Experiment 18 **Date:12/11/2025**

Set Data Structures

Aim:

Create the Abstract Data Type (ADT) using Set and perform the operations Union, Intersection and Difference operations. Implement using Bit Strings**.**

Algorithm:

1. Start.
2. Initialize arrays a[11], b[11], and res[11] to 0.
3. Read number of elements in Set A.
4. Call input(a, n) to read elements of Set A.
5. Read number of elements in Set B.
6. Call input(b, n) to read elements of Set B.
7. Display menu of operations.
8. Read choice.
9. If choice = 1 → call set_union().
10. If choice = 2 → call set_intersection().
11. If choice = 3 → call set_difference().
12. If choice = 4 → call set_equality().
13. If choice = 5 → exit.
14. Stop.

void input()

1. Start
2. Read number of elements.
3. Repeat for each element:
   Read element x.
   Set bs[x] = 1.
4. Stop

void set_union()

1. Start
2. Repeat for i = 1 to 10:
   res[i] = a[i] OR b[i]
3. Display union set.
4. Stop.

void set_intersection()

1. Start
2. Repeat for i = 1 to 10:
3. res[i] = a[i] AND b[i]
4. Display intersection set.
5. Stop.

void set_difference()

1. Start
2. Repeat for i = 1 to 10:
   res[i] = a[i] AND NOT b[i]
3. Display difference set.
4. Stop.

bool set_equality()

1. Start
2. Repeat for i = 1 to 10:
   If a[i] ≠ b[i], return false.
3. If all elements are equal, return true.
4. Stop.

Program

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

int a[11], b[11], res[11];
int U[11]={1,2,3,4,5,6,7,8,9,10};

void display(int bs[]){
for(int i=1;i<11;i++){
        printf("%d\t",bs[i]);
 }
}
void input(int bs[], int n){
 int x;
printf("Enter the elements : ");
for(int i=0;i<n;i++){
        scanf("%d",&x);
        bs[x]=1;
 }
}

void set_union(){
 for(int i=1;i<11;i++){
       res[i]=a[i] | b[i];
}
 printf("\nUnion Set : ");
display(res);
}

void set_intersection(){
for(int i=1;i<11;i++){
        res[i]=a[i] & b[i];
}
printf("\nIntersection Set : ");
display(res);
}

void set_difference(){
for(int i=1;i<11;i++){
        res[i]=a[i] & ~b[i];
}
 printf("\nDifference Set : ");
display(res);
}
```

```
    bool set_equality(){
        for(int i=1;i<11;i++){
            if(a[i] != b[i]){
                return false;
            }
        }
        return true;
    }

int main() {
        int n1, n2, choice;
    for(int i = 1; i < 11; i++) {
            a[i] = b[i] = res[i] = 0;
        }

        printf("Enter number of elements in Set A: ");
        scanf("%d", &n1);
        input(a, n1);

        printf("Enter number of elements in Set B: ");
        scanf("%d", &n2);
        input(b, n2);
        printf("\n1. Union");
        printf("\n2. Intersection");
        printf("\n3. Difference (A - B)");
        printf("\n4. Equality");
        printf("\n5. Exit");

        do {
            printf("\n\nEnter your choice: ");
            scanf("%d", &choice);

            switch(choice) {
                case 1:
                    set_union();
                    break;

                case 2:
                    set_intersection();
                    break;

                case 3:
                    set_difference();
                    break;
```

```
            case 4:
                if(set_equality())
                    printf("\nSets A and B are Equal");
                else
                    printf("\nSets A and B are Not Equal");
                break;

            case 5:
                printf("\nExiting program...");
                break;

            default:
                printf("\nInvalid choice");
        }

    } while(choice != 5);

    return 0;
}
```

Output

```
Enter number of elements in Set A: 4
Enter the elements : 1 2 3 4
Enter number of elements in Set B: 4
Enter the elements : 5 6 7 8

1. Union
2. Intersection
3. Difference (A - B)
4. Equality
5. Exit

Enter your choice: 1

Union Set : 1   1   1   1   1   1   1   1   0   0

Enter your choice: 2

Intersection Set : 0   0   0   0   0   0   0   0   0   0
```

Enter your choice: 3

Difference Set : 1 1 1 1 0 0 0 0 0 0

Enter your choice: 4

Sets A and B are Not Equal

Enter your choice: 5

Experiment 19 **Date:12/11/2025**

Disjoint Set Data Structures

Aim:

Implement the Disjoint set ADT with Create, Union and Find operations

Algorithm:

1. Start
2. declare i
3. numElements = 6
4. unionSets(0, 1);
5. unionSets(1, 2);
6. unionSets(3, 4);
7. unionSets(4, 5);
8. unionSets(2, 4);
9. set i = 0,
10. if i<numElements then
    Print find(i)
11. Stop

void initSets()

1. start
2. declare ;
3. i = 0,i<numElements
4. sets[i].parent = i
5. sets[i].rank = 0
6. stop

int find(int)

1. start
2. if (sets[element].parent != element) then
   sets[element].parent = find(sets[element].parent)
3. return sets[element].parent
4. stop

void unionSets(int, int)

1. start
2. int set1 = find(element1);
3. int set2 = find(element2);
4. if (set1 != set2) then
   if (sets[set1].rank> sets[set2].rank)
   sets[set2].parent = set1
   else if (sets[set1].rank< sets[set2].rank)
   sets[set1].parent = set2
5. else
   sets[set2].parent = set1;
6. sets[set1].rank++;
7. stop

void displaySets()

1. declare i;
2. i = 0, i<numElements
3. print i
4. for (i = 0; i<numElements; i++) {
5. print sets[i].parent
6. i = 0, i<numElements
7. print sets[i].rank

Program

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_ELEMENTS 1000
typedef struct Set {
    int parent;
    int rank;
} Set;

Set sets[MAX_ELEMENTS];
int numElements;
void initSets() {
    int i;
    for (i = 0; i<numElements; i++) {
        sets[i].parent = i;
        sets[i].rank = 0;
    }
}
int find(int element) {
    if (sets[element].parent != element) {
        sets[element].parent = find(sets[element].parent);
    }
    return sets[element].parent;
}
void unionSets(int element1, int element2) {
    int set1 = find(element1);
    int set2 = find(element2);
    if (set1 != set2) {
        // Union by rank
        if (sets[set1].rank> sets[set2].rank) {
            sets[set2].parent = set1;
        } else if (sets[set1].rank< sets[set2].rank) {
            sets[set1].parent = set2;
        } else {
            sets[set2].parent = set1;
            sets[set1].rank++;
        }
    }
}
```

```
void displaySets() {
    int i;
printf("Element:\t");
    for (i = 0; i<numElements; i++) {
printf("%d\t", i);
    }
printf("\nParent:\t");
    for (i = 0; i<numElements; i++) {
printf("%d\t", sets[i].parent);
    }
printf("\nRank:\t");
    for (i = 0; i<numElements; i++) {
printf("%d\t", sets[i].rank);
    }
printf("\n\n");
}

int main() {
    int i;
numElements = 6;
initSets();
displaySets();
unionSets(0, 1);
unionSets(1, 2);
unionSets(3, 4);
unionSets(4, 5);
unionSets(2, 4);
displaySets();
    for (i = 0; i<numElements; i++) {
printf("The representative element of element %d is %d\n", i, find(i));
    }
    return 0;
}
```

