);
void sort();
void print();
void kruskalAlgo() {
 int belongs[MAX], i, j, cno1, cno2;
 elist.n = 0;
 for (i = 1; i < n; i++)
```

```
for (j = 0; j < i; j++) {
if (Graph[i][j] != 0) {
elist.data[elist.n].u = i;
elist.data[elist.n].v = j;
elist.data[elist.n].w = Graph[i][j];
elist.n++;
} }
sort();
for (i = 0; i < n; i++)
belongs[i] = i;
spanlist.n = 0;
for (i = 0; i < elist.n; i++) {
cno1 = find(belongs, elist.data[i].u);
cno2 = find(belongs, elist.data[i].v);
if (cno1 != cno2) {
spanlist.data[spanlist.n] = elist.data[i];
spanlist.n = spanlist.n + 1;
applyUnion(belongs, cno1, cno2);
}
}
}
int find(int belongs[], int vertexno) {
return (belongs[vertexno]);
}
void applyUnion(int belongs[], int c1, int c2) {
int i;
for (i = 0; i < n; i++)
if (belongs[i] == c2)
belongs[i] = c1;
}
void sort() {
int i, j;
edge temp;
for (i = 1; i < elist.n; i++)
for (j = 0; j < elist.n - 1; j++)
if (elist.data[j].w > elist.data[j + 1].w) {
temp = elist.data[j];
elist.data[j] = elist.data[j + 1];
elist.data[j + 1] = temp;
}}
void print() {
int i, cost = 0;
for (i = 0; i < spanlist.n; i++) {
printf("\n%d - %d : %d", spanlist.data[i].u, spanlist.data[i].v, spanlist.data[i].w);
cost = cost + spanlist.data[i].w;
}
printf("\nSpanning tree cost: %d", cost);
printf("\n");
}
int main() {
```

```
int i, j, total_cost;
n = 6;
Graph[0][0] = 0;
Graph[0][1] = 4;
Graph[0][2] = 4;
Graph[0][3] = 0;
Graph[0][4] = 0;
Graph[0][5] = 0;
Graph[0][6] = 0;
Graph[1][0] = 4;
Graph[1][1] = 0;
Graph[1][2] = 2;
Graph[1][3] = 0;
Graph[1][4] = 0;
Graph[1][5] = 0;
Graph[1][6] = 0;
Graph[2][0] = 4;
Graph[2][1] = 2;
Graph[2][2] = 0;
Graph[2][3] = 3;
Graph[2][4] = 4;
Graph[2][5] = 0;
Graph[2][6] = 0;
Graph[3][0] = 0;
Graph[3][1] = 0;
Graph[3][2] = 3;
Graph[3][3] = 0;
Graph[3][4] = 3;
Graph[3][5] = 0;
Graph[3][6] = 0;
Graph[4][0] = 0;
Graph[4][1] = 0;
Graph[4][2] = 4;
Graph[4][3] = 3;
Graph[4][4] = 0;
Graph[4][5] = 0;
Graph[4][6] = 0;
Graph[5][0] = 0;
Graph[5][1] = 0;
Graph[5][2] = 2;
Graph[5][3] = 0;
Graph[5][4] = 3;
Graph[5][5] = 0;
Graph[5][6] = 0;
kruskalAlgo();
print();
}
```

RESULT : The above program is successfully executed and obtained the output

OUTPUT :

```
rony@rony-HP-Laptop-14s-cr2xxx:~/Documents/DS-lab$ cd "/home/rony/Documents/DS-lab/" &&

2 - 1 : 2
5 - 2 : 2
3 - 2 : 3
4 - 3 : 3
1 - 0 : 4
Spanning tree cost: 14
```

PROGRAM 17 : SINGLE SOURCE SHORTEST PATH ALGORITHM

AIM : Write a program to implement single source shortest path algorithm using any heap structure that supports mergeable heap operations.

ALGORITHM :

1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

3) While sptSet doesn't include all vertices

   a) Pick a vertex u which is not there in sptSet and has minimum distance value.

   b) Include u to sptSet.

   c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

PROGRAM CODE :

16.c

