ADVANCED DATA STRUCTURES (SE 313)

Practical File
(2023- 2024)

Submitted By
Sanoj (2K21/SE/159)

Under the Supervision of
Ms. Priya Singh

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)

Bawana Road, Delhi-110042

INDEX

S.No	Experiment Name	Date	Teacher Sign	Remarks
1.	Write a program to implement Huffman Coding using Priority Queue			
2.	Write a program to implement B Tree, Insertion, Deletion, and Traversal			
3.	Write a program to implement B+ Tree, Insertion, Deletion, and Traversal (Character data type)			
4.	Write a C++ program to perform the following operations for a Red-Black Tree (RBT) while ensuring that no property of the RedBlack Tree is violated			
5.	Write a C++ program to perform following operations for Interval Tree: Searching, Inserting, and Preorder Traversal.			
6.	Write a program to insert element in 2-3 Tree.			
7.	Write a program to detect if cycle is present or not using the concept of disjoint set.			
8.	Write a program to implement following operation on Binomial Heap: Make Heap, Insertion, Find Minimum element.			
9.	Write a program to implement Fibonacci Heap.			
10.	Write a program to count the total number of Spanning Trees for a given graph.			

SANOJ
2K21/SE/159

INDEX

S.No	Experiment Name	Date	Teacher Sign	Remarks
11.	Write a program to find all-pairs shortest path using a. Matrix Multiplication (dynamic programming paradigm) b. Floyd-Warshall algorithm			
12.	Write a program to count Cut Vertices in an undirected connected graph.			
13.	Write a program to identify if a given graph is Eulerian or not.			
14.	Check if a word occurs as a prefix of any word in a sentence using Trie.			

SANOJ
2K21/SE/159

EXPERIMENT 1 – HUFFMAN CODING

OBJECTIVE
Write a program to implement Huffman Coding using Priority Queue

INTRODUCTION
Huffman coding is a greedy approach based lossless data compression algorithm. It uses variable-length encoding to compress the data. The main idea in Huffman coding is to assign each character with a variable length code. The length of each character is decided on the basis of its occurrence frequency.

ALGORITHM
1. Push all the characters in **ch[]** mapped to corresponding frequency **freq[]** in priority queue.
2. To create Huffman Tree, pop two nodes from priority queue.
3. Assign two popped nodes from priority queue as left and right child of new node.
4. Push the new node formed in priority queue.
5. Repeat all above steps until size of priority queue becomes 1.
6. Traverse the Huffman Tree (whose root is the only node left in the priority queue) to store the Huffman Code
7. Print all the stored Huffman Code for every character in **ch[]**.

PROGRAM CODE
```
// C++ Program for Huffman Coding
#include <iostream>
#include <queue>
using namespace std;
#define MAX_SIZE 100

class HuffmanTreeNode {
public:
   char data;
   int freq;
   HuffmanTreeNode* left;
   HuffmanTreeNode* right;
   HuffmanTreeNode(char character, int frequency) {
      data = character;
      freq = frequency;
      left = right = NULL;
   }
};

class Compare {
public:
   bool operator()(HuffmanTreeNode* a, HuffmanTreeNode* b){
      return a->freq > b->freq;
   }
};
```

SANOJ
2K21/SE/159

```
HuffmanTreeNode* generateTree(priority_queue<HuffmanTreeNode*, vector<HuffmanTreeNode*>,
Compare> pq)
{
   while (pq.size() != 1) {
      HuffmanTreeNode* left = pq.top();
      pq.pop();
      HuffmanTreeNode* right = pq.top();
      pq.pop();
      HuffmanTreeNode* node = new HuffmanTreeNode('$', left->freq + right->freq);
      node->left = left;
      node->right = right;
      pq.push(node);
   }

   return pq.top();
}

void printCodes(HuffmanTreeNode* root, int arr[], int top) {

   if (root->left) {
      arr[top] = 0;
      printCodes(root->left,
            arr, top + 1);
   }
   if (root->right) {
      arr[top] = 1;
      printCodes(root->right, arr, top + 1);
   }
   if (!root->left && !root->right) {
      cout << root->data << " ";
      for (int i = 0; i < top; i++) {
         cout << arr[i];
      }
      cout << endl;
   }
}

void HuffmanCodes(char data[], int freq[], int size) {

   priority_queue<HuffmanTreeNode*, vector<HuffmanTreeNode*>, Compare> pq;
   for (int i = 0; i < size; i++) {
      HuffmanTreeNode* newNode
         = new HuffmanTreeNode(data[i], freq[i]);
      pq.push(newNode);
   }
   HuffmanTreeNode* root = generateTree(pq);
   int arr[MAX_SIZE], top = 0;
   printCodes(root, arr, top);
}
```

SANOJ
2K21/SE/159

```
int main()
{
   char data[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
   int freq[] = { 5, 9, 12, 13, 16, 45 };
   int size = sizeof(data) / sizeof(data[0]);

   HuffmanCodes(data, freq, size);
   return 0;
}
```

OUTPUT

```
Huffman Codes for the followings are
f 0
c 100
d 101
a 1100
b 1101
e 111
```

RESULT

As we have given in input that character ‘a’ is occurring 5 times, character ‘b’ is occurring 9 times and so on. So if we want to represent them in bit coding for them we have to take more than 600 bits which is a very larger number of bits. That why we use Huffman Coding it efficiently uses the bits number like, if ‘f’ is coming 45 times Huffman coding gives 1 bit to ‘f’ and so on. Using this we can decrease the number of bits.

LEARNING FROM EXPERIMENT

We can use Huffman coding in data compression. It can be used to compress all sorts of data, including text, images, and audio. It is beneficial for large compression of data. It can be used on those where there is series of occurring characters or something.

SANOJ
2K21/SE/159

EXPERIMENT 2 – B TREE IMPLEMENTATION

OBJECTIVE
Write a program to implement B Tree, Insertion, Deletion, and Traversal

INTRODUCTION
B-Trees maintain balance by ensuring that each node has a minimum number of keys, so the tree is always balanced. This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always O(log n), regardless of the initial shape of the tree.
Time Complexity: Search – O(log n), Insert – O(log n), Delete – O(log n)

ALGORITHM
Search:
- Start from the root and recursively traverse down.
- For every visited non-leaf node,
    - If the node has the key, we simply return the node.
    - Otherwise, we recur down to the appropriate child (The child which is just before the first greater key) of the node.
- If we reach a leaf node and don't find k in the leaf node, then return NULL.

Insertion:
- If the tree is empty, allocate a root node and insert the key.
- Update the allowed number of keys in the node.
- Search the appropriate node for insertion.
- If the node is full, follow the steps below.
- Insert the elements in increasing order.
- Now, there are elements greater than its limit. So, split at the median.
- Push the median key upwards and make the left keys as a left child and the right keys as a right child.
- If the node is not full, follow the steps below.
- Insert the node in increasing order.

Deletion:
- The key to be deleted lies in the leaf.
- If the key to be deleted lies in the internal node.
- If the target key lies in an internal node, and the deletion of the key leads to a fewer number of keys in the node (i.e. less than the minimum required), then look for the inorder predecessor and the inorder successor. If both the children contain a minimum number of keys then, borrowing cannot take place. This leads to Case II(3) i.e. merging the children.

PROGRAM CODE
```
// C++ Program for implementation of B Tree
#include<iostream>
using namespace std;
struct BTree {//node declaration
  int *d;
  BTree **child_ptr;
```

```
  bool l;
  int n;
}*r = NULL, *np = NULL, *x = NULL;

BTree* init() {//creation of node
  int i;
  np = new BTree;
  np->d = new int[6];//order 6
  np->child_ptr = new BTree *[7];
  np->l = true;
  np->n = 0;
  for (i = 0; i < 7; i++) {
    np->child_ptr[i] = NULL;
  }
  return np;
}

void traverse(BTree *p) { //traverse the tree
  cout<<endl;
  int i;
  for (i = 0; i < p->n; i++) {
    if (p->l == false) {
      traverse(p->child_ptr[i]);
    }
    cout << " " << p->d[i];
  }
  if (p->l == false) {
    traverse(p->child_ptr[i]);
  }
  cout<<endl;
}

void sort(int *p, int n){ //sort the tree
  int i, j, t;
  for (i = 0; i < n; i++) {
    for (j = i; j <= n; j++) {
      if (p[i] >p[j]) {
        t = p[i];
        p[i] = p[j];
        p[j] = t;
      }
    }
  }
}

int split_child(BTree *x, int i) {
  int j, mid;
  BTree *np1, *np3, *y;
  np3 = init();//create new node
  np3->l = true;
  if (i == -1) {
    mid = x->d[2];//find mid
    x->d[2] = 0;
```

SANOJ
2K21/SE/159

```
    x->n--;
    np1 = init();
    np1->l= false;
    x->l= true;
    for (j = 3; j < 6; j++) {
      np3->d[j - 3] = x->d[j];
      np3->child_ptr[j - 3] = x->child_ptr[j];
      np3->n++;
      x->d[j] = 0;
      x->n--;
    }
    for (j = 0; j < 6; j++) {
      x->child_ptr[j] = NULL;
    }
    np1->d[0] = mid;
    np1->child_ptr[np1->n] = x;
    np1->child_ptr[np1->n + 1] = np3;
    np1->n++;
    r = np1;
  } else {
    y = x->child_ptr[i];
    mid = y->d[2];
    y->d[2] = 0;
    y->n--;
    for (j = 3; j <6 ; j++) {
      np3->d[j - 3] = y->d[j];
      np3->n++;
      y->d[j] = 0;
      y->n--;
    }
    x->child_ptr[i + 1] = y;
    x->child_ptr[i + 1] = np3;
  }
  return mid;
}

void insert(int a) {
  int i, t;
  x = r;
  if (x == NULL) {
    r = init();
    x = r;
  } else {
    if (x->l== true && x->n == 6) {
      t = split_child(x, -1);
      x = r;
      for (i = 0; i < (x->n); i++) {
        if ((a >x->d[i]) && (a < x->d[i + 1])) {
          i++;
          break;
        } else if (a < x->d[0]) {
          break;
        } else {
```

```
            continue;
          }
        }
        x = x->child_ptr[i];
      } else {
        while (x->l == false) {
          for (i = 0; i < (x->n); i++) {
            if ((a >x->d[i]) && (a < x->d[i + 1])) {
              i++;
              break;
            } else if (a < x->d[0]) {
              break;
            } else {
              continue;
            }
          }
          if ((x->child_ptr[i])->n == 6) {
            t = split_child(x, i);
            x->d[x->n] = t;
            x->n++;
            continue;
          } else {
            x = x->child_ptr[i];
          }
        }
      }
    }
    x->d[x->n] = a;
    sort(x->d, x->n);
    x->n++;
}

int main() {
    int i, n, t;
    insert(10);
    insert(20);
    insert(30);
    insert(40);
    insert(50);
    cout<<"B tree:\n";
    traverse(r);
}
```

OUTPUT

```
B tree:

 10 20 30 40 50
```

SANOJ
2K21/SE/159

RESULT

B-Tree is a data structure that efficiently assists in data operations, including insertion, deletion, and traversal. The usefulness of data structure is evident not in small applications but is impactful in real-world situations involving huge datasets.

LEARNING FROM EXPERIMENT

- For each node x, the keys are stored in increasing order.
- In each node, there is a Boolean value x->leaf which is true if x is a leaf.
- If n is the order of the tree, each internal node can contain at most n - 1 keys along with a pointer to each child.
- Each node except root can have at most n children and at least n/2 children.
- All leaves have the same depth (i.e., height-h of the tree).
- The root has at least 2 children and contains a minimum of 1 key.
- If $n \geq 1$, then for any n-key B-tree of height h and minimum degree $t \geq 2$, $h \geq \log_t (n+1)/2$.

SANOJ
2K21/SE/159

EXPERIMENT 3 – B+ TREE IMPLEMENTATION

OBJECTIVE:
Write a program to implement B+ Tree, Insertion, Deletion, and Traversal (Character data type)

INTRODUCTION:
A B+ tree is a self-balancing tree data structure that is particularly useful for storing and managing large datasets efficiently. It is commonly used in databases and file systems due to its ability to maintain balance, perform range queries, and provide fast insertions and deletions.