Output

Element:	0	1	2	3	4	5
Parent:	0	1	2	3	4	5
Rank: 0	0	0	0	0	0	

Element:	0	1	2	3	4	5
Parent:	0	0	0	0	3	3
Rank: 2	0	0	1	0	0	

The representative element of element 0 is 0
The representative element of element 1 is 0
The representative element of element 2 is 0
The representative element of element 3 is 0
The representative element of element 4 is 0
The representative element of element 5 is 0

Experiment 20 **Date:19/11/2025**

Binary Search Tree Operations

Aim:

Menu Driven program to implement Binary Search Tree (BST) Operations Insertion of node, Deletion of a node, In-order traversal, Pre-order traversal and post-order traversal**.**

Algorithm:

1. Start
2. Declare root = NULL
3. Display menu
4. Read user choice
5. If choice == 1
    1. Read value
    2. root = insert(root, value)
6. If choice == 2
    1. Read value
    2. root = deleteNode(root, value)
7. If choice == 3
    1. Call inorder(root)
8. If choice == 4
    1. Call preorder(root)
9. If choice == 5
    1. Call postorder(root)
10. Repeat steps 2–15 until choice == 6
11. Stop

createNode(value)

1. Start
2. Allocate memory for new node
3. Assign value to newNode->data
4. Set newNode->left = NULL
5. Set newNode->right = NULL
6. Return newNode
7. Stop

insert(root, value)

1. Start
2. If root == NULL then
    Create and return new node
3. If value < root->data then

root->left = insert(root->left, value)
4. Else if value > root->data
   root->right = insert(root->right, value)
5. Return root
6. Stop

void findMin(root)
1. Repeat while root->left != NULL
   root = root->left
2. Return root

deleteNode(root, value)
1. Start
2. If root == NULL
   Return root
3. If value < root->data then
   root->left = deleteNode(root->left, value)
4. Else if value > root->data then
   root->right = deleteNode(root->right, value)
5. Else
   If root->left == NULL
      temp = root->right
   Free root
   Return temp
6. Else if root->right == NULL then
   temp = root->left
   Free root
   Return temp
7. root->data = temp->data
8. root->right = deleteNode(root->right, temp->data)
9. Return root
10. Stop

void inorder(root)
1. Start
2. If root != NULL then
   inorder(root->left)
   Print root->data
   inorder(root->right)
3. Stop

void preorder(root)
1. Start
2. If root != NULL then
   Print root->data

preorder(root->left)
preorder(root->right)

3. Stop

void postorder(root)

1. Start
2. If root != NULL
   postorder(root->left)
   postorder(root->right)
   Print root->data
3. Stop

Program

```
#include  <stdio.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *left, *right;
};

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

struct node* insert(struct node* root, int value) {
   if (root == NULL)
      return createNode(value);

   if (value < root->data)
      root->left = insert(root->left, value);
   else if (value > root->data)
      root->right = insert(root->right, value);

   return root;
}

struct node* findMin(struct node* root) {
   while (root->left != NULL)
      root = root->left;
```

```
    return root;
}

struct node* deleteNode(struct node* root, int value) {
    if (root == NULL)
        return root;

    if (value < root->data)
        root->left = deleteNode(root->left, value);
    else if (value > root->data)
        root->right = deleteNode(root->right, value);
    else {
        // Node with one 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
        struct node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
void inorder(struct node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
void preorder(struct node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}
```

```
void postorder(struct node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}
int main() {
    struct node* root = NULL;
    int choice, value;
    do {
        printf("\n--- Binary Search Tree Menu ---\n");
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Inorder Traversal\n");
        printf("4. Preorder Traversal\n");
        printf("5. Postorder Traversal\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                root = insert(root, value);
                break;

            case 2:
                printf("Enter value to delete: ");
                scanf("%d", &value);
                root = deleteNode(root, value);
                break;

            case 3:
                printf("Inorder Traversal: ");
                inorder(root);
                printf("\n");
                break;

            case 4:
                printf("Preorder Traversal: ");
                preorder(root);
                printf("\n");
                break;
```

```
            case 5:
                printf("Postorder Traversal: ");
                postorder(root);
                printf("\n");
                break;
            case 6:
                exit(0);
            default:
                printf("Invalid choice!\n");
        }  } while (choice != 6);
 return 0;
}
```

Output

```
1. Insert
2. Delete
3. Inorder Traversal
4. Preorder Traversal
5. Postorder Traversal
6. Exit
Enter your choice: 1
Enter value to insert: 50

Enter your choice: 1
Enter value to insert: 30

Enter your choice: 1
Enter value to insert: 70

Enter your choice: 3
Inorder Traversal: 30 50 70

Enter your choice: 4
Preorder Traversal: 50 30 70

Enter your choice: 5
Postorder Traversal: 30 70 50

Enter your choice: 2
Enter value to delete: 30

Enter your choice: 3
Inorder Traversal: 50 70

Enter your choice: 6
```

Experiment 21 **Date:28/11/2025**

Red Black Tree Operations

Aim:

Create Red Black Tree and perform the following operations.

