Eric Blasko

1. **B-Tree Implementation**

// C++ program for B-Tree insertion

// For simplicity, assume order m = 2 \* t

#include<iostream>

#include<vector>

#include<sstream>

using namespace std;

//forward declaration

template <class keyType>

class BTree;

// A BTree node

template <class keyType>

class Node

{

private:

keyType \*keys; // An array of keys

int t; // m = 2 \* t

Node<keyType> \*\*C; // An array of child pointers

int nKeys; // Current number of keys

bool isLeaf; // Is true when node is leaf. Otherwise false

public:

Node(int \_t, bool \_isLeaf); // Constructor

// Inserting a new key in the subtree rooted with

// this node. The node must be non-full when this

// function is called

void insertNonFull(keyType k);

// Spliting 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, Node<keyType> \*y);

// Traversing all nodes in a subtree rooted with this node

void traverse();

// A function to search a key in subtree rooted with this node.

Node \*search(keyType k); // returns NULL if k is not present.

// returns the index of the first key that is greater or equal to k

int findKey(keyType k);

// borrow a key from the c[index-1]-th node and place it into the

// c[idnex]

void promoteFromPrev(int index);

// borrow a key from the c[index+1]-th node and place it into the

// c[index]

void promoteFromNext(int index);

// remove the key present in the index position in the node which is a

// leaf

void removeFromLeaf(int index);

// remove the key present in the index position in the node with is non

// leaf

void removeFromNonLeaf(int index);

// get the predecesor of the key where the key is present in the index

// position of the node

keyType getPred(int index);

// get the successor of the key where the key is present in the indext

// pos

// in the node

keyType getSucc(int index);

// Merges the index-th child of the node with index+1-th child of the

// node

void merge(int index);

// fills up the child node present in the index pos in the c[] array if

// that child has less than t-1 keys

void fill(int index);

// remove the key k in the subtree rooted with this node

void remove(keyType k);

// Make BTree friend of this so that we can access private members of this

// class in BTree functions

friend class BTree<keyType>;

};

// A BTree

template <class keyType>

class BTree

{

private:

Node<keyType> \*root; // Pointer to root node

int t; // Minimum degree

public:

// Constructor (Initializes tree as empty)

BTree(int t0 )

{ root = NULL; t = t0; }

// function to traverse the tree

void traverse()

{ if (root != NULL) root->traverse(); }

// function to search a key in this tree

//Node<int>\* search(keyType k)

Node<keyType>\* search(keyType k)

{ return (root == NULL)? NULL : root->search(k); }

// The main function that inserts a new key in this B-Tree

//void insert(keyType k);

void insert(keyType k);

void remove(keyType k);

};

// Constructor for Node class

template<class keyType>

Node<keyType>::Node(int t0, bool isLeaf0)

{

// Copy the given minimum degree and leaf property

t = t0;

isLeaf = isLeaf0;

// Allocate memory for maximum number of possible keys

// and child pointers

keys = new keyType[2\*t-1];

C = new Node<keyType> \*[2\*t];

// Initialize the number of keys as 0

nKeys = 0;

}

// Traverse all nodes in a subtree rooted at this node

template<class keyType>

void Node<keyType>::traverse()

{

// Depth-first traversal

// There are nKeys keys and nKeys+1 children, traverse through nKeys keys

// and first nKeys children

for (int i = 0; i < nKeys; i++)

{

// If this is not leaf, then before printing key[i],

// traverse the subtree rooted at child C[i].

if (isLeaf == false)

C[i]->traverse();

cout << " " << keys[i];

}

// Print the subtree rooted with last child

if (isLeaf == false)

C[nKeys]->traverse();

}

// Search key k in subtree rooted with this node

template<class keyType>

Node<keyType> \*Node<keyType>::search(keyType k)

{

// Find the first key >= k

int i = 0;

while (i < nKeys && k > keys[i])

i++;

// If the found key is equal to k, return this node

if ( i < nKeys ) // added by Tong

if (keys[i] == k)

return this;

// If key is not found here and this is a Leaf node

if (isLeaf == 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

template <class keyType>

void BTree<keyType>::insert( keyType k)

{

// If tree is empty

if (root == NULL)

{

// Allocate memory for root

root = new Node<keyType>(t, true);

root->keys[0] = k; // Insert key

root->nKeys = 1; // Update number of keys in root

}

else // If tree is not empty

{

//if we find node with key k do not insert into tree

Node<keyType> \*temp = search(k);

if(temp != NULL)

return;

// If root is full, then tree grows in height

if (root->nKeys == 2\*t-1)

{

// Allocate memory for new root

Node<keyType> \*s = new Node<keyType>(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

template <class keyType>

void Node<keyType>::insertNonFull(keyType k)

{

// Initialize index as index of rightmost element

int i = nKeys-1;

// If this is a Leaf node

if (isLeaf == 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;

nKeys++;

}

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]->nKeys == 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);

}

}

// Spliting the child y of this node

// Note that y must be full when this function is called

template<class keyType>

void Node<keyType>::splitChild(int i, Node \*y)