Key Characteristics of B+ Trees:

- Balanced Nature: B+ trees maintain balance, ensuring that all leaf nodes are at the same level. This balance is essential for predictable and efficient performance.
- Sorted Data: Data in a B+ tree is organized in a sorted manner within leaf nodes. This allows for quick searching and range queries.
- Node Splitting and Merging: Similar to B-trees, B+ trees can split and merge nodes when inserting or deleting values to maintain balance.
- Leaf-Node Structure: In a B+ tree, all data is stored in leaf nodes, while non-leaf nodes are used for navigation.

Operations to be Implemented in a B+ Tree:

1. Insertion: The insertion operation involves adding a new character value to the B+ tree while adhering to the tree's properties. When an insertion causes a leaf node to exceed its maximum allowed size, the node is split, and the process continues recursively up the tree if necessary.
2. Deletion: Deletion in a B+ tree removes a specified character value while ensuring that the tree remains balanced. If a deletion operation results in a leaf node having too few elements, it can borrow from neighboring nodes or merge with them to maintain the tree's balance.
3. Traversal: Traversal of a B+ tree is typically performed in an in-order manner. In an inorder traversal, all leaf nodes are visited in ascending order. This operation allows examination of the contents of the B+ tree in a sorted sequence, which is valuable for various applications, including searching for specific values or printing the tree's contents.

ALGORITHM:
Insertion: To insert a character value into the B+ tree:

- Start from the root and recursively find the appropriate leaf node to insert the value.
- If the leaf node is not full, insert the value into the node in its correct position.
- If the leaf node is full, split it into two nodes, redistribute the values, and update the parent node accordingly.
- Continue this process until the root is reached, and if the root is also split, create a new root.

Deletion: To delete a character value from the B+ tree:

- Start from the root and recursively find the leaf node containing the value.
- If the value is found in a leaf node, simply remove it.
- Ensure that after deletion, the tree's properties are maintained. This may involve redistributing values among leaf nodes or merging leaf nodes if necessary.

SANOJ
2K21/SE/159

Traversal: To perform an in-order traversal of the B+ tree:

- Start from the leftmost leaf node and traverse leaf nodes in order.
- Visit the current leaf node and output its character values.
- Traverse to the next leaf node.
- Repeat this process until all leaf nodes have been visited.

CODE:

```
#include <iostream>
using namespace std;

class BPlusTreeNode {
        char *values;
         int t;
         BPlusTreeNode **pointers;
         int n;
         bool leaf;
public:
        BPlusTreeNode(int _t, bool _leaf);
         int findValue(char val);
         void remove(char val);
         void removeFromLeaf(int idx);
         void removeFromNonLeaf(int idx);
         char getPred(int idx);
         char getSucc(int idx);
         void fill(int idx);
         void borrowFromPrev(int idx);
         void borrowFromNext(int idx);
         void merge(int idx);
         void insertNonFull(char val);
         void splitChild(int i, BPlusTreeNode *y);
         void traverse();
         BPlusTreeNode *search(char val);
         friend class BPlusTree;
};

class BPlusTree {
        BPlusTreeNode *root;
         int t;
public:
         BPlusTree(int _t) {
                  root = nullptr;
                 t = _t;
         }
         void traverse() {
                  if (root != nullptr)
                  root->traverse();
         }
         BPlusTreeNode *search(char val) {
                  return (root == nullptr) ? nullptr : root->search(val);
         }
```

```
        void insert(char val);
        void remove(char val);
};

BPlusTreeNode::BPlusTreeNode(int t1, bool leaf1) {
        t = t1;
        leaf = leaf1;
        values = new char[2 * t - 1];
        pointers = new BPlusTreeNode *[2 * t];
        n = 0;
}

int BPlusTreeNode::findValue(char val) {
        int idx = 0;
        while (idx < n && values[idx] < val)
        idx++;
        return idx;
}

void BPlusTreeNode::remove(char val) {
        int idx = findValue(val);
        if (idx < n && values[idx] == val) {
                if (leaf)
                        removeFromLeaf(idx);
        else
                removeFromNonLeaf(idx);
        } else {
                if (leaf) {
                        cout << "The value " << val << " does not exist in the tree\n";
                        return;
                }
        bool flag = (idx == n);
        if (pointers[idx]->n < t)
        fill(idx);
        if (flag && idx > n)
        pointers[idx - 1]->remove(val);
        else
                pointers[idx]->remove(val);
        }
}

void BPlusTreeNode::removeFromLeaf(int idx) {
        for (int i = idx + 1; i < n; ++i)
        values[i - 1] = values[i];
        n--;
}

void BPlusTreeNode::removeFromNonLeaf(int idx) {
        char val = values[idx];
        if (pointers[idx]->n >= t) {
        char pred = getPred(idx);
        values[idx] = pred;
        pointers[idx]->remove(pred);
```

```
 } else if (pointers[idx + 1]->n >= t) {
          char succ = getSucc(idx);
           values[idx] = succ;
           pointers[idx + 1]->remove(succ);
} else {
           merge(idx);
           pointers[idx]->remove(val);
           }
}

char BPlusTreeNode::getPred(int idx) {
           BPlusTreeNode *cur = pointers[idx];
           while (!cur->leaf)
           cur = cur->pointers[cur->n];
           return cur->values[cur->n - 1];
}

char BPlusTreeNode::getSucc(int idx) {
           BPlusTreeNode *cur = pointers[idx + 1];
           while (!cur->leaf)
           cur = cur->pointers[0];
           return cur->values[0];
}

void BPlusTreeNode::fill(int idx) {
           if (idx != 0 && pointers[idx - 1]->n >= t)
           borrowFromPrev(idx);
           else if (idx != n && pointers[idx + 1]->n >= t)
           borrowFromNext(idx);
           else {
                     if (idx != n)
                     merge(idx);
           else
           merge(idx - 1);
          }
}

void BPlusTreeNode::borrowFromPrev(int idx) {
           BPlusTreeNode *child = pointers[idx];
           BPlusTreeNode *sibling = pointers[idx - 1];
           for (int i = child->n - 1; i >= 0; --i)
           child->values[i + 1] = child->values[i];
           child->values[0] = values[idx - 1];
           if (!child->leaf) {
                     for (int i = child->n; i >= 0; --i)
                     child->pointers[i + 1] = child->pointers[i];
          }
           values[idx - 1] = sibling->values[sibling->n - 1];
           child->n += 1;
           sibling->n -= 1;
}
```

SANOJ
2K21/SE/159

```
void BPlusTreeNode::borrowFromNext(int idx) {
	BPlusTreeNode *child = pointers[idx];
	BPlusTreeNode *sibling = pointers[idx + 1];
	child->values[(child->n)] = values[idx];
	if (!child->leaf)
		child->pointers[(child->n) + 1] = sibling->pointers[0];
		values[idx] = sibling->values[0];
		for (int i = 1; i < sibling->n; ++i)
		sibling->values[i - 1] = sibling->values[i];
		if (!sibling->leaf) {
			for (int i = 1; i <= sibling->n; ++i)
			sibling->pointers[i - 1] = sibling->pointers[i];
		}
	child->n += 1;
	sibling->n -= 1;
}

void BPlusTreeNode::merge(int idx) {
	BPlusTreeNode *child = pointers[idx];
	BPlusTreeNode *sibling = pointers[idx + 1];
	child->values[t - 1] = values[idx];
	for (int i = 0; i < sibling->n; ++i)
	child->values[i + t] = sibling->values[i];
	if (!child->leaf) {
		for (int i = 0; i <= sibling->n; ++i)
		child->pointers[i + t] = sibling->pointers[i];
	}
	for (int i = idx + 1; i < n; ++i)
	values[i - 1] = values[i];
	for (int i = idx + 2; i <= n; ++i)
	pointers[i - 1] = pointers[i];
	child->n += sibling->n + 1;
	n--;
	delete sibling;
}

void BPlusTreeNode::insertNonFull(char val) {
	int i = n - 1;
	if (leaf) {
		while (i >= 0 && values[i] > val) {
		values[i + 1] = values[i];
		i--;
	}
	values[i + 1] = val;
	n = n + 1;
	} else {
	while (i >= 0 && values[i] > val)
	i--;
	if (pointers[i + 1]->n == (2 * t - 1)) {
		splitChild(i + 1, pointers[i + 1]);
			if (values[i + 1] < val)
				i++;
	}
```

SANOJ
2K21/SE/159

```
            pointers[i + 1]->insertNonFull(val);
            }
}

void BPlusTreeNode::splitChild(int i, BPlusTreeNode *y) {
            BPlusTreeNode *z = new BPlusTreeNode(y->t, y->leaf);
            n++;
            for (int j = n - 1; j > i; j--) {
                        values[j] = values[j - 1];
                        pointers[j + 1] = pointers[j];
            }
            values[i] = y->values[t - 1];
            for (int j = t - 1; j < y->n - 1; j++)
                        z->values[j - t] = y->values[j];
                        if (!y->leaf) {
                                    for (int j = t; j <= y->n; j++)
                                    z->pointers[j - t] = y->pointers[j];
                        }
                        y->n = t - 1;
                        pointers[i + 1] = z;
}

void BPlusTreeNode::traverse() {
            int i;
            for (i = 0; i < n; i++) {
            if (!leaf)
            pointers[i]->traverse();
            cout << " " << values[i];
 }
            if (!leaf)
                        pointers[i]->traverse();
            }

BPlusTreeNode *BPlusTreeNode::search(char val) {
            int i = 0;
            while (i < n && val > values[i])
            i++;
            if (values[i] == val)
            return this;
            if (leaf)
                        return nullptr;
                        return pointers[i]->search(val);
            }

void BPlusTree::insert(char val) {
            if (root == nullptr) {
            root = new BPlusTreeNode(t, true);
            root->values[0] = val;
            root->n = 1;
            } else {
            if (root->n == (2 * t - 1)) {
                        BPlusTreeNode *s = new BPlusTreeNode(t, false);
            s->pointers[0] = root;
```

```
        s->splitChild(0, root);
        int i = 0;
        if (s->values[0] < val)
                i++;
                s->pointers[i]->insertNonFull(val);
                root = s;
        } else
                root->insertNonFull(val);
                }
}

void BPlusTree::remove(char val) {
        if (!root) {
        cout << "The tree is empty\n";
        return;
 }
        root->remove(val);
        if (root->n == 0) {
        BPlusTreeNode *tmp = root;
        if (root->leaf)
        root = nullptr;
        else
                root = root->pointers[0];
        delete tmp;
        }
}

int main() {
        BPlusTree t(3);
        char values[] = {'c', 'a', 'h', 'f', 'd', 'b', 'e', 'i', 'j', 'g'};
        int n = sizeof(values) / sizeof(values[0]);
        for (int i = 0; i < n; i++)
                t.insert(values[i]);
                cout << "Traversal of the constructed B+ tree: ";
                t.traverse();
                cout << endl;
                char val = 'f';
        (t.search(val) != nullptr) ? cout << "Present\n" : cout << "Not Present\n";
        val = 'k';
        (t.search(val) != nullptr) ? cout << "Present\n" : cout << "Not Present\n";
        val = 'b';
        t.remove(val);
        cout << "Traversal after removing " << val << ": ";
        t.traverse();
        cout << endl;
        val = 'h';
        t.remove(val);
        cout << "Traversal after removing " << val << ": ";
        t.traverse();
        cout << endl;
        return 0;
}
```

SANOJ
2K21/SE/159

OUTPUT:

```
/tmp/ACcfscswzn.o
Traversal of the constructed B+ tree:  a b c d e g i j
Not Present
Not Present
Traversal after removing b:  a c d e g i j
The value h does not exist in the tree
Traversal after removing h:  a c d e g i j
```

RESULT:

The provided C++ program successfully demonstrates the implementation of operations in a B+ tree with character data, including insertion, deletion, and traversal. The program constructs a B+ tree, inserts character values, deletes values, and performs an in-order traversal to display the tree's contents. The results of specific operations are also checked and displayed.