1. Create
2. Insert a new node
3. Left rotate
4. Right rotate
5. Delete a node
6. Inorder traversal

Algorithm:

1. Start.
2. Create an empty Red-Black Tree.
3. Display menu
4. Read user choice.
5. If choice = 1 then
   Read value.
   Call insert(tree, value).
6. If choice = 2 then
   Read value.
   Call delete(tree, value).
7. If choice = 3 then
   Call inOrderTraversal(tree->root).
8. If choice = 4 then
   Call freeMemory(tree->root).
   Exit.
9. Else print "Wrong selection".
10. Repeat steps 3–9 until choice = 4.
11. Stop.

void insert ()

1. Create a new node with given value.
2. Insert the node as in Binary Search Tree.
3. Color the new node RED.

4. Call insertFixup() to maintain Red-Black properties.
5. Ensure root is colored BLACK.
6. Stop.

void insertFix-up ()

1. While parent of new node is RED:
2. Identify uncle node.
3. If uncle is RED:
   Recolor parent and uncle to BLACK.
   Recolor grandparent to RED.
4. Else:
   Perform appropriate rotation (left or right).
   Recolor nodes.
5. Set root color to BLACK.
6. Stop.

void left-rotate()

1. Start
2. Set y ← x.right
3. Set x.right ← y.left
4. If y.left ≠ NULL, then
   set y.left.parent ← x
5. Set y.parent ← x.parent
6. If x.parent = NULL, then
   set T.root ← y
7. Else if x is the left child of its parent, then
   set x.parent.left ← y
8. Else
   set x.parent.right ← y
9. Set y.left ← x
10. Set x.parent ← y
11. Stop

void right-rotate()

1. Set x ← y.left
2. Set y.left ← x.right
3. If x.right ≠ NULL, then
   set x.right.parent ← y
4. Set x.parent ← y.parent
5. If y.parent = NULL, then
   set T.root ← x
6. Else if y is the left child of its parent, then
   set y.parent.left ← x
7. Else
   set y.parent.right ← x

8. Set x.right ← y
9. Set y.parent ← x

void delete ()
12. Search the node to be deleted.
13. If node not found, display message and stop.
14. Delete node using BST deletion logic.
15. Store original color of deleted node.
16. If deleted node was BLACK:
    Call deleteFixup() to restore Red-Black properties.
17. Stop.

void deleteFix-up ()
1. While node is not root and color is BLACK:
2. Identify sibling node.
3. Perform recoloring and rotations based on sibling's color and children.
4. Restore Red-Black Tree properties.
5. Color root BLACK.
6. Stop.

void inOrderTraversal **()**
1. Traverse left subtree.
2. Display node value and color.
3. Traverse right subtree.
4. Stop.

void Free Memory ()
1. Traverse left subtree.
2. Traverse right subtree.
3. Free current node.
4. Stop.

Program

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

typedef struct Node {
   int data;
   struct Node* parent;
   struct Node* left;
   struct Node* right;
   int color; // 0 for black, 1 for red
} Node;
```

```
typedef struct RedBlackTree {
    Node* root;
} RedBlackTree;

Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Memory allocation error\n");
        exit(1);
    }
    newNode->data = data;
    newNode->parent = newNode->left = newNode->right = NULL;
    newNode->color = 1; // New nodes are initially red
    return newNode;
}

RedBlackTree* createRedBlackTree() {
    RedBlackTree* newTree = (RedBlackTree*)malloc(sizeof(RedBlackTree));
    if (newTree == NULL) {
        printf("Memory allocation error\n");
        exit(1);
    }
    newTree->root = NULL;
    return newTree;
}
void leftRotate(RedBlackTree* tree, Node* x) {
    Node* y = x->right;
    x->right = y->left;
    if (y->left != NULL)
        y->left->parent = x;
    y->parent = x->parent;
    if (x->parent == NULL)
        tree->root = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;
    y->left = x;
    x->parent = y;
}
void rightRotate(RedBlackTree* tree, Node* y) {
    Node* x = y->left;
    y->left = x->right;
    if (x->right != NULL)
        x->right->parent = y;
    x->parent = y->parent;
```

```
    if (y->parent == NULL)
        tree->root = x;
    else if (y == y->parent->left)
        y->parent->left = x;
    else
        y->parent->right = x;
    x->right = y;
    y->parent = x;
}
void insertFixup(RedBlackTree* tree, Node* z) {
    while (z->parent != NULL && z->parent->color == 1) {
        if (z->parent == z->parent->parent->left) {
            Node* y = z->parent->parent->right;
            if (y != NULL && y->color == 1) {
                z->parent->color = 0;
                y->color = 0;
                z->parent->parent->color = 1;
                z = z->parent->parent;
            } else {
                if (z == z->parent->right) {
                    z = z->parent;
              leftRotate(tree, z);
                }

                z->parent->color = 0; // Black
                z->parent->parent->color = 1; // Red
                rightRotate(tree, z->parent->parent);
            }
        } else {
            Node* y = z->parent->parent->left;
            if (y != NULL && y->color == 1) {
                z->parent->color = 0; // Black
                y->color = 0; // Black
                z->parent->parent->color = 1;
                z = z->parent->parent;
            } else {
                if (z == z->parent->left) {
                    z = z->parent;
                    rightRotate(tree, z);
                }
                z->parent->color = 0;
                z->parent->parent->color = 1;
                leftRotate(tree, z->parent->parent);
            }
        }
```