{

// Create a new node which is going to store (t-1) keys

// of y

Node \*z = new Node(y->t, y->isLeaf);

z->nKeys = 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->isLeaf == false)

{

for (int j = 0; j < t; j++)

z->C[j] = y->C[j+t];

}

// Reduce the number of keys in y

y->nKeys = t - 1;

// Since this node is going to have a new child,

// create space of new child

for (int j = nKeys; 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 location of

// new key and move all greater keys one space ahead

for (int j = nKeys-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

nKeys++;

}

//A function to remove the index key from the node. Leaf node

template<class keyType>

void Node<keyType>::removeFromLeaf(int index)

{

//move all the keys after the index pos one place backwards

for(int i=index+1; i<nKeys; ++i)

keys[i-1] = keys[i];

//reduce the count of keys

nKeys--;

return;

}

//A function to remove the index key from the node

//which is a non leaf node

template<class keyType>

void Node<keyType>::removeFromNonLeaf(int index)

{

keyType k = keys[index];

//if the child that precedes k C[index] has at least t keys,

//find the predecesor of k in the subtree rooted at

//c[index]. Replace k by pred. Recuively delete pred in c[index

if(C[index]->nKeys >= t)

{

keyType pred = getPred(index);

keys[index] = pred;

C[index]->remove(pred);

}

//if the child c[index] has less than t keys, check c[index+1].

//if c[index+1] has at least t keys, find the successor of k in

//the subtree rooted at c[index+1].Replace k by succ, Recusively

//delete succ in C[index+1]

else if (C[index+1]->nKeys >= t)

{

keyType succ = getSucc(index);

keys[index] = succ;

C[index + 1]->remove(succ);

}

//if both c[index] and c[index+1] has less than t keys, merge k

//and all of the c[index+1] into c[index]. C[index] will contain

//2t-1 keys. Free c[index+1] and rescursivly delete k from c[index].

else

{

merge(index);

C[index]->remove(k);

}

return;

}

//A function to get predecessor of keys[index]

template<class keyType>

keyType Node<keyType>::getPred(int index)

{

//key moving to the right most node until we reach a leaf

Node<keyType> \*cur=C[index];

while(!cur->isLeaf)

cur = cur->C[cur->nKeys];

//return the last key of the leaf

return cur->keys[cur->nKeys-1];

}

//A function to get successor of key[index]

template<class keyType>

keyType Node<keyType>::getSucc(int index)

{

//keep moving the left most node starting from C[index+1] until

//we reach a leaf

Node<keyType> \*cur=C[index+1];

while(!cur->isLeaf)

cur = cur->C[0];

//return the first key of the leaf

return cur->keys[0];

}

//A function to merge c[index] with c[index+1]

//c[index+1 is freed after merge

template<class keyType>

void Node<keyType>::merge (int index)

{

Node<keyType> \*child = C[index];

Node<keyType> \*sibling = C[index+1];

//Pulling a key from the current node and inserting it into t-1

//pos of c[index]

child->keys[t-1] = keys[index];

//copying the keys from c[index+1] to c[index] at the end

for(int i = 0; nKeys; ++i)

child->keys[i+t] = sibling->keys[i];

//Copying the child pointers from c[index+1] to c[index]

if(!child->isLeaf)

{

for(int i = 0; i <=sibling->nKeys; ++i)

child->C[i+t] = child->C[i];

}

//Moving all keys after index in the current node one step before

//to fill the gap created by moving keys[index] to c[index]

for(int i = index+1; i<nKeys; ++i)

keys[i-1] = keys[i];

//moving the child pointers after index+1 in the current node one

//step before

for(int i = index+2; i<=nKeys; ++i)

C[i-1] = C[i];

//updating the key count of child and the current node

child->nKeys += sibling->nKeys+1;

nKeys--;

//freeing the memory occupied by sibling

delete(sibling);

return;

}

//A function to fill child C[index] which has less than t-1 keys

template<class keyType>

void Node<keyType>::fill(int index)

{

//if the previous child(c[index-1]) has more than t-1 keys,

//borrow a key from that child

if(index != 0 && C[index-1]->nKeys>=t)

promoteFromPrev(index);

//if the next child(c[index+1]) has more than t-1 keys, borrow

//a key from that child

else if(index != nKeys && C[index+1]->nKeys>=t)

promoteFromNext(index);

//merge c[index] with its sibling. if c[index] is the last child,

//merge it with its previous sibling. Otherwise merge it with its

//next sibling

else

{

if(index != nKeys)

merge(index);

else

merge(index-1);

}

return;

}

//A function to borrow a key from c[index-1] and insert it into

//c[index]

template<class keyType>

void Node<keyType>::promoteFromPrev(int index)

{

Node \*child = C[index];

Node \*sibling = C[index - 1];

//the last key from c[index-1] goes up to the parent and

//key[index-1] from parent is inserted as the first key in

//c[index]. thus, loses sibling one key and child gains one

//key. Moving all key in c[index] one step ahead

for(int i = child->nKeys - 1; i >= 0; --i)

child->keys[i+1] = child->keys[i];

//if c[index] is not a leaf, move all its child pointers one

//step ahead

if(!child->isLeaf)

