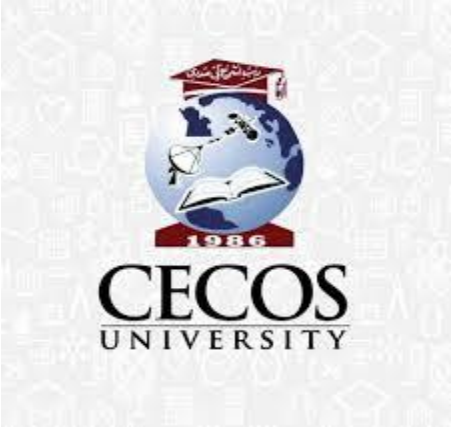


PROJECT :

B-Tree Implementation.

Project Member :

Kashif khan (4657)

Muhamad Huzaifa (4658)

Abdul wahab (4784)

Overview

A **B-Tree** is a self-balancing search tree used in databases and file systems for efficient storage and retrieval. Unlike a binary search tree, B-Trees minimize disk reads and writes by maintaining balanced multi-way branches.

How It Works

1. **Nodes and Order**:
    - Each node contains multiple keys and child pointers.
    - The order **ttt** defines the minimum and maximum number of keys in a node:
        - A node can have at most **2t−12t - 12t−1** keys.
        - A node must have at least **t−1t - 1t−1** keys.
        - Internal nodes (except root) must have at least **ttt** children.
        - The root can have fewer keys but must follow structural properties.
2. **Operations**
    - **Insertion**:

        - Insert a key into a leaf.
        - If the leaf overflows, split it and promote a middle key to the parent.
    - **Deletion**:
        - If the key is in a leaf, remove it.
        - If the key is in an internal node:
            - Replace it with the predecessor or successor.
            - Merge nodes if required to maintain balance.
    - **Search**:
        - Traverse down from the root, choosing the appropriate subtree.

Project Documentation

1. Requirements

- C++ (with STL for basic operations)
- Understanding of tree structures and recursion

2. Dependencies

- No external libraries are needed; only basic C++ STL is used.

3. Implementation Steps

Step 1: Define the B-Tree Node Structure

- Store keys and children in an array.
- Keep a boolean to check if a node is a leaf.

Step 2: Implement Insert Operation

- Traverse down the tree.
- If a node gets full, split it.

Step 3: Implement Delete Operation

- Handle cases based on key location (leaf or internal).
- Merge nodes when necessary.

Step 4: Implement Search and Display Operations

- Provide a function to find keys efficiently.
- Display the tree structure.

Programiz
C++ Online Compiler

main.cpp

Share Run

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

class BTreeNode {
public:
    int *keys;
    int t;
    BTreeNode **C;
    int n;
    bool leaf;

    BTreeNode(int _t, bool _leaf);
    void traverse();
    BTreeNode *search(int k);
    void insertNonFull(int k);
    void splitChild(int i, BTreeNode *y);
    int findKey(int k);
    void remove(int k);
    void removeFromLeaf(int idx);
    void removeFromNonLeaf(int idx);
```

main.cpp

Share

Run

```
21      int getPred(int idx);
22      int getSucc(int idx);
23      void fill(int idx);
24      void borrowFromPrev(int idx);
25      void borrowFromNext(int idx);
26      void merge(int idx);
27
28      friend class BTree;
29  };
30
31  class BTree {
32  public:
33      BTreeNode *root;
34      int t;
35
36      BTree(int _t) { root = NULL; t = _t; }
37
38      void traverse() { if (root != NULL) root->traverse(); }
39      BTreeNode *search(int k) { return (root == NULL) ? NULL :
            root->search(k); }
```

main.cpp

Share Run

```
40      void insert(int k);
41      void remove(int k);
42  };
43
44  // Constructor
45  BTreeNode::BTreeNode(int _t, bool _leaf) {
46      t = _t;
47      leaf = _leaf;
48      keys = new int[2 * t - 1];
49      C = new BTreeNode *[2 * t];
50      n = 0;
51  }
52
53  // Traverse the tree
54  void BTreeNode::traverse() {
55      int i;
56      for (i = 0; i < n; i++) {
57          if (!leaf) C[i]->traverse();
58          cout << " " << keys[i];
59      }
```

main.cpp

Share

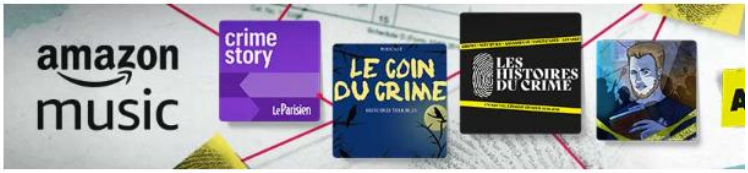
Run