```
    }
    tree->root->color = 0;
}
void insert(RedBlackTree* tree, int data) {
    Node* z = createNode(data);
    Node* y = NULL;
    Node* x = tree->root;

    while (x != NULL) {
        y = x;
        if (z->data < x->data)
            x = x->left;
        else
            x = x->right;
    }
 z->parent = y;
 if (y == NULL){
        tree->root = z;
    else if (z->data < y->data)
        y->left = z;
    else
        y->right = z;

    insertFixup(tree, z);
}
Node* findMinValueNode(Node* node) {
    Node* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}
void deleteFixup(RedBlackTree* tree, Node* x) {
    while (x != tree->root && x->color == 0) {
        if (x == x->parent->left) {
            Node* w = x->parent->right;
            if (w->color == 1) {
                w->color = 0;
                x->parent->color = 1;
                leftRotate(tree, x->parent);
                w = x->parent->right;
            }
            if (w->left->color == 0 && w->right->color == 0) {
                w->color = 1;
                x = x->parent;
            } else {
                if (w->right->color == 0) {
```

```
                w->left->color = 0;
                w->color = 1;
                rightRotate(tree, w);
                w = x->parent->right;
            }
            w->color = x->parent->color;
            x->parent->color = 0;
            w->right->color = 0;
            leftRotate(tree, x->parent);
            x = tree->root;
        }
    } else {
        Node* w = x->parent->left;
        if (w->color == 1) {
            w->color = 0;
            x->parent->color = 1;
            rightRotate(tree, x->parent);
            w = x->parent->left;
        }
        if (w->right->color == 0 && w->left->color == 0) {
            w->color = 1;
            x = x->parent;
        } else {
            if (w->left->color == 0) {
                w->right->color = 0;
                w->color = 1;
                leftRotate(tree, w);
                w = x->parent->left;
            }
            w->color = x->parent->color;
            x->parent->color = 0;
            w->left->color = 0;
            rightRotate(tree, x->parent);
            x = tree->root;
        }
    }
    x->color = 0;
}

void transplant(RedBlackTree* tree, Node* u, Node* v) {
    if (u->parent == NULL) {
        tree->root = v;
    } else if (u == u->parent->left) {
        u->parent->left = v;
    } else {
```

```
        u->parent->right = v;
    }
    if (v != NULL) {
        v->parent = u->parent;
    }
}
void delete(RedBlackTree* tree, int data) {
    Node* z = tree->root;
    while (z != NULL && z->data != data) {
        if (data < z->data)
            z = z->left;
        else
            z = z->right;
    }

    if (z == NULL) {
        printf("Node not found in the tree\n");
        return;
    }

    Node* y = z;
    Node* x;
    int yOriginalColor = y->color;

    if (z->left == NULL) {
        x = z->right;
        transplant(tree, z, z->right);
    } else if (z->right == NULL) {
        x = z->left;
        transplant(tree, z, z->left);
    } else {
        y = findMinValueNode(z->right); subtree
        yOriginalColor = y->color;
        x = y->right;

        if (y->parent == z) {
            x->parent = y;
        } else {
            transplant(tree, y, y->right);
            y->right = z->right;
            y->right->parent = y;
        }

        transplant(tree, z, y);
        y->left = z->left;
```

```
        y->left->parent = y;
        y->color = z->color;
    }
 free(z);

    if (yOriginalColor == 0) {
        deleteFixup(tree, x);
    }
}
void inOrderTraversal(Node* root) {
    char c[2][6]={"BLACK","RED"};
    if (root != NULL) {
        inOrderTraversal(root->left);
        printf("%d,%s -> ", root->data, c[root->color]);
        inOrderTraversal(root->right);
    }
}
void freeMemory(Node* root) {
    if (root == NULL)
        return;
    freeMemory(root->left);
    freeMemory(root->right);
    free(root);
}

int main() {
    int choice,value;
    RedBlackTree* tree = createRedBlackTree();
    printf("\n1. Insertion\n2. Deletion\n3. Display\n4. Exit");
    do
    {

        printf("\n\nEnter your choice: ");
        scanf("%d",&choice);
        switch(choice)
        {
        case 1: printf("Enter the value to be insert: ");
                scanf("%d",&value);
                insert(tree, value);
                break;
        case 2: printf("Enter the value to be deleted: ");
                scanf("%d",&value);
                delete(tree, value);
                break;
```

```
            case 3: inOrderTraversal(tree->root);
                break;
             case 4: freeMemory(tree->root);
                break;
           default: printf("\nWrong selection!!! Try again!!!");
           }
    }while(choice!=4);
    return(0);
}
```

Output