LEARNING:

This experiment on implementing operations in a B+ tree with character data provides several valuable learning outcomes:

1. **Understanding B+ Trees:** You gain a deeper understanding of B+ trees, a selfbalancing tree data structure used in various applications, including databases and file systems.
2. **Balancing Techniques:** You learn the techniques for maintaining balance in a B+ tree during insertions and deletions. These techniques involve node splitting, merging, and redistribution.
3. **Character Data Handling:** Handling character data in a tree structure enhances your skills in working with different data types, which is essential in real-world programming.
4. **Node Structure:** Understanding the structure of B+ tree nodes, including values and pointers, is fundamental for working with tree-based data structures.
5. **Recursive Algorithms:** The program's recursive structure for inserting and deleting values teaches you how to design and debug recursive algorithms, a valuable skill for solving complex problems.
6. **Efficient Data Structures:** B+ trees are known for their efficiency in handling large datasets. Learning to work with such structures equips you with the ability to design efficient data storage solutions.
7. **Search and Removal Operations:** You gain experience in searching for and removing specific values from a tree structure while maintaining its integrity.
8. **Error Handling:** The code handles various scenarios, such as node merging and borrowing, which enhances your ability to handle different cases and troubleshoot issues in your code.
9. **Traversing Tree Structures:** Implementing in-order traversal improves your understanding of tree traversal algorithms, which can be applied to various tree-based data structures.
10. **Code Optimization:** You practice writing efficient and organized code to handle insertions, deletions, and traversals in a B+ tree.
11. **Data Structure Design:** You gain insights into designing complex data structures that are essential for building robust software systems.
12. **Real-World Application:** B+ trees are commonly used in database systems and file storage, making this knowledge valuable for building efficient data management systems

SANOJ
2K21/SE/159

EXPERIMENT 4 – RED BLACK TREE IMPLEMENTATION

OBJECTIVE:

Write a C++ program to perform the following operations for a Red-Black Tree (RBT) while ensuring that no property of the RedBlack Tree is violated:

- Insertion: Insert new nodes into the Red-Black Tree while maintaining the Red-Black Tree properties.
- Finding Its Black Height: Calculate and display the black height (the number of black nodes on any path from the root to a leaf) of the Red-Black Tree.

INTRODUCTION:

A Red-Black Tree (RBT) is a self-balancing binary search tree where each node has a color (either red or black) and satisfies several properties to ensure its balance.

These properties include:

1. Each node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, both its children must be black (no two consecutive red nodes).
5. Every simple path from a node to a descendant leaf must have the same black height.

To maintain these properties during insertions and deletions, Red-Black Trees use various rotations and color changes.

ALGORITHM:

1. **Insertion:**
    - Insert a new node as in a standard binary search tree.
    - If the parent of the newly inserted node is red (violating the "no two consecutive red nodes" property), perform necessary rotations and color changes to restore the Red-Black Tree properties.
    - Continue up the tree until all properties are satisfied.

2. **Finding Its Black Height:**
    - Perform a depth-first traversal of the Red-Black Tree.
    - For each path from the root to a leaf, count the number of black nodes.
    - Return the minimum black height among all paths as the black height of the tree.

CODE:

```
#include <iostream>
#include <climits>
using namespace std;

enum Color { RED, BLACK };

class RBTreeNode {
public:
        int data;
         RBTreeNode* parent;
         RBTreeNode* left;
```

```
        RBTreeNode* right;
        Color color;
};

class RedBlackTree {
        private:
                RBTreeNode* root;
                RBTreeNode* TNULL;
        void findBlackHeightUtil(RBTreeNode* node, int currentBlackHeight, int& minBlackHeight) {
        if (node == TNULL) {
                if (currentBlackHeight < minBlackHeight) {
                        minBlackHeight = currentBlackHeight;
        }
        return;
}
        findBlackHeightUtil(node->left, currentBlackHeight + (node->color == BLACK ? 1 : 0),
minBlackHeight);
        findBlackHeightUtil(node->right, currentBlackHeight + (node->color == BLACK ? 1 : 0),
minBlackHeight);
}

void inOrderTraversal(RBTreeNode* node) {
        if (node == TNULL) {
        return;
        }
        inOrderTraversal(node->left);
        cout << node->data << " ";
        inOrderTraversal(node->right);
        }

void leftRotate(RBTreeNode* x) {
        RBTreeNode* y = x->right;
        x->right = y->left;
        if (y->left != TNULL) {
        y->left->parent = x;
}
        y->parent = x->parent;
        if (x->parent == nullptr) {
                this->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(RBTreeNode* y) {
        RBTreeNode* x = y->left;
        y->left = x->right;
        if (x->right != TNULL) {
                x->right->parent = y;
```

```
}
        x->parent = y->parent;
        if (y->parent == nullptr) {
                this->root = x;
        } else if (y == y->parent->left) {
                y->parent->left = x;
        } else {
                y->parent->right = x;
}
       x->right = y;
        y->parent = x;
}

void fixInsert(RBTreeNode* k) {
        RBTreeNode* u;
        while (k->parent->color == RED) {
        if (k->parent == k->parent->parent->right) {
        u = k->parent->parent->left;
        if (u->color == RED) {
                u->color = BLACK;
                k->parent->color = BLACK;
                k->parent->parent->color = RED;
                k = k->parent->parent;
        } else {
                if (k == k->parent->left) {
                        k = k->parent;
        rightRotate(k);
}
       k->parent->color = BLACK;
        k->parent->parent->color = RED;
        leftRotate(k->parent->parent);
}
        } else {
                u = k->parent->parent->right;
                if (u->color == RED) {
                        u->color = BLACK;
                        k->parent->color = BLACK;
                        k->parent->parent->color = RED;
                        k = k->parent->parent;
                } else {
        if (k == k->parent->right) {
                k = k->parent;
                leftRotate(k);
}
        k->parent->color = BLACK;
        k->parent->parent->color = RED;
        rightRotate(k->parent->parent);
        }
}
       if (k == root) {
                break;
        }
}
```

SANOJ
2K21/SE/159

```
        root->color = BLACK;
}

void insertNodeHelper(RBTreeNode* node) {
        RBTreeNode* x = root;
        RBTreeNode* y = nullptr;
        while (x != TNULL) {
                y = x;
                if (node->data < x->data) {
                        x = x->left;
                } else {
                        x = x->right;
                }
}
        node->parent = y;
        if (y == nullptr) {
        root = node;
        } else if (node->data < y->data) {
                y->left = node;
        } else {
                y->right = node;
        }
        if (node->parent == nullptr) {
                node->color = BLACK;
        } else if (node->parent->parent != nullptr) {
                fixInsert(node);
       }
}

public:
        RedBlackTree() {
        TNULL = new RBTreeNode;
        TNULL->color = BLACK;
        TNULL->left = nullptr;
        TNULL->right = nullptr;
        root = TNULL;
}

void insertNode(int key) {
        RBTreeNode* node = new RBTreeNode;
        node->parent = nullptr;
        node->data = key;
        node->left = TNULL;
        node->right = TNULL;
        node->color = RED;
        insertNodeHelper(node);
}
int findBlackHeight() {
        if (root == TNULL) {
        return 0;
        }
        int minBlackHeight = INT_MAX;
        int currentBlackHeight = 0;
```

```
        findBlackHeightUtil(root, currentBlackHeight, minBlackHeight);
        return minBlackHeight;
 }

 void printInOrder() {
        cout << "In-Order Traversal: ";
        inOrderTraversal(root);
        cout << endl;
        }
};

int main() {
        RedBlackTree rbt;
        rbt.insertNode(10);
        rbt.insertNode(20);
        rbt.insertNode(30);
        rbt.insertNode(40);
        rbt.insertNode(50);
        int blackHeight = rbt.findBlackHeight();
        cout << "Black Height of the Red-Black Tree: " << blackHeight << endl;
        rbt.printInOrder();
        return 0;
}
```

OUTPUT:

```
/tmp/ACcfscswzn.o
Black Height of the Red-Black Tree: 2
In-Order Traversal: 10 20 30 40 50
```

RESULT:

- The program inserts nodes into a Red-Black Tree while ensuring that all Red-Black. Tree properties are maintained, especially the properties related to node colors and balance.
- The program accurately calculates and displays the black height of the Red-Black Tree

LEARNING:

1. **Red-Black Trees**: Understanding the principles of Red-Black Trees, including their balancing rules and properties, is essential for maintaining balanced binary search trees.
2. **Insertion in Red-Black Trees**: The program demonstrates how to insert nodes into a Red-Black Tree while adhering to the Red-Black Tree properties. It includes cases for node rotations and color changes to ensure balance.
3. **Node Colour and Balancing**: Learning how to handle node colours (red and black) and apply rotations to maintain balance is a crucial skill when working with Red-Black Trees.

SANOJ
2K21/SE/159

EXPERIMENT 5 – INTERVAL TREE IMPLEMENTATION

OBJECTIVE:

Write a C++ program to perform the following operations for an Interval Tree: a. Inserting an interval into the Interval Tree while maintaining the properties of the Interval Tree. b. Searching for an interval within the Interval Tree. c. Performing a Preorder traversal of the available intervals in the Interval Tree.

INTRODUCTION:

An Interval Tree is a tree data structure used to hold intervals. Each node in the Interval Tree represents an interval. Interval Trees are mainly used for searching for overlapping intervals efficiently. The key feature of an Interval Tree is that it augments each node with the maximum endpoint value in its subtree, allowing for quick retrieval of overlapping intervals. This data structure is beneficial in various applications such as database systems and computational geometry.

ALGORITHM:

1. **Insertion:**

- Start at the root of the Interval Tree.
- Compare the low endpoint of the new interval with the low endpoint of the current node. If it is less, traverse to the left subtree; otherwise, traverse to the right subtree.
- Update the maximum endpoint of the current node if the new interval has a higher high endpoint.
- Recursively insert the interval in the appropriate position, maintaining the properties of a binary search tree.

2. **Search:**

- Begin the search at the root of the Interval Tree.
- Check if the current interval overlaps with the target interval. If it does, record or return the interval.
- Traverse the left subtree if the maximum endpoint of the left child is greater than or equal to the low endpoint of the target interval.
- Continue the search in the right subtree.

3. **Preorder Traversal:**

- Start the traversal at the root of the Interval Tree.
- Print the current interval.
- Recursively traverse the left subtree in a Preorder manner.
- Recursively traverse the right subtree in a Preorder manner.

SANOJ
2K21/SE/159

CODE:

```
#include <iostream>
using namespace std;
struct Interval {
        int low, high;
};
class IntervalTreeNode {
        public:
                IntervalTreeNode* left;
                IntervalTreeNode* right;
                Interval interval;
                int max;
                IntervalTreeNode(Interval i) : interval(i), max(i.high), left(nullptr),
        right(nullptr) {}
};
class IntervalTree {
        private:
         IntervalTreeNode* root;
         IntervalTreeNode* insertIntervalUtil(IntervalTreeNode* root, Interval i) {
         if (root == nullptr) {
                return new IntervalTreeNode(i);
         }
         int l = root->interval.low;
         if (i.low < l) {
                root->left = insertIntervalUtil(root->left, i);
         } else {
                root->right = insertIntervalUtil(root->right, i);
         }
         if (root->max < i.high) {
                root->max = i.high;
         }
         return root;
 }
 bool doOverlap(Interval i1, Interval i2) {
 if (i1.low <= i2.high && i2.low <= i1.high) {
                return true;
 }
        return false;
 }

 void searchOverlapIntervalsUtil(IntervalTreeNode* root, Interval i) {
         if (root == nullptr) {
         return;
 }
         if (doOverlap(root->interval, i)) {
         cout << "[" << root->interval.low << "," << root->interval.high << "] ";
```

```
}
        if (root->left != nullptr && root->left->max >= i.low) {
        searchOverlapIntervalsUtil(root->left, i);
}

        searchOverlapIntervalsUtil(root->right, i);
}
void preorderTraversalUtil(IntervalTreeNode* root) {
        if (root == nullptr) {
        return;
}
       cout << "[" << root->interval.low << "," << root->interval.high << "] ";
        preorderTraversalUtil(root->left);
        preorderTraversalUtil(root->right);
}
       public:
        IntervalTree() : root(nullptr) {}
        void insertInterval(Interval i) {
        root = insertIntervalUtil(root, i);
}
        void searchOverlapIntervals(Interval i) {
        searchOverlapIntervalsUtil(root, i);
        }
void preorderTraversal() {
        preorderTraversalUtil(root);
}
};
int main() {
IntervalTree it;
Interval intervals[] = { {15, 20}, {10, 30}, {17, 19}, {5, 20}, {12, 15}, {30, 40} };
int n = sizeof(intervals) / sizeof(intervals[0]);
for (int i = 0; i < n; i++) {
it.insertInterval(intervals[i]);
}
cout << "Intervals in the tree: ";
it.preorderTraversal();
cout << endl;
Interval searchInterval = {14, 16};
cout << "Intervals that overlap with [" << searchInterval.low << "," << searchInterval.high
<< "]: ";
it.searchOverlapIntervals(searchInterval);
cout << endl;
return 0;
}
```

SANOJ
2K21/SE/159

OUTPUT:

```
/tmp/r3MMAJHwKP.o
Intervals in the tree: [15,20] [10,30] [5,20] [12,15] [17,19] [30,40]
Intervals that overlap with [14,16]: [15,20] [10,30] [5,20] [12,15]
```

RESULT:

- The program inserts intervals into the Interval Tree while maintaining the properties of the Interval Tree, including the updates of the maximum endpoint values.
- It accurately searches for intervals that overlap with a target interval and performs a Preorder traversal of the available intervals in the Interval Tree.

