B190839CS

ROSE S JOSE

DBMS LAB – EVALUATION 7

SOURCE CODE (CPP)

```
//B190839CS Rose S Jose
#include<iostream>
using namespace std;
struct Node
{
	int *key;
	Node **child;
	bool leaf;
	int n;
	Node(int deg)
	{
		key = new int[deg];
		leaf = true;
		child = new Node*[deg+1];
		for(int i=0;i<deg;i++)
			child[i] = NULL;
	}
};
class bptree
{
private:
	Node *root;
	int m;
	void insertInternal(int data, Node* cur, Node* C);
```

```
        Node* findParent(Node* cur, Node* C);

public:
        bptree(int degree)
        {
                root = NULL;
                m = degree;
        }
        void search(int data);
        void insert(int data);
        Node* getRoot();
        void display(Node *cur);
};

void bptree::insert(int data)
{

        if(root == NULL) //If root is null, first element
        {
                root = new Node(m);
                root->key[0] = data;
                root->leaf = true;
                root->n = 1;
        }

        else
        {
                Node* cur = root;
                Node* p;
```

```
while (cur->leaf == false) //finding the position of the child where key to be inserted if cur is not leaf
{
    p = cur;
    for (int i = 0; i < cur->n; i++)
    {

        if (data < cur->key[i])
        {
            cur = cur->child[i];
            break;
        }
        if (i == cur->n - 1)
        {
            cur = cur->child[i + 1];
            break;
        }
    }
}

if (cur->n < m) //if the current node has less than max elements, insert in that node
{
    int i = 0;
    while (data > cur->key[i] && i < cur->n)
        i++;

    for (int j = cur->n; j > i; j--)
    {
        cur->key[j] = cur->key[j - 1];
    }
```

```
                cur->key[i] = data;
                cur->n++;
                cur->child[cur->n] = cur->child[cur->n - 1];
                cur->child[cur->n - 1] = NULL;
        }

        else
        {
                // Create a new_leaf node if the current node has maximum elements,
splitting
                Node* new_leaf = new Node(m);

                int temp_node[m + 1];

                for (int i = 0; i < m; i++) //filling the temp node
                {
                        temp_node[i] = cur->key[i];
                }
                int i = 0, j;

                while (data > temp_node[i] && i < m) //finding the position of new node
                {
                        i++;
                }

                for (int j = m + 1; j > i; j--) //making space for insertion
                        temp_node[j] = temp_node[j - 1];

                temp_node[i] = data;
                new_leaf->leaf = true;
```

```
cur->n = (m + 1) / 2;
new_leaf->n = m + 1 - (m + 1) / 2;

cur->child[cur->n] = new_leaf;
new_leaf->child[new_leaf->n] = cur->child[m];
cur->child[m] = NULL;

for (i = 0; i < cur->n; i++) //updating the current node's keys
{
        cur->key[i] = temp_node[i];
}

// Update the new_leaf key
for (i = 0, j = cur->n; i < new_leaf->n; i++, j++)
{
        new_leaf->key[i] = temp_node[j];
}

// If cur is the root node
if (cur == root) {

        Node* newRoot = new Node(m);

        newRoot->key[0] = new_leaf->key[0];
        newRoot->child[0] = cur;
        newRoot->child[1] = new_leaf;
        newRoot->leaf = false;
        newRoot->n = 1;
        root = newRoot;
}
else {
```

```
                                        // Recursive Call for insert in internal
                                        insertInternal(new_leaf->key[0], p, new_leaf);
                                }
                        }
                }
}
void bptree::insertInternal(int data, Node* cur, Node* C)
{

        if (cur->n < m)
        {
                int i = 0;

                while (data > cur->key[i] && i < cur->n) //finding the position
                {
                        i++;
                }

                for (int j = cur->n; j > i; j--) // creating spaces for insertion
                {
                        cur->key[j] = cur->key[j - 1];
                }

                for (int j = cur->n + 1; j > i + 1; j--) //moving the child pointers
                {
                        cur->child[j] = cur->child[j - 1];
                }

                cur->key[i] = data;
                cur->n++;
```

```
            cur->child[i + 1] = C;
}

else
{
            Node* new_internal = new Node(m);
            int temp_key[m + 1];
            Node* temp_child[m + 2];