```
1. Insertion
2. Deletion
3. Display
4. Exit

Enter your choice: 1
Enter the value to be insert: 1

Enter your choice: 1
Enter the value to be insert: 2

Enter your choice: 1
Enter the value to be insert: 3

Enter your choice: 3
1,RED -> 2,BLACK -> 3,RED ->

Enter your choice: 1
Enter the value to be insert: 4

Enter your choice: 1
Enter the value to be insert: 5

Enter your choice: 3
1,BLACK -> 2,BLACK -> 3,RED -> 4,BLACK -> 5,RED ->
```

```
Enter your choice: 2
Enter the value to be deleted: 3

Enter your choice: 3
1,BLACK -> 2,BLACK -> 4,BLACK -> 5,RED ->

Enter your choice: 2
Enter the value to be deleted: 5

Enter your choice: 3
1,BLACK -> 2,BLACK -> 4,BLACK ->

Enter your choice: 4
```

Experiment 22 **Date:28/11/2025**

B-Tree Operations

Aim:

Write a program to implement the following operation on B Tree

1. Creation
2. Insertion
3. Searching
4. Inorder traversal

INSERT(root, key)

1. Start
2. If root is full then
   Create a new node newRoot
3. Make old root its child
4. Split the child
5. Insert key into appropriate child
6. Update root
7. Else:
   Call INSERT-NON-FULL(root, key)
8. Stop

INSERT-NON-FULL(node, key)

1. Start
2. Set i = node.numKeys − 1
3. If node is a leaf then
   Shift keys to maintain order
   Insert key at correct position
   Increment numKeys
4. Else:
   Find child where key should go
   If that child is full then
   Split the child
   Decide which child to descend
   Recursively insert into child
5. Stop

SPLIT-CHILD(parent, index, child)

1. Start
2. Create a new node newChild
3. Move second half of child's keys to newChild
4. If child is not leaf, move corresponding children

5. Reduce number of keys in child
6. Move middle key up to parent
7. Attach newChild to parent
8. Stop

TRAVERSE(node)
1. Start
2. For each key in node:
3. Traverse left child
4. Print key
5. Traverse rightmost child
6. Stop

SEARCH(node, key)
1. Start
2. Set i = 0
3. Repeat while i < numKeys and key > node.keys[i]:
   Increment i
4. If key == node.keys[i] then
   Return node (key found)
5. If node is a leaf then
   Return NULL (key not found)
6. Else:
   Recursively search in children[i]
7. Stop

Program

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_KEYS 3 // Maximum keys in a node (t-1 where t is the minimum degree)
#define MIN_KEYS 1 // Minimum keys in a node (ceil(t/2) - 1)
#define MAX_CHILDREN (MAX_KEYS + 1) // Maximum children in a node (t)

typedef struct BTreeNode {
    int keys[MAX_KEYS];
    struct BTreeNode* children[MAX_CHILDREN];
    int numKeys;
    int isLeaf;
} BTreeNode;

BTreeNode* createNode(int isLeaf) {
```

```
BTreeNode* node = (BTreeNode*)malloc(sizeof(BTreeNode));
    node->isLeaf = isLeaf;
    node->numKeys = 0;
    for (int i = 0; i < MAX_CHILDREN; i++) {
        node->children[i] = NULL;
    }
    return node;
}
void splitChild(BTreeNode* parent, int index, BTreeNode* child) {
    BTreeNode* newChild = createNode(child->isLeaf);
    newChild->numKeys = MIN_KEYS;
    for (int i = 0; i < MIN_KEYS; i++) {
        newChild->keys[i] = child->keys[i + MIN_KEYS + 1];
    }

    if (!child->isLeaf) {
        for (int i = 0; i < MIN_KEYS + 1; i++) {
            newChild->children[i] = child->children[i + MIN_KEYS + 1];
        }
    }
    child->numKeys = MIN_KEYS;
    for (int i = parent->numKeys; i >= index + 1; i--) {
        parent->children[i + 1] = parent->children[i];
    }
    parent->children[index + 1] = newChild;
    for (int i = parent->numKeys - 1; i >= index; i--) {
        parent->keys[i + 1] = parent->keys[i];
    }

    parent->keys[index] = child->keys[MIN_KEYS];
    parent->numKeys++;
}
void insertNonFull(BTreeNode* node, int key) {
    int i = node->numKeys - 1;
    if (node->isLeaf) {
        while (i >= 0 && node->keys[i] > key) {
            node->keys[i + 1] = node->keys[i];
            i--;
        }
        node->keys[i + 1] = key;
        node->numKeys++;
    } else {
        while (i >= 0 && node->keys[i] > key) {
            i--;
        }
```

```
        if (node->children[i + 1]->numKeys == MAX_KEYS) {
            splitChild(node, i + 1, node->children[i + 1]);
            if (key > node->keys[i + 1]) {
                i++;
            }
        }
        insertNonFull(node->children[i + 1], key);
    }
}
void insert(BTreeNode** root, int key) {
    BTreeNode* r = *root;
    if (r->numKeys == MAX_KEYS) {
        BTreeNode* newRoot = createNode(0);
        newRoot->children[0] = r;
        splitChild(newRoot, 0, r);
        int i = 0;
        if (newRoot->keys[0] < key) {
            i++;
        }
        insertNonFull(newRoot->children[i], key);
        *root = newRoot;
    } else {
        insertNonFull(r, key);
    }
}
void traverse(BTreeNode* root) {
    if (root == NULL) return;
    int i;
    for (i = 0; i < root->numKeys; i++) {
        if (!root->isLeaf) {
            traverse(root->children[i]);
        }
        printf("%d ", root->keys[i]);
    }

    if (!root->isLeaf) {
        traverse(root->children[i]);
    }
}

BTreeNode* search(BTreeNode* root, int key) {
    int i = 0;

    // Find first key greater than or equal to key
    while (i < root->numKeys && key > root->keys[i]) {
        i++;
    }
```

```
// If key found
    if (i < root->numKeys && key == root->keys[i]) {
        return root;
    }

    // If leaf node, key not found
    if (root->isLeaf) {
        return NULL;
    }

    // Go to appropriate child
    return search(root->children[i], key);
}

int main() {
    BTreeNode* root = createNode(1);
    insert(&root, 10);
    insert(&root, 20);
    insert(&root, 5);
    insert(&root, 6);
    insert(&root, 12);
    insert(&root, 30);
    insert(&root, 7);
    insert(&root, 17);
    printf("Traversal of the constructed B-tree is:\n");
    traverse(root);
     int v;
     printf(“Enter a value to search:\n”);
      scanf(“%d”,&v);
    BTreeNode* result = search(root, v);
    if (result != NULL)
          printf("\nKey %d found in the B-tree",v);
    else
          printf("\nKey %d not found in the B-tree",v);
          return 0;
}
```