LEARNING:

1. Interval Trees: Understanding the structure and functionality of Interval Trees is crucial for efficiently managing and searching intervals in various applications.
2. Interval Insertion: Implementing interval insertion in the Interval Tree while updating the maximum endpoint values facilitates efficient interval management within the tree.
3. Overlapping Intervals: Learning to identify and retrieve intervals that overlap with a given interval aids in effective interval searching and processing.
4. Preorder Traversal: Implementing a Preorder traversal of the Interval Tree enables the retrieval and processing of intervals in a specific order.
5. Application in Computational Geometry: Interval Trees find applications in computational geometry and database systems where efficient interval management and searching are vital.
6. Binary Search Tree Operations: Understanding and implementing operations within a binary search tree are essential for building specialized tree data structures such as Interval Trees.
7. Efficient Tree Operations: Writing code that efficiently handles tree operations aids in the effective management of large datasets and improves overall program performance.
8. Data Structure Augmentation: Augmenting tree nodes with additional information, such as maximum endpoint values, enables faster retrieval of relevant data within the tree.
9. Interval Management in Databases: Understanding Interval Trees is vital for the efficient management and querying of temporal data in database systems.

SANOJ
2K21/SE/159

EXPERIMENT 6 – 2 -3 TREE IMPLEMENTATION

OBJECTIVE:
Write a program to insert elements in 2-3 Trees.

INTRODUCTION:
A 2-3 tree is a type of self-balancing search tree where each node can have either two or three children. It is used to maintain a sorted set of elements, and it ensures that the tree remains balanced, which leads to efficient insertion, deletion, and search operations.

ALGORITHM:
1. **Search for the appropriate leaf node**: Start at the root and traverse down the tree to find the leaf node where the new element should be inserted. During this traversal, follow these rules:

- If you reach a node with two elements, choose the appropriate child subtree based on the element values.

2. **Insert the element in the leaf node:**

- If the leaf node has only one element, simply add the new element to it.
- If the leaf node already has two elements, you need to split it:
    - Insert the new element in the appropriate position within the node (maintaining the order of elements).
    - Create a new node if necessary, containing the middle element.
    - Adjust the tree structure to maintain the 2-3 tree properties:
        - If the parent of the current node has only one element, insert the middle element into it.
        - If the parent already has two elements, recursively split it, following the same procedure.

3. **Balancing the tree:**

- If a node at any level has three elements after insertion, split it and propagate the middle element to its parent node. Continue this process until you reach the root or a node that doesn't have three elements.

4. **Update the root if necessary:**

- If the root was split, create a new root with the middle element and its children.

CODE:
```
// C++ program for B-Tree insertion
#include<iostream>
using namespace std;

// A BTree node
class BTreeNode
{
	int *keys; // An array of keys
	int t;	// Minimum degree (defines the range for number of keys)
	BTreeNode **C; // An array of child pointers
```

SANOJ
2K21/SE/159

```
        int n;     // Current number of keys
        bool leaf; // Is true when node is leaf. Otherwise false
public:
        BTreeNode(int _t, bool _leaf); // Constructor

        // A utility function to insert a new key in the subtree rooted with
        // this node. The assumption is, the node must be non-full when this
        // function is called
        void insertNonFull(int k);

        // A utility function to split the child y of this node. i is index of y in
        // child array C[]. The Child y must be full when this function is called
        void splitChild(int i, BTreeNode *y);

        // A function to traverse all nodes in a subtree rooted with this node
        void traverse();

        // A function to search a key in the subtree rooted with this node.
        BTreeNode *search(int k); // returns NULL if k is not present.

// Make BTree friend of this so that we can access private members of this
// class in BTree functions
friend class BTree;
};

// A BTree
class BTree
{
        BTreeNode *root; // Pointer to root node
        int t; // Minimum degree
public:
        // Constructor (Initializes tree as empty)
        BTree()
        { root = NULL; t = 3; }

        // function to traverse the tree
        void traverse()
        { if (root != NULL) root->traverse(); }

        // function to search a key in this tree
        BTreeNode* search(int k)
        { return (root == NULL)? NULL : root->search(k); }

        // The main function that inserts a new key in this B-Tree
        void insert(int k);
};

// Constructor for BTreeNode class
BTreeNode::BTreeNode(int t1, bool leaf1)
{
        // Copy the given minimum degree and leaf property
        t = t1;
        leaf = leaf1;
```

SANOJ
2K21/SE/159

```
        // Allocate memory for maximum number of possible keys
        // and child pointers
        keys = new int[2*t-1];
        C = new BTreeNode *[2*t];

        // Initialize the number of keys as 0
        n = 0;
}

// Function to traverse all nodes in a subtree rooted with this node
void BTreeNode::traverse()
{
        // There are n keys and n+1 children, traverse through n keys
        // and first n children
        int i;
        for (i = 0; i < n; i++)
        {
                // If this is not leaf, then before printing key[i],
                // traverse the subtree rooted with child C[i].
                if (leaf == false)
                        C[i]->traverse();
                cout << " " << keys[i];
        }

        // Print the subtree rooted with last child
        if (leaf == false)
                C[i]->traverse();
}

// Function to search key k in subtree rooted with this node
BTreeNode *BTreeNode::search(int k)
{
        // Find the first key greater than or equal to k
        int i = 0;
        while (i < n && k > keys[i])
                i++;

        // If the found key is equal to k, return this node
        if (keys[i] == k)
                return this;

        // If key is not found here and this is a leaf node
        if (leaf == true)
                return NULL;

        // Go to the appropriate child
        return C[i]->search(k);
}

// The main function that inserts a new key in this B-Tree
void BTree::insert(int k)
{
```

```
	// If tree is empty
	if (root == NULL)
	{
		// Allocate memory for root
		root = new BTreeNode(t, true);
		root->keys[0] = k; // Insert key
		root->n = 1; // Update number of keys in root
	}
	else // If tree is not empty
	{
		// If root is full, then tree grows in height
		if (root->n == 2*t-1)
		{
			// Allocate memory for new root
			BTreeNode *s = new BTreeNode(t, false);

			// Make old root as child of new root
			s->C[0] = root;

			// Split the old root and move 1 key to the new root
			s->splitChild(0, root);

			// New root has two children now. Decide which of the
			// two children is going to have new key
			int i = 0;
			if (s->keys[0] < k)
				i++;
			s->C[i]->insertNonFull(k);

			// Change root
			root = s;
		}
		else // If root is not full, call insertNonFull for root
			root->insertNonFull(k);
	}
}

// A utility function to insert a new key in this node
// The assumption is, the node must be non-full when this
// function is called
void BTreeNode::insertNonFull(int k)
{
	// Initialize index as index of rightmost element
	int i = n-1;

	// If this is a leaf node
	if (leaf == true)
	{
		// The following loop does two things
		// a) Finds the location of new key to be inserted
		// b) Moves all greater keys to one place ahead
		while (i >= 0 && keys[i] > k)
		{
```

```
                keys[i+1] = keys[i];
                i--;
            }

            // Insert the new key at found location
            keys[i+1] = k;
            n = n+1;
        }
        else // If this node is not leaf
        {
            // Find the child which is going to have the new key
            while (i >= 0 && keys[i] > k)
                i--;

            // See if the found child is full
            if (C[i+1]->n == 2*t-1)
            {
                // If the child is full, then split it
                splitChild(i+1, C[i+1]);

                // After split, the middle key of C[i] goes up and
                // C[i] is splitted into two. See which of the two
                // is going to have the new key
                if (keys[i+1] < k)
                    i++;
            }
            C[i+1]->insertNonFull(k);
        }
}

// A utility function to split the child y of this node
// Note that y must be full when this function is called
void BTreeNode::splitChild(int i, BTreeNode *y)
{
        // Create a new node which is going to store (t-1) keys
        // of y
        BTreeNode *z = new BTreeNode(y->t, y->leaf);
        z->n = t - 1;

        // Copy the last (t-1) keys of y to z
        for (int j = 0; j < t-1; j++)
            z->keys[j] = y->keys[j+t];

        // Copy the last t children of y to z
        if (y->leaf == false)
        {
            for (int j = 0; j < t; j++)
                z->C[j] = y->C[j+t];
        }

        // Reduce the number of keys in y
        y->n = t - 1;
```

SANOJ
2K21/SE/159

```
        // Since this node is going to have a new child,
        // create space of new child
        for (int j = n; j >= i+1; j--)
                C[j+1] = C[j];

        // Link the new child to this node
        C[i+1] = z;

        // A key of y will move to this node. Find the location of
        // new key and move all greater keys one space ahead
        for (int j = n-1; j >= i; j--)
                keys[j+1] = keys[j];

        // Copy the middle key of y to this node
        keys[i] = y->keys[t-1];

        // Increment count of keys in this node
        n = n + 1;
}

// Driver program to test above functions
int main()
{
        BTree t; // A B-Tree with minimum degree 3
        t.insert(10);
        t.insert(20);
        t.insert(5);
        t.insert(6);
        t.insert(12);
        t.insert(30);
        t.insert(7);
        t.insert(17);

        cout << "Traversal of the constructed tree is ";
        t.traverse();

        return 0;
}
```

OUTPUT:

```
/tmp/HiRtJayZtq.o
Traversal of the constructed tree is  5 6 7 10 12 17 20 30
```

SANOJ
2K21/SE/159

RESULT:

1. **Balanced Structure**: A 2-3 tree is a self-balancing tree, which means that after every insertion, the tree rearranges itself to ensure that it remains balanced. The balancing involves splitting nodes and redistributing elements as needed.
2. **Sorted Order**: A key feature of a 2-3 tree is that it keeps its elements in sorted order. When you insert elements, the tree ensures that they are placed in the correct position within the tree to maintain this order.
3. **Efficient Operations**: The balanced structure of a 2-3 tree ensures that search, insertion, and deletion operations have a time complexity of O(log n), making it efficient for managing sorted data.
4. **No Overflows**: The tree avoids overflowing nodes by splitting them when they contain too many elements (three elements in the case of 2-3 trees). This splitting maintains the integrity of the tree structure.

LEARNING:

1. **Balanced Search Trees**: 2-3 trees are an example of self-balancing search trees, which ensure that the height of the tree remains logarithmic, leading to efficient search and insertion operations. Learning about self-balancing trees provides insight into how to maintain data structures for optimal performance.
2. **Sorted Data**: 2-3 trees maintain elements in sorted order. This is crucial for scenarios where you need to keep data sorted, such as dictionaries, databases, or indexing systems.
3. **Tree Node Splitting**: Understanding how nodes are split during insertions in 2-3 trees is essential. When a node contains too many elements, it is divided into multiple nodes to maintain the balance of the tree. This concept can be applied to other self-balancing trees like AVL and Red-Black trees.
4. **Recursive Algorithms**: Implementing 2-3 trees involves recursive algorithms to traverse and manipulate the tree structure. Learning to work with recursive algorithms can be valuable for various programming tasks.
5. **Efficiency vs. Simplicity Trade-offs**: While 2-3 trees offer efficient operations, they are more complex to implement compared to simpler data structures like arrays or linked lists. Learning to choose the right data structure for a given problem involves weighing trade-offs between simplicity and efficiency.
6. **Data Structure Design**: The implementation of a 2-3 tree highlights the importance of designing data structures to meet specific requirements. Choosing the appropriate data structure for a problem is a fundamental skill in computer science.
7. **Debugging and Testing**: When working with complex data structures like 2-3 trees, you may encounter bugs and issues. Learning to debug and test your code is essential for maintaining code quality and correctness.
8. **Maintaining Code Quality**: This project underscores the significance of writing clean, well-structured, and organized code to avoid errors and facilitate maintenance.