```
60      if (!leaf) C[i]->traverse();
61  }
62
63  // Search key
64  BTreeNode *BTreeNode::search(int k) {
65      int i = 0;
66      while (i < n && k > keys[i]) i++;
67      if (keys[i] == k) return this;
68      if (leaf) return NULL;
69      return C[i]->search(k);
70  }
71
72  // Insert a key
73  void BTree::insert(int k) {
74      if (root == NULL) {
75          root = new BTreeNode(t, true);
76          root->keys[0] = k;
77          root->n = 1;
78      } else {
79          if (root->n == 2 * t - 1) {
```

main.cpp

Share Run

```
79          if (root->n == 2 * t - 1) {
80              BTreeNode *s = new BTreeNode(t, false);
81              s->C[0] = root;
82              s->splitChild(0, root);
83              int i = (s->keys[0] < k) ? 1 : 0;
84              s->C[i]->insertNonFull(k);
85              root = s;
86          } else root->insertNonFull(k);
87      }
88  }
89
90  // Insert in non-full node
91  void BTreeNode::insertNonFull(int k) {
92      int i = n - 1;
93      if (leaf) {
94          while (i >= 0 && keys[i] > k) {
95              keys[i + 1] = keys[i];
96              i--;
97          }
98          keys[i + 1] = k;
```



main.cpp

Share Run

```
99          n++;
100     } else {
101         while (i >= 0 && keys[i] > k) i--;
102         if (C[i + 1]->n == 2 * t - 1) {
103             splitChild(i + 1, C[i + 1]);
104             if (keys[i + 1] < k) i++;
105         }
106         C[i + 1]->insertNonFull(k);
107     }
108 }
109
110 // Split child node
111 void BTreeNode::splitChild(int i, BTreeNode *y) {
112     BTreeNode *z = new BTreeNode(y->t, y->leaf);
113     z->n = t - 1;
114     for (int j = 0; j < t - 1; j++) z->keys[j] = y->keys[j +
            t];
115     if (!y->leaf) for (int j = 0; j < t; j++) z->C[j] = y->C[j
            + t];
116     y->n = t - 1;
```

main.cpp

Share

Run

```
116     y->n = t - 1;
117     for (int j = n; j >= i + 1; j--) C[j + 1] = C[j];
118     C[i + 1] = z;
119     for (int j = n - 1; j >= i; j--) keys[j + 1] = keys[j];
120     keys[i] = y->keys[t - 1];
121     n++;
122 }
123
124 // Remove key from tree
125 void BTree::remove(int k) {
126     if (!root) return;
127     root->remove(k);
128     if (root->n == 0) {
129         BTreeNode *tmp = root;
130         if (root->leaf) root = NULL;
131         else root = root->C[0];
132         delete tmp;
133     }
134 }
135
```

main.cpp

Share

Run

```
135
136 // Remove key from node
137 void BTreeNode::remove(int k) {
138     int idx = findKey(k);
139     if (idx < n && keys[idx] == k) {
140         if (leaf) removeFromLeaf(idx);
141         else removeFromNonLeaf(idx);
142     } else {
143         if (leaf) return;
144         bool flag = (idx == n);
145         if (C[idx]->n < t) fill(idx);
146         if (flag && idx > n) C[idx - 1]->remove(k);
147         else C[idx]->remove(k);
148     }
149 }
150
151 // Remove from leaf node
152 void BTreeNode::removeFromLeaf(int idx) {
153     for (int i = idx + 1; i < n; ++i) keys[i - 1] = keys[i];
154     n--;
```

main.cpp

Share Run

```cpp
}

int BTreeNode::findKey(int k) {
    int idx = 0;
    while (idx < n && keys[idx] < k) idx++;
    return idx;
}

// Main function
int main() {
    BTree t(3);
    t.insert(10);
    t.insert(20);
    t.insert(5);
    t.insert(6);
    t.insert(12);
    t.insert(30);
    t.traverse();
    cout << "\nDeleting 6\n";
    t.remove(6);
```

main.cpp

Share

Run

```
159     while (idx < n && keys[idx] < k) idx++;
160     return idx;
161 }
162
163 // Main function
164 int main() {
165     BTree t(3);
166     t.insert(10);
167     t.insert(20);
168     t.insert(5);
169     t.insert(6);
170     t.insert(12);
171     t.insert(30);
172     t.traverse();
173     cout << "\nDeleting 6\n";
174     t.remove(6);
175     t.traverse();
176     return 0;
177 }
178
```