Output

```
Traversal of the constructed B-tree is:
5 6 7 10 12 17 20 30
Enter a value to search:
15
Key 15 not found in the B-tree
```

Experiment 23 **Date:01/12/2025**

Heap Data Structures

Aim:

Create a Max-Heap and Min-Heap from the array

Algorithm:

7. Start
8. Declare an integer array A
9. Initialize array A with given elements
10. Calculate the number of elements
    n ← size of A
11. MaxHeap[n]
12. MinHeap[n]
13. Copy elements of array A into MaxHeap
14. Copy elements of array A into MinHeap
15. Call BUILD-MAX-HEAP(MaxHeap, n)
16. Display elements of MaxHeap
17. Call BUILD-MIN-HEAP(MinHeap, n)
18. Display elements of MinHeap
19. Stop

BUILD-MAX-HEAP(A, n)

1. Start from the last non-leaf node: i = n/2 − 1
2. For each node i down to 0:
3. Call MAX-HEAPIFY(A, n, i)

MAX-HEAPIFY(A, n, i)

1. Start
2. Set largest = i
3. If left child exists and A[left] > A[largest], update largest
4. If right child exists and A[right] > A[largest], update largest
5. If largest ≠ i, swap A[i] and A[largest]
6. Recursively call MAX-HEAPIFY on largest

BUILD-MIN-HEAP(A, n)

1. Start from the last non-leaf node: i = n/2 − 1
2. For each node i down to 0:
3. Call MIN-HEAPIFY(A, n, i)
4. Stop

MIN-HEAPIFY(A, n, i)

1. Start
2. Set smallest = i
3. If left child exists and A[left] < A[smallest], update smallest
4. If right child exists and A[right] < A[smallest], update smallest
5. If smallest ≠ i, swap A[i] and A[smallest]
6. Recursively call MIN-HEAPIFY on smallest
7. Stop

Program

```
#include <stdio.h>
void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}
void maxHeapify(int arr[], int n, int i) {
   int largest = i;
   int left = 2*i + 1;
   int right = 2*i + 2;

   if (left < n && arr[left] > arr[largest])
      largest = left;

   if (right < n && arr[right] > arr[largest])
      largest = right;

   if (largest != i) {
      swap(&arr[i], &arr[largest]);
      maxHeapify(arr, n, largest);
   }
}

void minHeapify(int arr[], int n, int i) {
   int smallest = i;
   int left = 2*i + 1;
   int right = 2*i + 2;

   if (left < n && arr[left] < arr[smallest])
      smallest = left;

   if (right < n && arr[right] < arr[smallest])
      smallest = right;
```

```
    if (smallest != i) {
        swap(&arr[i], &arr[smallest]);
        minHeapify(arr, n, smallest);
    }
}

void buildMaxHeap(int arr[], int n) {
    for (int i = n/2 - 1; i >= 0; i--)
        maxHeapify(arr, n, i);
}
void buildMinHeap(int arr[], int n) {
    for (int i = n/2 - 1; i >= 0; i--)
        minHeapify(arr, n, i);
}

void printHeap(int arr[], int n) {
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {10, 3, 5, 2, 8};
    int n = sizeof(arr)/sizeof(arr[0]);

    int maxHeap[5], minHeap[5];
    for (int i = 0; i < n; i++) {
        maxHeap[i] = arr[i];
        minHeap[i] = arr[i];
    }

    buildMaxHeap(maxHeap, n);
    buildMinHeap(minHeap, n);

    printf("Max Heap: ");
    printHeap(maxHeap, n);

    printf("Min Heap: ");
    printHeap(minHeap, n);

    return 0;
}
```

Output:

Max Heap: 10 8 5 2 3
Min Heap: 2 3 5 10 8

Experiment 24 Date:01/12/2025

Implement BFS and DFS in undirected graph

Aim:

Write a program to implement BFS and DFS on a connected undirected graph

Algorithm:

1. Start.
2. Read number of vertices n.
3. Read adjacency matrix a[i][j] for the graph.
4. Display adjacency matrix.
5. Repeat:
   - Initialize all vertices as unvisited (vis[i] = 0).
   - Display menu: 1 → BFS, 2 → DFS.
   - Read choice and source vertex s.
   - If choice = 1 → call bfs(s,n).
   - If choice = 2 → call dfs(s,n).
   - Ask user if they want to continue.
6. Until user chooses to exit.
7. Stop.

BFS (Breadth-First Search)

1. Start.
2. Enqueue source vertex s.
3. Mark s as visited.
4. Repeat while queue is not empty:
   - Dequeue a vertex p.
   - Print p.
   - Repeat for each unvisited adjacent vertex i of p:
     - Enqueue i.
     - Mark i as visited.
5. For any unvisited vertex, call bfs recursively.
6. Stop.

DFS (Depth-First Search)

1. Start.
2. Push source vertex s onto stack.
3. Mark s as visited.
4. Repeat while stack is not empty:
   - Pop a vertex k.
   - Print k.

Repeat for each unvisited adjacent vertex i of k:
Push i onto stack.
Mark i as visited.