SANOJ
2K21/SE/159

EXPERIMENT 7 – DISJOINT SET IMPLEMENTATION

OBJECTIVE:
Write a program to detect if cycle is present in the graph or not using concept of disjoint set.

INTRODUCTION:
The objective of the program is to determine whether there is a closed loop or cycle within an undirected graph. It achieves this by using the disjoint set (union-find) data structure to efficiently track and merge the nodes in the graph, checking for cycles in the process. If a cycle is found, the program reports its presence; otherwise, it indicates that no cycles exist. This program is valuable for applications involving graph analysis and connectivity, including network routing and algorithm design.

ALGORITHM:
1. Create a Disjoint Set (DSU) data structure with a function to initialize and manage disjoint sets. Each node in the graph will initially be in its own set.
2. Iterate through the edges of the graph:
    i. For each edge (u, v), find the representative (root) of the set containing node u and the representative of the set containing node v. You can use the **find** operation of the DSU for this.
    ii. If the representatives of u and v are the same, it means both nodes are already part of the same set, and adding the edge would create a cycle. In this case, return "Cycle found."
    iii. If the representatives of u and v are different, merge the two sets by using the **union** operation of the DSU. This represents connecting node u and node v.
3. If you finish processing all the edges without finding a cycle (i.e., no "Cycle found" condition is triggered), return "No cycle found."

Here's a summary of the algorithm:
- Initialize disjoint sets, each containing a single node.
- Iterate through the edges, checking for cycles:
    - If the representatives of the two nodes in an edge are the same, a cycle is found, and the algorithm stops.
    - Otherwise, merge the sets by updating their representatives.
- If the iteration completes without finding a cycle, there are no cycles in the graph.

CODE:
```
#include <iostream>
#include <vector>
using namespace std;
class Graph {
public:
   int vertices;
   vector<pair<int, int> > edges;

   Graph(int V) : vertices(V) {}

   void addEdge(int u, int v) {
      edges.push_back(make_pair(u, v));
   }
```

```
    int findParent(vector<int>& parent, int node) {
        if (parent[node] == -1) {
            return node;
        }
        return findParent(parent, parent[node]);
    }

    void unionSets(vector<int>& parent, int x, int y) {
        int xSet = findParent(parent, x);
        int ySet = findParent(parent, y);
        parent[xSet] = ySet;
    }

    bool isCyclic() {
        vector<int> parent(vertices, -1);

        for (auto edge : edges) {
            int x = findParent(parent, edge.first);
            int y = findParent(parent, edge.second);

            if (x == y) {
                return true; // Cycle detected
            }

            unionSets(parent, x, y);
        }

        return false; // No cycle detected
    }
};

int main() {
    Graph graph(4);
    graph.addEdge(0, 1);
    graph.addEdge(1, 2);
    graph.addEdge(2, 3);
    graph.addEdge(3, 0);

    if (graph.isCyclic()) {
        std::cout << "Graph contains a cycle." << std::endl;
    } else {
        std::cout << "Graph does not contain a cycle." << std::endl;
    }

    return 0;
}
```

SANOJ
2K21/SE/159

OUTPUT:

```
/tmp/HiRtJayZtq.o
Edges are 0-1, 1-2, 2-3, 3-0
Graph contains a cycle.
```

RESULT:

1. It initializes a disjoint set data structure with a certain number of vertices.
2. It processes the edges of the graph and checks for the presence of cycles by using the union-find algorithm.
3. If a cycle is found during the processing of edges, the code prints "Cycle is present in the graph."
4. If no cycle is detected after processing all the edges, the code prints "No cycle is present in the graph."

The primary purpose of the code is to determine the presence or absence of cycles in the graph using the union-find algorithm.

LEARNING:

1. **Understanding Graph Cycles:** The program demonstrates how to identify cycles in an undirected graph. Learning how to detect cycles is fundamental in graph theory and can be applied to various real-world problems, such as identifying dependencies, avoiding deadlocks, and more.
2. **Disjoint Set Data Structure:** The program introduces the concept of disjoint sets and how to use them to efficiently manage and analyze connectivity in a graph. Learning about data structures like disjoint sets is crucial for various graph-related algorithms and data analysis tasks.
3. **Algorithmic Thinking:** This program involves algorithmic thinking by iterating through the edges of a graph, tracking connectivity, and making decisions based on the properties of the data structure. It helps improve problem-solving skills and algorithm design.
4. **Conditional Statements:** The use of conditional statements (if statements) to determine whether a cycle is present or not enhances your understanding of control flow in programming.
5. **Graph Representation:** The program uses a simple representation of a graph as a list of edges, making it easy to work with graphs for various applications.
6. **Practical Application:** Learning to detect cycles in graphs has practical applications in computer science, data science, and network theory. You can use this knowledge for tasks such as network routing, topological sorting, and constraint resolution.
7. **Testing and Debugging:** Experimenting with this code allows you to understand the importance of testing and debugging. You can test the code with different graphs to see how it performs and verify its correctness.
8. **Code Reusability:** The code can be used as a foundation for more complex graph algorithms, and it showcases the principles of modularity and code reuse.

SANOJ
2K21/SE/159

EXPERIMENT 8 – BINOMIAL HEAP IMPLEMENTATION

OBJECTIVE:
Write a program to implement following operation on Binomial Heap:
- Make_heap
- Insert_element
- Find_min

INTRODUCTION:
A Binomial Heap is a specialized form of a priority queue that is particularly efficient for these operations. The **Make_heap** operation initializes a Binomial Heap, the **Insert_element** operation adds elements to the heap, and the **Find_min** operation retrieves the minimum element from the heap.

ALGORITHM:
1. **Make_heap Algorithm**:
    - Create an empty Binomial Heap.
    - Initialize the **head** pointer to **nullptr**.
2. **Insert_element Algorithm**:
    - Create a new Binomial Heap with a single element (node).
    - Merge this new heap with the existing heap.
    - Update the **head** pointer to the merged heap.
3. **Find_min Algorithm**:
    - Initialize a variable **minValue** to positive infinity.
    - Traverse through the list of binomial trees in the heap.
    - For each tree, find the minimum value among its nodes.
    - If the minimum value of the current tree is less than **minValue**, update **minValue**.
    - Return **minValue** as the minimum value in the Binomial Heap.

CODE:
```
#include <iostream>
#include <vector>
#include<bits/stdc++.h>

// Structure to represent a node in the Binomial Heap
struct Node {
   int key;
   int degree;
   Node* parent;
   Node* child;
   Node* sibling;
};

class BinomialHeap {
private:
   Node* head;

   Node* mergeTrees(Node* tree1, Node* tree2) {
```

```
    if (tree1->key > tree2->key) {
        std::swap(tree1, tree2);
    }

    tree2->sibling = tree1->child;
    tree1->child = tree2;
    tree1->degree++;
    return tree1;
}

void mergeHeap(BinomialHeap& otherHeap) {
    Node* newHead = nullptr;
    Node* currentNode = nullptr;
    Node* tree1 = head;
    Node* tree2 = otherHeap.head;

    while (tree1 || tree2) {
        if (tree1 && (!tree2 || tree1->degree <= tree2->degree)) {
            if (currentNode) {
                currentNode->sibling = tree1;
            } else {
                newHead = tree1;
            }
            currentNode = tree1;
            tree1 = tree1->sibling;
        } else {
            if (currentNode) {
                currentNode->sibling = tree2;
            } else {
                newHead = tree2;
            }
            currentNode = tree2;
            tree2 = tree2->sibling;
        }
    }

    head = newHead;
}

void consolidate() {
    int maxDegree = -1;
    Node* prevNode = nullptr;
    Node* currentNode = head;
    Node* nextNode = head->sibling;
    std::vector<Node*> degreeArray(64, nullptr);

    while (currentNode) {
        int degree = currentNode->degree;
        while (degreeArray[degree]) {
            Node* other = degreeArray[degree];
            if (currentNode->key > other->key) {
                std::swap(currentNode, other);
            }
```

```
            degreeArray[degree] = nullptr;
            currentNode = mergeTrees(currentNode, other);
            degree++;
        }
        degreeArray[degree] = currentNode;
        maxDegree = std::max(maxDegree, degree);
        prevNode = currentNode;
        currentNode = nextNode;
        if (nextNode) {
            nextNode = nextNode->sibling;
        }
    }

    head = nullptr;
    for (int i = 0; i <= maxDegree; i++) {
        if (degreeArray[i]) {
            degreeArray[i]->sibling = nullptr;
            if (!head) {
                head = degreeArray[i];
            } else {
                prevNode->sibling = degreeArray[i];
            }
            prevNode = degreeArray[i];
        }
    }
}

public:
    BinomialHeap() : head(nullptr) {}

    // Function to insert an element into the Binomial Heap
    void insertElement(int key) {
        BinomialHeap newHeap;
        Node* newNode = new Node{key, 0, nullptr, nullptr, nullptr};
        newHeap.head = newNode;
        mergeHeap(newHeap);
    }

    // Function to find the minimum element in the Binomial Heap
    int findMin() {
        if (!head) {
            return INT_MAX; // If the heap is empty
        }

        Node* minNode = head;
        Node* currentNode = head;

        while (currentNode) {
            if (currentNode->key < minNode->key) {
                minNode = currentNode;
            }
            currentNode = currentNode->sibling;
        }
```

```
        return minNode->key;
    }
};

int main() {
    BinomialHeap heap;

    heap.insertElement(5);
    heap.insertElement(3);
    heap.insertElement(8);
    std::cout<<"Element of the heaps are 5, 3, 8"<<std::endl;
    std::cout << "Minimum element in the heap: " << heap.findMin() << std::endl;

    return 0;
}
```

OUTPUT:

```
/tmp/HiRtJayZtq.o
Element of the heaps are 5, 3, 8
Minimum element in the heap: 3
```

RESULT:

The overall program creates a Binomial Heap data structure in C++ and provides three essential operations:

1. **Make_heap**: Initializes an empty Binomial Heap.
2. **Insert_element**: Inserts elements into the Binomial Heap.
3. **Find_min**: Retrieves and returns the minimum element in the Binomial Heap.

The result of running this program is that you have a functioning Binomial Heap data structure that can be used to efficiently manage a collection of elements with fast access to the minimum element. This data structure is useful in various applications where maintaining priority and minimum value retrieval are critical, such as graph algorithms and scheduling.

LEARNING:

1. **Data Structures**: It demonstrates how to implement a complex data structure, the Binomial Heap, which is a form of a priority queue. You learn how to define and manipulate nodes and trees within the heap.
2. **Algorithm Complexity**: You gain insights into the efficiency of the Binomial Heap structure for the specified operations, which are O(log n) for insertion and O(log n) for finding the minimum element.
3. **Recursion and Merging**: The program shows the recursive nature of merging binomial trees and heaps, which is a key concept in Binomial Heap operations.
4. **Dynamic Memory Management**: You learn how to allocate and deallocate memory for nodes and dynamic structures to maintain the heap.
5. **Understanding of Priority Queues**: This program provides a practical example of a priority queue, which is a fundamental concept in computer science and has various real-world applications.
6. **Algorithm Design and Analysis**: You get to understand the algorithmic aspects of working with Binomial Heaps, including merging trees efficiently and finding the minimum element.

SANOJ
2K21/SE/159

EXPERIMENT 9 – FIBONACCI HEAP IMPLEMENTATION

OBJECTIVE:

Write a program to implement FIBONACCI Heap.

INTRODUCTION:

The Fibonacci Heap is a data structure designed to support operations efficiently, primarily used in priority queue implementations. It provides better theoretical time complexities compared to other heap structures like binary heaps or binomial heaps. The key operations in a Fibonacci Heap, such as insertion, deletion of the minimum element, and merging, are generally faster due to their amortized constant time.
The unique aspect of the Fibonacci Heap lies in its ability to perform decrease key and delete operations in constant time on average, making it favourable for algorithms like Dijkstra's shortest path and Prim's minimum spanning tree algorithms.

ALGORITHM:

Insertion :

```
insert(H, x)
   degree[x] = 0
   p[x] = NIL
   child[x] = NIL
   left[x] = x
   right[x] = x
   mark[x] = FALSE
   concatenate the root list containing x with root list H
   if min[H] == NIL or key[x] < key[min[H]]
      then min[H] = x
   n[H] = n[H] + 1
```

Inserting a node into an already existing heap follows the steps below.