```
#include <stdio.h>
#define INFINITY 9999
#define MAX 10

void Dijkstra(int Graph[MAX][MAX], int n, int start);

void Dijkstra(int Graph[MAX][MAX], int n, int start) {
  int cost[MAX][MAX], distance[MAX], pred[MAX];
  int visited[MAX], count, mindistance, nextnode, i, j;

  // Creating cost matrix
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
```

```
    if (Graph[i][j] == 0)
      cost[i][j] = INFINITY;
    else
      cost[i][j] = Graph[i][j];

  for (i = 0; i < n; i++) {
    distance[i] = cost[start][i];
    pred[i] = start;
    visited[i] = 0;
  }

  distance[start] = 0;
  visited[start] = 1;
  count = 1;

  while (count < n - 1) {
    mindistance = INFINITY;

    for (i = 0; i < n; i++)
      if (distance[i] < mindistance && !visited[i]) {
        mindistance = distance[i];
        nextnode = i;
      }

    visited[nextnode] = 1;
    for (i = 0; i < n; i++)
      if (!visited[i])
        if (mindistance + cost[nextnode][i] < distance[i]) {
          distance[i] = mindistance + cost[nextnode][i];
          pred[i] = nextnode;
        }
    count++;
  }

  // Printing the distance
  for (i = 0; i < n; i++)
    if (i != start) {
      printf("\nDistance from source to %d: %d", i, distance[i]);
    }
}
int main() {
  int Graph[MAX][MAX], i, j, n, u;
  n = 7;

  Graph[0][0] = 0;
  Graph[0][1] = 0;
  Graph[0][2] = 1;
  Graph[0][3] = 2;
  Graph[0][4] = 0;
  Graph[0][5] = 0;
```

```
Graph[0][6] = 0;

Graph[1][0] = 0;
Graph[1][1] = 0;
Graph[1][2] = 2;
Graph[1][3] = 0;
Graph[1][4] = 0;
Graph[1][5] = 3;
Graph[1][6] = 0;

Graph[2][0] = 1;
Graph[2][1] = 2;
Graph[2][2] = 0;
Graph[2][3] = 1;
Graph[2][4] = 3;
Graph[2][5] = 0;
Graph[2][6] = 0;

Graph[3][0] = 2;
Graph[3][1] = 0;
Graph[3][2] = 1;
Graph[3][3] = 0;
Graph[3][4] = 0;
Graph[3][5] = 0;
Graph[3][6] = 1;

Graph[4][0] = 0;
Graph[4][1] = 0;
Graph[4][2] = 3;
Graph[4][3] = 0;
Graph[4][4] = 0;
Graph[4][5] = 2;
Graph[4][6] = 0;

Graph[5][0] = 0;
Graph[5][1] = 3;
Graph[5][2] = 0;
Graph[5][3] = 0;
Graph[5][4] = 2;
Graph[5][5] = 0;
Graph[5][6] = 1;

Graph[6][0] = 0;
Graph[6][1] = 0;
Graph[6][2] = 0;
Graph[6][3] = 1;
Graph[6][4] = 0;
Graph[6][5] = 1;
Graph[6][6] = 0;
```

```
  u = 0;
  Dijkstra(Graph, n, u);
  printf("\n");
  return 0;
}
```

RESULT : The above program is successfully executed and obtained the output

OUTPUT :

```
rony@rony-HP-Laptop-14s-cr2xxx:~/Documents/just_do_it/DS$ cd "/home/ror
o_it/DS/"dijkstra

Distance from source to 1: 3
Distance from source to 2: 1
Distance from source to 3: 2
Distance from source to 4: 4
Distance from source to 5: 4
Distance from source to 6: 3
```