5. For any unvisited vertex, call dfs recursively.
6. Stop

Program

```
#include<stdio.h>
int q[20],top=-1,front=-1,rear=-1,a[20][20],vis[20],stack[20];
int dequeue();
void enqueue(int item);
void bfs(int s,int n);
void dfs(int s,int n);
void push(int item);
int pop();
void main()
{
int n,i,s,ch,j;
char c,dummy;
printf("ENTER THE NUMBER OF VERTICES :");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
printf("ENTER 1 IF %d HAS A NODE WITH %d ELSE 0 :",i,j);
scanf("%d",&a[i][j]);
}
}
printf("THE ADJACENCY MATRIX IS\n");
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
printf(" %d",a[i][j]);
}
printf("\n");
}
printf("\n1.B.F.S");
printf("\n2.D.F.S");
do
{
```

```
for(i=1;i<=n;i++)
vis[i]=0;
printf("\nENTER YOUR CHOICE:");
scanf("%d",&ch);
printf("ENTER THE SOURCE VERTEX :");
scanf("%d",&s);
switch(ch)
{
case 1:bfs(s,n);
break;
case 2:
dfs(s,n);
break;
}
printf("\nDO U WANT TO CONTINUE(Y/N) ?: ");
scanf("%c",&dummy);
scanf("%c",&c);
}while((c=='y')||(c=='Y'));
}
void bfs(int s,int n)
{
   int p,i;
   enqueue(s);
   vis[s]=1;
   p=dequeue();
   if(p!=0)
      printf(" %d",p);
   while(p!=0)
   {
      for(i=1;i<=n;i++)
         if((a[p][i]!=0)&&(vis[i]==0))
         {
            enqueue(i);  // Corrected here
            vis[i]=1;
         }
      p=dequeue();
      if(p!=0)
         printf(" %d ",p);
   }
   for(i=1;i<=n;i++)
      if(vis[i]==0)
         bfs(i,n);
}
```

```
void enqueue(int item)
{
if(rear==19)
printf("QUEUE FULL");
else
{
if(rear==-1)
{
q[++rear]=item;
front++;
}
else
q[++rear]=item;
}
}
int dequeue()
{
int k;
if((front>rear)||(front==-1))
return(0);
else
{
k=q[front++];
return(k);
}
}
void dfs(int s,int n)
{
int i,k;
push(s);
vis[s]=1;
k=pop();
if(k!=0)
printf(" %d ",k);
while(k!=0)
{
for(i=1;i<=n;i++)
if((a[k][i]!=0)&&(vis[i]==0))
{
push(i);
vis[i]=1;
}
```

```
k=pop();
if(k!=0)
printf(" %d ",k);
}
for(i=1;i<=n;i++)
if(vis[i]==0)
dfs(i,n);
}
void push(int item)
{
if(top==19)
printf("Stack overflow ");
else
stack[++top]=item;
}
int pop()
{
int k;
if(top==-1)
return(0);
else
{
k=stack[top--];
return(k);
}
}
```

Output

```
ENTER THE NUMBER OF VERTICES :3
ENTER 1 IF 1 HAS A NODE WITH 1 ELSE 0 :1
ENTER 1 IF 1 HAS A NODE WITH 2 ELSE 0 :0
ENTER 1 IF 1 HAS A NODE WITH 3 ELSE 0 :1
ENTER 1 IF 2 HAS A NODE WITH 1 ELSE 0 :0
ENTER 1 IF 2 HAS A NODE WITH 2 ELSE 0 :0
ENTER 1 IF 2 HAS A NODE WITH 3 ELSE 0 :1
ENTER 1 IF 3 HAS A NODE WITH 1 ELSE 0 :1
ENTER 1 IF 3 HAS A NODE WITH 2 ELSE 0 :1
ENTER 1 IF 3 HAS A NODE WITH 3 ELSE 0 :0
THE ADJACENCY MATRIX IS
 1 0 1
 0 0 1
 1 1 0
```

```
1.B.F.S
2.D.F.S
ENTER YOUR CHOICE:1
ENTER THE SOURCE VERTEX :1
 1 3  2
DO U WANT TO CONTINUE(Y/N) ?: y

ENTER YOUR CHOICE:2
ENTER THE SOURCE VERTEX :2
 2  3  1
 DO U WANT TO CONTINUE(Y/N) ?: n
```

Experiment 25 Date:03/12/2025

Implement Prim's Algorithm

Aim:

Program to implement Prim's Algorithm for finding the minimum cost spanning tree

Algorithm:

1. Start
2. b1.Read n
3. int i=1,i<=n
4. Repeat for j from 1 to n do
   read cost[i][j]
5. cost[i][j]==0
6. cost[i][j]=INF
7. visited[1]=1
8. Repeat while (no_edges<n-1) do
   min=INF
   a=0
   b=0
   i=1,i<=n
   if(visited[i]==1) then
   j=1,j<=n
   if (visited[j]==0 && cost[i][j]!=INF) then
   if (cost[i][j]<min) then
   min=cost[i][j];
   a=i
   b=j
   no_edges++
   visited[b]=1
   total_cost=total_cost+min
9. Print total cost
10. Stop

Program

```
#include<stdio.h>
#define INF 999
int cost[10][10],visited[10]={0,0,0,0,0,0,0,0,0,0};
int n,i,j,no_edges=0,total_cost=0,min,a,b;
int main()
{
printf("Enter the number of vertices : ");
scanf("%d",&n);
printf("Enter the cost adjacency matrix : \n");
for(int i=1;i<=n;i++)
   {
for(int j=1;j<=n;j++)
      {
 scanf("%d",&cost[i][j]);
          if (cost[i][j]==0)
          {
             cost[i][j]=INF;
          }
      }
   }
printf("\nThe minimum cost spanning tree edges are:\n");
visited[1]=1;
   while (no_edges<n-1)
   {
      min=INF;
      a=0;
      b=0;
      for(i=1;i<=n;i++)
      {
         if(visited[i]==1)
         {
            for (j=1;j<=n;j++)
            {
```

```
                if (visited[j]==0 && cost[i][j]!=INF)
                {
                   if (cost[i][j]<min)
                   {
                      min=cost[i][j];
                      a=i;
                      b=j;
                   }
                }
             }
          }
       }
no_edges++;
       visited[b]=1;
printf("%d-%d:%d\n",a,b,min);
total_cost=total_cost+min;
   }
printf("Total cost : %d\n",total_cost);
}
```

Output

Enter the number of vertices : 3

Enter the cost adjacency matrix :

0 1 2

3 2 0

4 3 1

The minimum cost spanning tree edges are:

1-2:1

1-3:2

Total cost : 3

Experiment 26 Date:03/12/2025

Implement Kruskal's Algorithm

Aim:

Program to implement Kruskal's algorithm using Disjoint sets.