            for (int i = 0; i < m; i++)
            {
                        temp_key[i] = cur->key[i];
            }

            for (int i = 0; i < m + 1; i++) {
                        temp_child[i] = cur->child[i];
            }

            int i = 0, j;
            while (data > temp_key[i] && i < m) //finding position of insertion
            {
                        i++;
            }

            for (int j = m + 1; j > i; j--)
            {

                        temp_key[j] = temp_key[j - 1];
            }

            temp_key[i] = data;
```

```
for (int j = m + 2; j > i + 1; j--)
{
        temp_child[j] = temp_child[j - 1];
}

temp_child[i + 1] = C;
new_internal->leaf = false;

cur->n = (m + 1) / 2;
new_internal->n = m - (m + 1) / 2;

// Insert new node as an internal node
for (i = 0, j = cur->n + 1; i < new_internal->n; i++, j++)
{
        new_internal->key[i] = temp_key[j];
}

for (i = 0, j = cur->n + 1; i < new_internal->n + 1; i++, j++)
{
        new_internal->child[i] = temp_child[j];
}

if (cur == root)
{
        Node* newRoot = new Node(m);
        newRoot->key[0] = cur->key[cur->n];

        newRoot->child[0] = cur;
        newRoot->child[1] = new_internal;
        newRoot->leaf = false;
        newRoot->n = 1;
```

```
                root = newRoot;
            }

            else
            {
                insertInternal(cur->key[cur->n], findParent(root, cur), new_internal);
            }
        }
}

// Function to find the parent node
Node* bptree::findParent(Node* cur, Node* C)
{
        Node* parent;
        if (cur->leaf || (cur->child[0])->leaf)
        {
            return NULL;
        }

        // Traverse the current node with all its Children
        for (int i = 0; i < cur->n + 1; i++)
        {

            // Update the parent for the
            // C Node
            if (cur->child[i] == C)
            {
                parent = cur;
                return parent;
            }
```

```
                // Else recursively traverse to
                // find C node
                else
                {
                        parent = findParent(cur->child[i], C);

                        // If parent is found, then
                        // return that parent node
                        if (parent != NULL)
                                return parent;
                }
        }

        // Return parent node
        return parent;
}
Node* bptree::getRoot()
{
        return root;
}
void bptree::display(Node *cur) {
  if (cur != NULL)
  {
        if(cur->leaf)
        {
        for(int i = 0; i < cur->n; i++)
        {
                std::cout << cur->key[i] << " ";
        }
        //std::cout << "\n";
    }
```

```
    if (cur->leaf != true)
    {
      for (int i = 0; i < cur->n + 1; i++)
      {
        display(cur->child[i]);
      }
    }
  }
}
void bptree::search(int data)
{
    if (root == NULL)
    {
        std::cout << "FALSE\n";
    }

    else
    {
        Node* cur = root;
        int level = 1;
        while (cur->leaf == false)
        {
            for (int i = 0; i < cur->n; i++)
            {
                if (data < cur->key[i])
                {
                    cur = cur->child[i];
                    level+=1;
                    break;
                }
                if (i == cur->n - 1) {
```

```
            cur = cur->child[i + 1];
            level += 1;
            break;
          }
        }
      }
      for (int i = 0; i < cur->n; i++)
      {
        if (cur->key[i] == data)
        {
          std::cout << "Level "<<level<<"\n";
          return;
        }
      }
      cout << "FALSE\n";
    }
}
int main()
{
        int m, k;
        char c;
        std::cout<<"Order: ";
        std::cin>>m;
        bptree node(m-1);
        do
        {
                std::cin>>c;
                switch(c)
                {
                        case 'i':std::cin>>k;
                                        node.insert(k);
```

```
                                        break;
                        case 's':std::cin>>k;
                                        node.search(k);
                                        break;
                        case 'p':node.display(node.getRoot());
                                        break;
                        case 'e':break;
                }
                if(c=='e')
                        break;
        }while(1);
        return 0;
}
```