{

for(int i = child->nKeys; i >= 0; --i)

child->C[i+1] = child->C[i];

}

//setting childs first key equal to keys[index-1] from the current

//node

child->keys[0] = keys[index - 1];

//moving siblings last child as c[index]s fisrt child

if(!child->isLeaf)

child->C[0] = sibling->C[sibling->nKeys];

//moving the key from the sibling to the parent. this reduces

//the nmber of keys in the sibling

keys[index-1] = sibling->keys[sibling->nKeys-1];

child->nKeys += 1;

sibling->nKeys -= 1;

return;

}

//A function to borrow a key from the c[index+1] and place it in

//c[index]

template<class keyType>

void Node<keyType>::promoteFromNext(int index)

{

Node \*child = C[index];

Node \*sibling = C[index + 1];

//keys[index] is inserted as the last key in c[index]

child->keys[(child->nKeys)] = keys[index];

//isblings first child is inserted as the last child into

//c[index]

if(!child->isLeaf)

child->C[(child->nKeys) + 1] = sibling->C[0];

//The first key from sibling is inserted into keys[index]

keys[index] = sibling->keys[0];

//moving all keys in sibling one step behind

for(int i = 1; i < sibling->nKeys; ++i)

sibling->keys[i-1] = sibling->keys[i];

//Moving the child pointers one step behind

if(!child->isLeaf)

{

for(int i = 1; i <= sibling->nKeys; ++i)

sibling->C[i-1] = sibling->C[i];

}

//increasing and decreasing the key count of c[index] and

//c[index+1] respectively

child->nKeys += 1;

sibling->nKeys -= 1;

return;

}

//A function to remove the key k from the sub tree rooted with

//this node

template<class keyType>

void Node<keyType>::remove(keyType k)

{

int index = findKey(k);

//the key to be removed is present in this node

if(index < nKeys && keys[index] == k)

{

//if the node is leaf removeFromLeaf is called

//else removefromNonLeaf is called

if(isLeaf)

removeFromLeaf(index);

else

removeFromNonLeaf(index);

}

else

{

//if this node is a leaf, then the key is not present in tree

if(isLeaf)

{

cout << "The key " << k << " not found in the tree\n";

return;

}

//the key to be removed is present in the sub tree rooted with

//this node. the flag indicates wheter the key is present in

//the subtree rooted with the last child of this node

bool isLast = ((index==nKeys) ? true : false);

//if the child where the key is supposed to exist has less than

//t keys, we fill the child

if(C[index]->nKeys < t)

fill(index);

//if the last child has been merged, it must have merged with the

//previous child and so recurse one the index-1 child. else, we

//recuse on the index child which now has at least t keys

if(isLast && index > nKeys)

C[index-1]->remove(k);

else

C[index]->remove(k);

}

return;

}

//A utility function that returns the index of the first key that is

//greater than or equal to k

template<class keyType>

int Node<keyType>::findKey(keyType k)

{

int index = 0;

while(index < nKeys && keys[index] < k)

++index;

return index;

}

//A function to remove the key k from the subtree

template <class keyType>

void BTree<keyType>::remove(keyType k)

{

if(!root)

{

cout << "Tree empty\n";

return;

}

//Call the remove function for root

root->remove(k);

//if the root node has 0 keys, make its first child as the new

//root. if it has a child, otherwise set root as null

if(root->nKeys == 0)

{

Node<keyType> \*tmp = root;

if(root->isLeaf)

root = NULL;

else

root = root->C[0];

//free old root

delete tmp;

}

return;

}

// Driver program to test above functions

int main()

{

BTree<string> t(3); // A B-Tree with minimum degree 3, order 6

string text = "In computer science, a B-tree is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree is a generalization of a binary search tree in that a node can have more than two children. Unlike self-balancing binary search trees, the B-tree is well suited for storage systems that read and write relatively large blocks of data, such as discs. It is commonly used in databases and file systems. In B-trees, internal (non-leaf) nodes can have a variable number of child nodes within some pre-defined range. When data is inserted or removed from a node, its number of child nodes changes. In order to maintain the pre-defined range, internal nodes may be joined or split. Because a range of child nodes is permitted, B-trees do not need re-balancing as frequently as other self-balancing search trees, but may waste some space, since nodes are not entirely full. The lower and upper bounds on the number of child nodes are typically fixed for a particular implementation. For example, in a 2-3 B-tree (often simply referred to as a 2-3 tree), each internal node may have only 2 or 3 child nodes.";

//using stream to parse paragraph and insert int tree.

//tree is then traversed and printed. Next keys are removed

//from tree and then traversed and printed again

istringstream str(text);

string temp = "";

while(str >> temp)

t.insert(temp);

t.traverse();

cout << endl << endl;

t.remove("B-trees,");

t.remove("nodes.");

t.remove("range");

t.remove("tree");

t.remove("tree),");

t.remove("trees,");

t.remove("changes.");

t.remove("space,");

t.remove("data,");

t.remove("example,");

t.remove("data,");

t.remove("example,");

t.remove("searches,");

t.remove("range,");

t.remove("insertions,");

cout << endl;

t.traverse();

cout << endl;

return 0;

}

**Screen shots**