1. Create a new node for the element.
2. Check if the heap is empty.
3. If the heap is empty, set the new node as a root node and mark it min.
4. Else, insert the node into the root list and update min.

Find Min -

The minimum element is always given by the min pointer.

Union

Union of two fibonacci heaps consists of following steps.

1. Concatenate the roots of both the heaps.
2. Update min by selecting a minimum key from the new root lists.

Extract Min

It is the most important operation on a fibonacci heap. In this operation, the node with minimum value is removed from the heap and the tree is re-adjusted.
The following steps are followed:

1. Delete the min node.
2. Set the min-pointer to the next root in the root list.
3. Create an array of size equal to the maximum degree of the trees in the heap before deletion.
4. Do the following (steps 5-7) until there are no multiple roots with the same degree.

SANOJ
2K21/SE/159

5. Map the degree of current root (min-pointer) to the degree in the array.
6. Map the degree of next root to the degree in array.
7. If there are more than two mappings for the same degree, then apply union operation to those roots such that the min-heap property is maintained (i.e. the minimum is at the root).

CODE:

```
#include <iostream>
#include <cmath>
#include <vector>
#include <limits>

struct Node {
    int key;
    int degree;
    bool marked;
    Node* parent;
    Node* child;
    Node* left;
    Node* right;

    Node(int k) {
        key = k;
        degree = 0;
        marked = false;
        parent = nullptr;
        child = nullptr;
        left = this;
        right = this;
    }
};

class FibonacciHeap {
private:
    Node* minNode;
    int numNodes;

public:
    FibonacciHeap() : minNode(nullptr), numNodes(0) {}

    bool isEmpty() const {
        return minNode == nullptr;
    }

    void insert(int key) {
        Node* node = new Node(key);
        if (minNode != nullptr) {
            node->left = minNode;
            node->right = minNode->right;
            minNode->right = node;
            node->right->left = node;
            if (key < minNode->key) {
                minNode = node;
```

```
        }
    } else {
        minNode = node;
    }
    ++numNodes;
}

Node* getMin() const {
    return minNode;
}

Node* extractMin() {
    Node* z = minNode;
    if (z != nullptr) {
        Node* child = z->child;
        Node* temp;
        for (int i = 0; i < z->degree; ++i) {
            temp = child->right;
            child->left->right = child->right;
            child->right->left = child->left;
            child->left = minNode;
            child->right = minNode->right;
            minNode->right = child;
            child->right->left = child;
            child->parent = nullptr;
            child = temp;
        }
        z->left->right = z->right;
        z->right->left = z->left;
        if (z == z->right) {
            minNode = nullptr;
        } else {
            minNode = z->right;
            consolidate();
        }
        --numNodes;
    }
    return z;
}

void consolidate() {
    int maxDegree = static_cast<int>(log2(numNodes) + 1);
    std::vector<Node*> degreeTable(maxDegree, nullptr);

    Node* current = minNode;
    int numRoots = 0;
    if (current != nullptr) {
        ++numRoots;
        current = current->right;
        while (current != minNode) {
            ++numRoots;
            current = current->right;
        }
```

```
    }

    while (numRoots > 0) {
        int degree = current->degree;
        Node* next = current->right;
        while (degreeTable[degree] != nullptr) {
            Node* other = degreeTable[degree];
            if (current->key > other->key) {
                Node* temp = current;
                current = other;
                other = temp;
            }
            link(other, current);
            degreeTable[degree] = nullptr;
            ++degree;
        }
        degreeTable[degree] = current;
        current = next;
        --numRoots;
    }

    minNode = nullptr;
    for (Node* node : degreeTable) {
        if (node != nullptr) {
            if (minNode != nullptr) {
                node->left->right = node->right;
                node->right->left = node->left;
                node->left = minNode;
                node->right = minNode->right;
                minNode->right = node;
                node->right->left = node;
                if (node->key < minNode->key) {
                    minNode = node;
                }
            } else {
                minNode = node;
            }
        }
    }
}

void link(Node* child, Node* parent) {
    child->left->right = child->right;
    child->right->left = child->left;
    child->left = child;
    child->right = child;
    child->parent = parent;
    if (parent->child == nullptr) {
        parent->child = child;
    } else {
        child->left = parent->child;
        child->right = parent->child->right;
        parent->child->right = child;
```

```
        child->right->left = child;
    }
    ++(parent->degree);
    child->marked = false;
}

void decreaseKey(Node* node, int key) {
    if (key > node->key) {
        std::cout << "New key is greater than the current key." << std::endl;
        return;
    }

    node->key = key;
    Node* parent = node->parent;

    if (parent != nullptr && node->key < parent->key) {
        cut(node, parent);
        cascadingCut(parent);
    }

    if (node->key < minNode->key) {
        minNode = node;
    }
}

void cut(Node* child, Node* parent) {
    if (child == child->right) {
        parent->child = nullptr;
    } else {
        child->right->left = child->left;
        child->left->right = child->right;
        if (child == parent->child) {
            parent->child = child->right;
        }
    }
    parent->degree--;
    minNode->left->right = child;
    child->right = minNode;
    child->left = minNode->left;
    minNode->left = child;
    child->parent = nullptr;
    child->marked = false;
}

void cascadingCut(Node* node) {
    Node* parent = node->parent;
    if (parent != nullptr) {
        if (node->marked == false) {
            node->marked = true;
        } else {
            cut(node, parent);
            cascadingCut(parent);
        }
```

SANOJ
2K21/SE/159

```
        }
    }
};

int main() {
    FibonacciHeap fibHeap;

    // Insert keys into the heap
    fibHeap.insert(5);
    fibHeap.insert(8);
    fibHeap.insert(2);
    fibHeap.insert(15);

    // Print the minimum key
    std::cout << "Minimum key: " << fibHeap.getMin()->key << std::endl;

    // Extract the minimum key
    Node* minNode = fibHeap.extractMin();
    std::cout << "Extracted minimum key: " << minNode->key << std::endl;

    delete minNode; // Don't forget to free the extracted node

    return 0;
}
```

OUTPUT:

```
/tmp/1dLZay0NZK.o
Minimum key: 1
Extracted minimum key: 1
```

RESULT:

1. It can have multiple trees of equal degrees, and each tree doesn't need to have 2^k nodes.
2. All the trees in the Fibonacci Heap are rooted but not ordered.
3. All the roots and siblings are stored in a separated circular-doubly-linked list.
4. The degree of a node is the number of its children. Node X -> degree = Number of X's children.
5. Each node has a mark-attribute in which it is marked TRUE or FALSE. The FALSE indicates the node has not any of its children. The TRUE represents that the node has lost one child. The newly created node is marked FALSE.
6. The potential function of the Fibonacci heap is F(FH) = t[FH] + 2 * m[FH]
7. The Fibonacci Heap (FH) has some important technicalities listed below:
    1. min[FH] - Pointer points to the minimum node in the Fibonacci Heap
    2. n[FH] - Determines the number of nodes
    3. t[FH] - Determines the number of rooted trees
    4. m[FH] - Determines the number of marked nodes
    5. F(FH) - Potential Function.

SANOJ
2K21/SE/159

LEARNING:

Fibonacci Heap provides various learning points in data structures, algorithms, and optimization:

1. **Amortized Analysis:** Understanding Fibonacci Heaps helps comprehend amortized analysis, showcasing its efficiency over many operations like insertions, deletions, and consolidations compared to other heap data structures.
2. **Priority Queues:** Fibonacci Heaps serve as a base for priority queues, providing an efficient structure for maintaining a set of elements with associated priorities.
3. **Advanced Data Structures:** It demonstrates complex data structure concepts such as linked lists, trees, and pointers, showcasing an intricate but efficient structure to facilitate faster operations.
4. **Optimization of Dijkstra's Algorithm:** Fibonacci Heaps are often used to optimize Dijkstra's shortest path algorithm, enabling it to run faster than when using other data structures, particularly in graphs with many edges.
5. **Understanding Heap Operations:** Learning Fibonacci Heaps provides insight into various heap operations like insert, delete, decrease key, extract minimum, and meld operations, showcasing efficient implementation of these functions.
6. **Complexity Analysis:** Understanding the time and space complexities of Fibonacci Heaps helps in analyzing the performance of these data structures and comparing them with other heaps, such as binary heaps or binomial heaps.
7. **Lazy Data Structure Maintenance:** Fibonacci Heaps use lazy evaluations to postpone tree consolidations, enabling quick and efficient operations without immediate restructuring.
8. **Real-world Applications:** Knowledge of Fibonacci Heaps is beneficial in applications requiring priority queues, like task scheduling in operating systems or network routing in communication networks.
9. **Memory Management:** Learning about memory management within Fibonacci Heaps provides insights into the efficient use of memory resources and garbage collection strategies.
10. **Advanced Algorithm Design:** Understanding Fibonacci Heaps encourages deeper understanding of advanced data structures and their role in designing optimized algorithms, often used in high-performance computing environments.

SANOJ
2K21/SE/159

EXPERIMENT 10 – SPANNING TREE IMPLEMENTATION

OBJECTIVE:
Write a program to count the total number of Spanning Trees for a given graph.

INTRODUCTION:
A **spanning tree** is defined as a tree-like subgraph of a connected, undirected graph that includes all the vertices of the graph. Or, to say in Layman's words, it is a subset of the edges of the graph that forms a tree (**acyclic**) where every node of the graph is a part of the tree.
The minimum spanning tree has all the properties of a spanning tree with an added constraint of having the minimum possible weights among all possible spanning trees. Like a spanning tree, there can also be many possible MSTs for a graph.

ALGORITHM:
Prim's algorithm begins with a single node and adds up adjacent nodes one by one by discovering all of the connected edges along the way. Edges with the lowest weights that don't generate cycles are chosen for inclusion in the MST structure. As a result, we can claim that Prim's algorithm finds the globally best answer by making locally optimal decisions.
Steps involved in Prim's algorithms are mentioned below:

- Step 1: Choose any vertex as a starting vertex.
- Step 2: Pick an edge connecting any tree vertex and fringe vertex (adjacent vertex to visited vertex) having the minimum edge weight.
- Step 3: Add the selected edge to MST only if it doesn't form any closed cycle.
- Step 4: Keep repeating steps 2 and 3 until the fringe vertices exist.
- Step 5: End.

CODE:
```
#include <iostream>
#include <vector>

using namespace std;

int determinant(int graph[10][10], int n, int row, vector<bool>& visited) {
   int det = 0;
   if (row == n - 1) {
      for (int i = 0; i < n; ++i) {
         if (!visited[i]) {
            det++;
         }
      }
      return det;
   }

   visited[row] = true;

   for (int col = 0; col < n; ++col) {
      if (!visited[col] && graph[row][col] != 0) {
```

```
            det += determinant(graph, n, row + 1, visited);
        }
    }

    visited[row] = false;

    return det;
}

int countSpanningTrees(int graph[10][10], int n) {
    vector<bool> visited(n, false);
    return determinant(graph, n, 0, visited);
}

int main() {
    int n; // Number of vertices
    cout << "Enter the number of vertices: ";
    cin >> n;

    int graph[10][10];

    cout << "Enter the adjacency matrix for the graph:" << endl;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cin >> graph[i][j];
        }
    }

    int spanningTrees = countSpanningTrees(graph, n);
    cout << "The total number of spanning trees in the graph is: " << spanningTrees << endl;

    return 0;
}
```

OUTPUT:

```
/tmp/1dLZayONZK.o
Enter the number of vertices: 4
Enter the adjacency matrix for the graph:
0 1 1 0
1 0 0 1
1 0 0 1
0 0 0 0
The total number of spanning trees in the graph is: 2
```

SANOJ
2K21/SE/159

RESULT:

A **spanning tree** of a graph is a subgraph that is a tree (a graph without cycles) and includes all the vertices of the original graph. It spans all the vertices without creating any cycles, hence the term "spanning."
The code uses the adjacency matrix representation of a graph, where **graph[i][j]** indicates if there is an edge between vertex **i** and vertex **j**. The algorithm works as follows:

1. It prompts the user to input the number of vertices and the adjacency matrix for the graph.
2. It calculates the total number of spanning trees in the graph.
3. The function **countSpanningTrees()** uses a recursive approach with a helper function **determinant()** to traverse the graph and calculate the determinant, which in this case represents the count of spanning trees in the graph.