Algorithm:

1. Start
2. Declare cost[10][10],visited[10]={0,0,0,0,0,0,0,0,0,0},parent[10]
3. Declare n,i,j,no_edges=0,total_cost=0,min,a,b,u,v
4. Read n
5. Set i=1,i<=n
6. Set j=1,j<=n
7. Read cost[i][j]
8. if (cost[i][j]==0) then
   cost[i][j]=INF
9. visited[1]=1
10. Repeat while (no_edges<n-1)
    min=INF
    a=0
    b=0
    Set i=1,i<=n
    Set j=1,j<=n
    if (cost[i][j]<min)
    min=cost[i][j]
    a=u=i
    b=v=j
    u=find(u)
    v=find(v)
    if(unit(u,v)) then
    Print a,b,min
    no_edges++
    total_cost=total_cost+min
    cost[a][b]=cost[b][a]=INF
11. Print total_cost
12. Stop

void find(int)

1. Start
2. Repeat while(parent[i]) do
   i=parent[i]
3. return i
4. Stop

void unit(int ,int)

1. Start
2. if(i!=j) then
   parent[j]=i
   return 1
3. return
4. Stop

Program

```
#include<stdio.h>
#define INF 999
int cost[10][10],visited[10]={0,0,0,0,0,0,0,0,0,0},parent[10];
int n,i,j,no_edges=0,total_cost=0,min,a,b,u,v;
int find(int);
int unit(int,int);
int main()
{
printf("Enter the no. of vertices : ");
scanf("%d",&n);
printf("Enter the adjacency matrix : \n");
for(int i=1;i<=n;i++)
   {
for(int j=1;j<=n;j++)
      {
scanf("%d",&cost[i][j]);
```

```
            if (cost[i][j]==0)
            {
               cost[i][j]=INF;
            }}}
printf("\nCost of edges\n");
visited[1]=1;
   while (no_edges<n-1)
   {
      min=INF;
      a=0;
      b=0;
      for(i=1;i<=n;i++)
      {
         for (j=1;j<=n;j++)
         {
            if (cost[i][j]<min)
            {
               min=cost[i][j];
               a=u=i;
               b=v=j;
            }
         }
      }
      u=find(u);
      v=find(v);
      if(unit(u,v))
      {
printf("%d-%d:%d\n",a,b,min);
no_edges++;
total_cost=total_cost+min;
      }
```

```
        cost[a][b]=cost[b][a]=INF;
    }
printf("Total cost : %d\n",total_cost);
}
int find(int i)
{
    while(parent[i])
i=parent[i];
    return i;
}
int unit(int i,int j)
{
    if(i!=j)
    {
        parent[j]=i;
        return 1;
    }
    return 0;
}
```

Output

```
Enter the no. of vertices : 3
Enter the adjacency matrix :
1 2 3
0 1 2
2 0 3

Cost of edges
1-2:2
2-3:2
Total cost : 4
```

Experiment 27 **Date:06/12/2025**

Implement Dijkstra's algorithm

Aim:

Program for single source shortest path algorithm using Dijkstras algorithm

Algorithm:

1. Start.
2. Define and initialize the adjacency matrix of the graph.
3. Select source vertex (0).
4. Call dijkstra(graph, source).
5. Display the shortest distance from source to all vertices.
6. Stop.

minDistance(int , bool)

1. Start
2. Set v = 0, v < V
3. if (sptSet[v] == false &&dist[v] <= min)
4. min = dist[v], min_index = v
5. return min_index
6. Stop

printSolution(dist[])

1. Start
2. Set i = 0,i< V
3. Print i, dist[i]
4. Stop

dijkstra(graph[V][V],src)

1. Start
2. declare dist[V]
3. declare sptSet[V]
4. declare i = 0; i< V; i++)
5. Set dist[i] = INT_MAX, sptSet[i] = false
6. Set dist[src] = 0
7. Set count = 0,count < V - 1
8. Set u = minDistance(dist, sptSet)
9. Set sptSet[u] = true

10. set v = 0,v< V
11. if !sptSet[v] && graph[u][v] then
12. Set dist[u] != INT_MAX
13. Set dist[u] + graph[u][v] <dist[v])
14. Set dist[v] = dist[u] + graph[u][v]
15. Print Solution(dist)
16. Stop

Program

```
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
#define V 9
int minDistance(int dist[], bool sptSet[])
{
        int min = INT_MAX, min_index;
        for (int v = 0; v < V; v++)
                if (sptSet[v] == false &&dist[v] <= min)
                        min = dist[v], min_index = v;
        return min_index;
}
void printSolution(int dist[])
{       printf("Vertex \t\t Distance from Source\n");
        for (int i = 0; i< V; i++)
                printf("%d \t\t\t\t %d\n", i, dist[i]);
}
void dijkstra(int graph[V][V], int src)
{
        int dist[V];
   bool sptSet[V];
        for (int i = 0; i< V; i++)
                dist[i] = INT_MAX, sptSet[i] = false;
        dist[src] = 0;
        for (int count = 0; count < V - 1; count++) {
                int u = minDistance(dist, sptSet);
                sptSet[u] = true;
                for (int v = 0; v < V; v++)
                        if (!sptSet[v] && graph[u][v]
                                &&dist[u] != INT_MAX
                                &&dist[u] + graph[u][v] <dist[v])
                                dist[v] = dist[u] + graph[u][v];
        }
        printSolution(dist);
}
```

```
int main()
{
        int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                            { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                            { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                            { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                            { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                            { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                            { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                            { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                            { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };
        dijkstra(graph, 0);
        return 0;
}
```

Output

Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14