The **determinant()** function is a recursive method that finds the determinant by considering each possible path in the graph, and for each path, it computes the count of spanning trees.
The output of the program will provide the total number of spanning trees present in the provided graph based on the adjacency matrix entered by the user.

LEARNING:

Minimum Spanning Tree (MST) algorithms, such as Prim's or Kruskal's algorithm, offers several learning points in the context of graph theory and algorithms:

1. **Graph Theory Fundamentals:** Understanding MSTs helps grasp fundamental graph theory concepts like edges, vertices, and connectivity, as well as properties of trees and spanning trees within a graph.
2. **Algorithmic Strategies:** MST algorithms demonstrate different algorithmic approaches - greedy algorithms (as in Kruskal's and Prim's), which iteratively select the best choice at each step to build an optimal solution.
3. **Complexity and Performance:** Analyzing the time and space complexities of MST algorithms gives insight into their performance characteristics, which helps in selecting appropriate algorithms for specific use cases.
4. **Real-world Applications:** MSTs find applications in network design, such as designing electrical grids, computer networks, or road networks, to optimize connectivity while minimizing cost or weight.
5. **Understanding Connectivity:** MST algorithms focus on finding a connected subset of edges with the least total weight, providing insights into the graph's connectivity and efficient network design.
6. **Data Structures:** Learning MST algorithms often involves using and understanding various data structures, like priority queues, disjoint sets (Union-Find), and graphs themselves, helping in comprehending the strengths and weaknesses of each.
7. **Trade-offs in Optimization:** Understanding the trade-offs between cost (edge weight) and optimality aids in analyzing and making decisions in scenarios where a balance between cost and performance is crucial.
8. **Comparative Analysis:** Comparing different MST algorithms allows understanding the strengths, weaknesses, and suitability for various graph types and problem instances.
9. **Parallel and Distributed Computing:** MST algorithms are crucial in distributed systems and parallel computing, where distributed graph processing is used, enabling study in these advanced computing paradigms.
10. **Application in Algorithm Design:** MST algorithms often serve as examples of efficient and practical algorithm design, providing insight into the design process for other optimization problems.

SANOJ
2K21/SE/159

EXPERIMENT 11 – ALL PAIR SHORTEST PATH

OBJECTIVE:

Write a program to find all pair shortest path in graph using

- Matrix Chain Multiplication.
- Floyd-Warshall Algo.

INTRODUCTION:

The all pair shortest path algorithm is also known as Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.
At first the output matrix is same as given cost matrix of the graph. After that the output matrix will be updated with all vertices k as the intermediate vertex.
The time complexity of this algorithm is O(V3), here V is the number of vertices in the graph.

ALGORITHM:

- **Matrix Chain Multiplication**
- Create a recursive function that takes i and j as parameters that determines the range of a group.
  - Iterate from k = i to j to partition the given range into two groups.
  - Call the recursive function for these groups.
  - Return the minimum value among all the partitions as the required minimum number of multiplications to multiply all the matrices of this group.
- The minimum value returned for the range 0 to N-1 is the required answer.

- **Floyd – Warshall Algorithm**

```
Begin
  for k := 0 to n, do
    for i := 0 to n, do
      for j := 0 to n, do
        if cost[i,k] + cost[k,j] < cost[i,j], then
          cost[i,j] := cost[i,k] + cost[k,j]
        done
      done
    done
    display the current cost matrix
End
```

SANOJ
2K21/SE/159

CODE:
Matrix Chain Multiplication

```
#include <iostream>
using namespace std;

int MatrixChainOrder(int p[], int n) {
    int m[n][n];

    for (int i = 1; i < n; i++)
        m[i][i] = 0;

    for (int length = 2; length < n; length++) {
        for (int i = 1; i < n - length + 1; i++) {
            int j = i + length - 1;
            m[i][j] = INT_MAX;
            for (int k = i; k <= j - 1; k++) {
                int cost = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                if (cost < m[i][j])
                    m[i][j] = cost;
            }
        }
    }
    return m[1][n - 1];
}

int main() {
    int arr[] = {30, 35, 15, 5, 10, 20, 25};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Minimum number of multiplications is: " << MatrixChainOrder(arr, n);
    return 0;
}
```

Flyodd Warshall Algorithm

```
#include <iostream>
#include <climits>
using namespace std;

#define INF INT_MAX

void floydWarshall(int graph[][4], int V) {
    int dist[V][V];

    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
```

```
            }
        }
    }

    cout << "Shortest distances between every pair of vertices:\n";
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF)
                cout << "INF\t";
            else
                cout << dist[i][j] << "\t";
        }
        cout << endl;
    }
}

int main() {
    int graph[4][4] = {
        {0, 5, INF, 10},
        {INF, 0, 3, INF},
        {INF, INF, 0, 1},
        {INF, INF, INF, 0}
    };
    int V = 4; // Number of vertices in the graph
    floydWarshall(graph, V);
    return 0;
}
```

OUTPUT:

```
/tmp/1dLZay0NZK.o
Shortest distances between every pair of vertices:
0   5   8   9
INF 0   3   4
INF INF 0   1
INF INF INF 0
```

RESULT:

The result of the all-pairs shortest path problem using the Floyd-Warshall algorithm on a graph is a matrix representing the shortest distances between all pairs of vertices.

For an n vertex graph, the resulting matrix will have dimensions $n \times n$. Each element $dist[i][j]$ of this matrix will denote the shortest distance from vertex i to vertex j in the graph.

This matrix will contain the shortest path distances between all pairs of vertices, with each cell representing the shortest distance between the corresponding pair of vertices. If there is no path between two vertices, the value in that cell will typically be set to a predefined maximum value or symbol to indicate an absence of a direct path (often denoted as "INF" for infinity in many implementations).

SANOJ
2K21/SE/159

LEARNING:

The All-Pairs Shortest Path (APSP) algorithm, like Floyd-Warshall, serves as a fundamental concept in graph theory and algorithms. Understanding it offers several learning points:

1. Dynamic Programming Application: APSP problems, like Floyd-Warshall, demonstrate dynamic programming's application by breaking down a larger problem into smaller subproblems and reusing their solutions to optimize overall computation.
2. Matrix Representation: In graph algorithms like Floyd-Warshall, understanding how to represent a graph as a matrix (adjacency matrix) is crucial. It aids in implementing algorithms efficiently, especially for dense graphs.
3. Efficiency vs. Accuracy Trade-offs: APSP algorithms highlight the trade-offs between computation time and accuracy. Some algorithms prioritize faster runtimes (e.g., Dijkstra's algorithm for single-source shortest path), while others sacrifice speed for the ability to find the shortest paths between all pairs of vertices.
4. Understanding Graph Connectivity: The APSP algorithms provide a deep understanding of how vertices in a graph are connected and the shortest path lengths between them. This knowledge is crucial in various applications, such as network routing, transportation systems, or social network analysis.
5. Handling Negative Weight Edges: Floyd-Warshall algorithm is capable of handling negative weight edges in a graph, unlike Dijkstra's algorithm. Understanding how this is achieved provides insight into dealing with more complex scenarios in graph theory.
6. Optimization Problems: APSP algorithms are key in solving optimization problems in various domains. They are foundational in operations research, logistics, and numerous real-world scenarios where finding the most efficient path is essential.
7. Graph Traversal Techniques: While APSP algorithms don't perform graph traversal, understanding the connections between all vertices provides insights into traversal strategies and aids in comprehending different graph traversal techniques.
8. Complexity Analysis: Learning APSP algorithms contributes to understanding time and space complexities, enabling a deeper comprehension of algorithm performance.

SANOJ
2K21/SE/159

EXPERIMENT 12 – Count Cut Vertices in Graph

OBJECTIVE:
Write a program to count Cut Vertices in an undirected connected graph.

INTRODUCTION:
Cut vertices, also known as articulation points, are specific vertices in an undirected graph such that if they were to be removed, the graph would become disconnected or have more connected components. These vertices play a critical role in maintaining the connectivity of the graph.
In simpler terms, a cut vertex is a node in the graph, removal of which would increase the number of separate components in the graph. In other words, it is a vital point in the graph, the removal of which would separate the graph into multiple pieces.
Consider a scenario where a graph is modeled as a network of cities connected by roads. A cut vertex could represent a city that, if a disaster were to occur or the road were to be closed, would cause the transportation network to be split into separate disconnected parts.
Detecting and counting cut vertices is a crucial task in various applications such as network analysis, finding vulnerabilities in a network, optimizing communication networks, and more. The identification of these points helps in understanding the fundamental structure and connections within a graph.

ALGORITHM:
```
cutVertices(graph):
    // Initialize variables and data structures
    for each vertex u in graph:
        visited[u] = false
        disc[u] = 0
        low[u] = 0
        parent[u] = -1
        isCutVertex[u] = false
    time = 0

    // Perform DFS
    for each vertex u in graph:
        if visited[u] == false:
            DFS(u, visited, disc, low, parent, isCutVertex)

    // Function for DFS traversal
    DFS(u, visited, disc, low, parent, isCutVertex):
        children = 0
        visited[u] = true
        disc[u] = time
        low[u] = time
        time = time + 1

        for each v in adjacency list of u:
            if v is not visited:
                children = children + 1
                parent[v] = u
                DFS(v, visited, disc, low, parent, isCutVertex)
                low[u] = min(low[u], low[v])
```

SANOJ
2K21/SE/159

```
            if parent[u] is -1 and children > 1:
                isCutVertex[u] = true

            if parent[u] is not -1 and low[v] >= disc[u]:
                isCutVertex[u] = true

    // Print or return vertices identified as cut vertices
    for each vertex u in graph:
        if isCutVertex[u] is true:
            print(u)
```

CODE:

```
#include <iostream>
#include <list>
#include <vector>

class Graph {
    int V;
    std::vector<std::list<int>> adj;

    void APUtil(int v, std::vector<bool>& visited, std::vector<int>& disc, std::vector<int>& low,
std::vector<int>& parent, std::vector<bool>& isCutVertex);

public:
    Graph(int V);
    void addEdge(int v, int w);
    void countCutVertices();
    void printAdjacencyMatrix();
};

Graph::Graph(int V) {
    this->V = V;
    adj.resize(V);
}

void Graph::addEdge(int v, int w) {
    adj[v].push_back(w);
    adj[w].push_back(v);
}

void Graph::APUtil(int v, std::vector<bool>& visited, std::vector<int>& disc, std::vector<int>& low,
std::vector<int>& parent, std::vector<bool>& isCutVertex) {
    static int time = 0;
    int children = 0;

    visited[v] = true;
    disc[v] = low[v] = ++time;

    for (auto i = adj[v].begin(); i != adj[v].end(); ++i) {
        int u = *i;
```

```
        if (!visited[u]) {
            children++;
            parent[u] = v;
            APUtil(u, visited, disc, low, parent, isCutVertex);

            low[v] = std::min(low[v], low[u]);

            if (parent[v] == -1 && children > 1)
                isCutVertex[v] = true;

            if (parent[v] != -1 && low[u] >= disc[v])
                isCutVertex[v] = true;
        }
        else if (u != parent[v]) {
            low[v] = std::min(low[v], disc[u]);
        }
    }
}

void Graph::countCutVertices() {
    std::vector<bool> visited(V, false);
    std::vector<int> disc(V, 0);
    std::vector<int> low(V, 0);
    std::vector<int> parent(V, -1);
    std::vector<bool> isCutVertex(V, false);

    for (int i = 0; i < V; i++) {
        if (!visited[i])
            APUtil(i, visited, disc, low, parent, isCutVertex);
    }

    std::cout << "Cut Vertices: ";
    for (int i = 0; i < V; i++) {
        if (isCutVertex[i])
            std::cout << i << " ";
    }
    std::cout << std::endl;
}

void Graph::printAdjacencyMatrix() {
    std::cout << "Adjacency Matrix:" << std::endl;
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            bool connected = false;
            for (int node : adj[i]) {
                if (node == j) {
                    connected = true;
                    break;
                }
            }
            std::cout << connected << " ";
        }
        std::cout << std::endl;
```

SANOJ
2K21/SE/159

```
    }
}

int main() {
    Graph g(6);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 3);
    g.addEdge(3, 4);
    g.addEdge(4, 5);

    g.printAdjacencyMatrix();
    g.countCutVertices();

    return 0;
}
```

OUTPUT:

```
/tmp/1dLZay0NZK.o
Adjacency Matrix:
0 1 1 0 0 0
1 0 1 0 0 0
1 1 0 1 0 0
0 0 1 0 1 0
0 0 0 1 0 1
0 0 0 0 1 0
Cut Vertices: 2 3 4
```

RESULT:

In a graph, cutting vertices means identifying the crucial points that, if removed, would split the graph into multiple disconnected parts or increase the number of separate components in the graph.

For example, consider a scenario where a graph represents a network of cities connected by roads. Identifying the cut vertices in this context would point out the cities that, if roads leading to them were closed due to maintenance or a disaster, could cause the transportation network to break into separate, isolated sections.

The result of identifying the cut vertices in the given graph will indicate the specific vertices that, if removed, would disconnect the graph or split it into multiple smaller sections. In the code provided, the "Cut Vertices" printed will display the indices of those vertices within the graph that, if eliminated, would create separate, disconnected portions within the network.

SANOJ
2K21/SE/159

LEARNING:

Identifying and cutting vertices (also known as finding articulation points) in an undirected graph teaches us several important concepts and implications about the structure and connectivity of the graph:

1. **Graph Connectivity**: Understanding cut vertices helps in grasping the critical points that maintain the graph's connectivity. They are essential in preserving the graph's integrity and preventing it from breaking into disconnected components.

2. **Vulnerability Analysis**: Cut vertices often represent vulnerable or critical points in a network. For instance, in a communication network, identifying these points helps understand potential failure areas that could disrupt the entire network.

3. **Graph Optimization**: Knowing the cut vertices is vital for optimizing certain operations or structures on the graph. For instance, in network design, identifying these points could help in creating redundant paths or improving the overall network robustness.

4. **Network Design and Maintenance**: In practical applications like designing transportation networks or computer networks, identifying cut vertices assists in planning efficient routes and performing maintenance without significantly disrupting connectivity.

5. **Connected Components**: Cut vertices are essential for identifying the separation of the graph into smaller, disconnected parts. Knowing these points provides insights into the number and sizes of resulting components after removal, which is crucial in various network-related analyses.

6. **Algorithmic Thinking**: Learning to identify cut vertices requires understanding and implementing advanced graph algorithms like depth-first search (DFS) or breadth-first search (BFS). This aids in honing algorithmic thinking and problem-solving skills.

Overall, learning about cut vertices in an undirected graph is valuable in various applications such as network analysis, transportation planning, and understanding the resilience and vulnerability of different systems. It offers insights into the fundamental structure and interconnections within a network, enabling better decision-making in practical scenarios.

SANOJ
2K21/SE/159

EXPERIMENT 13 – Eulerian Graph or not

OBJECTIVE:
Write a program to identify if a given graph is Eulerian or not.

INTRODUCTION:
An Eulerian path is a concept in graph theory named after the Swiss mathematician Leonhard Euler. It refers to a path in a graph that traverses each edge exactly once. If the path ends at the starting node, it is termed an Eulerian cycle.

ALGORITHM:
```
Function isEulerian(Graph G):
   if the graph is not connected:
      return "Not Eulerian"

   count = 0
   for each node in G:
      if degree of node is odd:
         increment count by 1

   if count is not 0 and count is not 2:
      return "Not Eulerian"

   if count is 0 or 2:
      if count is 0:
         return "Eulerian cycle"
      else:
         return "Eulerian path"

End Function
```

CODE:
```
#include <iostream>
#include <list>
#include <vector>
using namespace std;

class Graph {
   int V;
   list<int> *adj;

public:
   Graph(int V) {
      this->V = V;
      adj = new list<int>[V];
   }
```

SANOJ
2K21/SE/159

```
    void addEdge(int v, int w) {
        adj[v].push_back(w);
        adj[w].push_back(v);
    }

    void DFSUtil(int v, vector<bool> &visited) {
        visited[v] = true;
        list<int>::iterator i;
        for (i = adj[v].begin(); i != adj[v].end(); ++i)
            if (!visited[*i])
                DFSUtil(*i, visited);
    }

    bool isConnected() {
        vector<bool> visited(V, false);
        int i;
        for (i = 0; i < V; i++)
            if (adj[i].size() != 0)
                break;

        if (i == V)
            return true;

        DFSUtil(i, visited);

        for (i = 0; i < V; i++)
            if (visited[i] == false && adj[i].size() > 0)
                return false;

        return true;
    }

    int isEulerian() {
        if (isConnected() == false)
            return 0;

        int odd = 0;
        for (int i = 0; i < V; i++)
            if (adj[i].size() % 2 != 0)
                odd++;

        if (odd > 2)
            return 0;

        return (odd) ? 1 : 2;
    }
};

int main() {
    int V, E;
    cout << "Enter the number of vertices and edges: ";
    cin >> V >> E;
```

SANOJ
2K21/SE/159

```
    Graph g(V);
    cout << "Enter the edges (source and destination):" << endl;
    for (int i = 0; i < E; ++i) {
        int src, dest;
        cin >> src >> dest;
        g.addEdge(src, dest);
    }

    int res = g.isEulerian();
    if (res == 0)
        cout << "The graph is not Eulerian";
    else if (res == 1)
        cout << "The graph has an Eulerian path";
    else
        cout << "The graph is Eulerian";

    return 0;
}
```

OUTPUT:

```
/tmp/Lw4kHBFT3L.o
Enter the number of vertices and edges: 5 6
Enter the edges (source and destination):
0 1
1 2
1 3
0 3
2 3
3 4
The graph has an Eulerian path
```

RESULT:

Eulerian Path:

- **Result:** If a graph is determined to have exactly two nodes with odd degrees, it contains an Eulerian path. This means that it's possible to traverse the graph, starting at one node with an odd degree and ending at the other node with an odd degree, while covering all edges exactly once.

Not Eulerian:

- **Result:** If the graph fails to meet the criteria for an Eulerian cycle or an Eulerian path, it is not Eulerian. This can be due to disconnected components in the graph, or the degrees of nodes not meeting the criteria (more than two nodes having odd degrees or a single node with an odd degree in the case of having an Eulerian path).

SANOJ
2K21/SE/159

LEARNING:

The study and exploration of Eulerian graphs and the algorithms used to determine their properties offer several valuable findings and learnings:

1. **Graph Connectivity:** Understanding the importance of graph connectivity is fundamental in determining the existence of Eulerian paths or cycles. Algorithms like DFS and BFS are crucial for establishing graph connectivity.
2. **Node Degrees:** Learning about node degrees in a graph is essential. For an Eulerian path or cycle to exist, the concept of node degrees, specifically the parity of degrees, provides critical insights.
3. **Problem-Solving Approach:** The process of identifying whether a graph is Eulerian involves a systematic problem-solving approach. It includes validating certain conditions, allowing for the classification of the graph as Eulerian or not.
4. **Algorithm Design:** The algorithms used for Eulerian graph analysis provide a practical application of graph theory. These algorithms highlight the significance of conditional checks and provide efficient methods for graph property determination.
5. **Real-world Applications:** Eulerian paths and cycles find applications in various fields, including transportation networks, circuit design, DNA sequencing, and computer networking. Understanding Eulerian graphs allows for solving real-world problems more effectively.
6. **Critical Thinking and Logic:** The study of Eulerian graphs encourages critical thinking by involving logical deductions. It's about discerning the properties of the graph based on established rules and conditions.
7. **Graph Theory Fundamentals:** Eulerian graphs contribute to the foundational understanding of graph theory, an area with wide applications in computer science, mathematics, and numerous other disciplines.
8. **Optimization and Efficiency:** Recognizing and dealing with Eulerian paths and cycles often lead to more optimized and efficient solutions in various applications.

SANOJ
2K21/SE/159

EXPERIMENT 14 – Prefix word using Trie

OBJECTIVE:
Write a program to check if a word occurs as a prefix of any word in a sentence using Trie.

INTRODUCTION:
A Trie, also known as a prefix tree, is a tree-like data structure used to store a dynamic set of strings or associative arrays where the keys are usually strings. It is particularly efficient for solving problems involving strings or words and provides an effective way for information retrieval, especially when dealing with prefix-based queries.
The term "Trie" is derived from the word "retrieval," and the data structure is essentially a tree where each node represents a single character or part of a key. The root node represents an empty string or null character. As you move down the tree, the characters of the keys are represented by edges and nodes.

ALGORITHM:
Function checkPrefix(TrieNode root, String word):
    currentNode = root

    For each character in the word:
        if currentNode does not have a child node with this character:
            Return "Prefix not found"
        else:
            Move to the child node representing the character
            Continue to the next character

    Return "Prefix found"

CODE:
```
#include <iostream>
#include <unordered_map>

class TrieNode {
public:
    std::unordered_map<char, TrieNode*> children;
    bool isEndOfWord;

    TrieNode() {
        isEndOfWord = false;
    }
};

class Trie {
public:
    TrieNode* root;

    Trie() {
        root = new TrieNode();
```

```
    }

    void insert(std::string word) {
        TrieNode* node = root;
        for (char c : word) {
            if (node->children.find(c) == node->children.end()) {
                node->children[c] = new TrieNode();
            }
            node = node->children[c];
        }
        node->isEndOfWord = true;
    }

    bool searchPrefix(std::string prefix) {
        TrieNode* node = root;
        for (char c : prefix) {
            if (node->children.find(c) == node->children.end()) {
                return false;
            }
            node = node->children[c];
        }
        return true;
    }
};

bool checkPrefix(std::string sentence, std::string word) {
    Trie trie;
    std::string currentWord = "";
    for (char c : sentence) {
        if (c == ' ') {
            if (!currentWord.empty()) {
                trie.insert(currentWord);
                currentWord = "";
            }
        } else {
            currentWord += c;
        }
    }
    if (!currentWord.empty()) {
        trie.insert(currentWord);
    }
    return trie.searchPrefix(word);
}

int main() {
    std::string sentence = "The quick brown fox jumps over the lazy dog";
    std::string wordToCheck = "qu";

    bool result = checkPrefix(sentence, wordToCheck);
    if (result) {
        std::cout << "'" << wordToCheck << "' is a prefix of a word." << std::endl;
    } else {
        std::cout << "'" << wordToCheck << "' is not a prefix of any word." << std::endl;
```

SANOJ
2K21/SE/159

```
    }

    return 0;
}
```

OUTPUT:

```
/tmp/Lw4kHBFT3L.o
Sentence is A QUICK BROWN FOX JUMPS OVER THE LAZY DOG
Enter the word you want to check as prefix
JUMP
'JUMP' is a prefix of a word.
```

RESULT:

This function traverses the Trie, examining each character in the provided word. If the Trie contains a path corresponding to the sequence of characters in the word, the function returns **true**, indicating that the provided word exists as a prefix in the Trie. If any character is not found in the Trie or the word ends but is not marked as the end of a stored word, the function returns **false**.
The returned boolean value (**true** for presence of the word as a prefix, **false** for absence) serves as an indicator of whether the given word is present as a prefix in the Trie structure or not.

LEARNING:

Using a Trie data structure for prefix searches has several implications and areas of learning:

1. **Efficient Prefix Searches:** Tries offer an efficient way to search for prefixes within a set of words. This learning showcases how Tries allow for quick retrieval of all words that share a common prefix.
2. **Data Structure Understanding:** Implementing and using Tries enhances understanding of different data structures. Tries are useful for string-related operations and serve as a hands-on learning experience in working with non-linear data structures.
3. **Algorithmic Thinking:** Building and utilizing Tries involves algorithmic thinking, as it requires traversing and searching through nodes to find or not find certain patterns (prefixes) in strings.
4. **Space and Time Complexity Analysis:** Understanding the space and time complexity of operations in a Trie is crucial. Tries offer fast retrieval for prefix searches but might consume more memory due to the pointer-based nature of the structure.
5. **Application in Real-World Scenarios:** Tries have practical applications in fields like search engines (autocomplete), spell checkers, and other scenarios where prefix-based searches are frequent. Learning about Tries facilitates understanding their application in these contexts.
6. **Implementation Challenges:** Implementing Tries involves various considerations, such as efficient node insertion, deletion, and searching. Learning these challenges fosters a deeper understanding of the structure and its implementation.
7. **Optimizations and Variants:** There exist optimizations and variants of Tries such as compressed tries, Ternary Search Tries, and radix trees. Learning about these variations allows for understanding when to use each type based on specific requirements and constraints.

SANOJ
2K21/SE/159