Name: Ken Lin

SID: 006198682

Github repository of my code : <https://github.com/DehNutCase/CSE-461/tree/master/lab2>

I have successfully implemented the remove function for the B-tree, so all parts are finished successfully.

20 Points

**Report:**

Comparison of the outputs and expected outputs are identical.

Script of running the code:

Script started on 2020-04-25 02:43:21-07:00 [TERM="xterm" TTY="/dev/pts/0" COLUMNS="99" LINES="31"]

]0;006198682@csusb.edu@jb358-7:~/cse461

[006198682@csusb.edu@jb358-7 cse461]$ cd lab2[K[K[K[K[K[K[K

cd lab2

]0;006198682@csusb.edu@jb358-7:~/cse461/lab2

[006198682@csusb.edu@jb358-7 lab2]$ cd btree

]0;006198682@csusb.edu@jb358-7:~/cse461/lab2/btree

[006198682@csusb.edu@jb358-7 btree]$ ls

btree btree.cpp [0m[01;32mlab2[0m

]0;006198682@csusb.edu@jb358-7:~/cse461/lab2/btree

[006198682@csusb.edu@jb358-7 btree]$ ./lab2

Traversal of the constucted tree is:

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

The key example, not found in the tree

Traversal of the tree after removing certain keys is:

(non-leaf) (often 2 2-3 3 B-tree B-trees Because For In It The Unlike When a access, allows and are as be binary blocks bounds but can child children. commonly computer data databases deletions discs. do each entirely file fixed for frequently from full. generalization have implementation. in inserted internal is its joined large logarithmic lower maintain maintains may more need node nodes not number of on only or order other particular permitted, pre-defined range re-balancing read referred relatively removed science, search self-balancing sequential simply since some sorted split. storage structure such suited systems systems. than that the time. to tree two typically upper used variable waste well within write]0;006198682@csusb.edu@jb358-7:~/cse461/lab2/btree

[006198682@csusb.edu@jb358-7 btree]$ exit

Script done on 2020-04-25 02:43:50-07:00 [COMMAND\_EXIT\_CODE="0"]

**Source Code:**

Also inside the github repository.

// C++ program for B-Tree insertion

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

#include<iostream>

#include<string.h>

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.

//KL, function decs

void promoteFromNext(int index);

void promoteFromPrev(int index);

void merge (int index);

keyType getSucc(int index);

keyType getPred(int index);

void removeFromLeaf (int index);

void removeFromNonLeaf (int index);

void fill (int index);

void remove ( keyType k );

int findKey(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 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++;

}

//KL, modifications start here

template<class keyType>

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

{

// Shift all the keys after the index position one place

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

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

// Reduce the count of keys

nKeys -= 1;

return;

}

template<class keyType>

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

{

keyType k = keys[index];

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

// find the predecessor 'pred' of k which is the rightmost key of

// the subtree rooted at

// C[index]. Replace k by pred and delete the rightmost key, which

// is at a leaf ( calling remove() recursively)

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

keyType pred = getPred(index);

keys[index] = pred;

C[index]->remove(pred);

}

// If the child C[index] has less that t keys, examine C[index+1].

// If C[index+1] has atleast t keys, find the successor 'succ' of k in

// the subtree rooted at C[idx+1]

// Replace k by succ and remove 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 that t keys,merge k and all of C[index+1]

// into C[index]

// Now C[index] contains 2t-1 keys

// Free C[index+1] and remove k from C[index]

else

{

merge(index);

C[index]->remove(k);// remove k from C[index]

}

return;

}

// Get predecessor of keys[index]

template<class keyType>

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

{

// Keep moving to the rightmost node until we reach a leaf

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

while (!cur->isLeaf)

cur = cur->C[cur->nKeys]; // rightmost child pointer

// cur now points to a leaf node

// Return the last key (rightmost, at position cur->nKeys-1) of the leaf

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

}

//Get successor of keys[index]

template<class keyType>

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

{

// Keep moving the leftmost 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 (leftmost) of the leaf

return cur->keys[0];

}

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)th

// position 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; i < sibling->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)

//KL, double check index

child->C[i+t+1] = sibling->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)

//KL, double check

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 node that has less than t-1 keys after deletion

template<class keyType>

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

{

// If the previous child(C[index-1]) has more than t-1 keys, promote 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, promote 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 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 promote a key from C[index-1] and insert it

// into C[index]

template< class keyType>

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

{

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

Node< keyType> \*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, the 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 child's first key equal to keys[index-1] from the current node

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

// Moving sibling's last child as C[index]'s first child

if(!child->isLeaf)

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

// Moving the key from the sibling to the parent

// This reduces the number of keys in the sibling

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

child->nKeys += 1;

sibling->nKeys -= 1;

return;

}

// A function to promote a key from the C[index+1] and place

// it in C[index]

template< class keyType>

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

{

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

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

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

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

// Sibling's 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 (!sibling->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;

}

template<class keyType>

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

{

// Find the first key >= k

int i = 0;

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

i++;

return i;

}

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 (isLeaf) // The node is a leaf

removeFromLeaf(index);

else // The node is an internal node

removeFromNonLeaf(index);

} else { // The key is not in the node, but in a descendant

// If this node is a leaf node, 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 at this node

// The flag isLast indicates whether the key is present in the sub-tree rooted

// at the last child of this node

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

// If the child where the key is supposed to exist is underflow,

// we fill that child

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

fill(index);// call a function to fill the child

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

// child and so we recurse on the (index-1)th child. Else, we recurse on the

// (index)th child which now has atleast t keys

if (isLast && index > nKeys)

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

else

C[index]->remove(k);

}

return;

}

template <class keyType>

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

{

if (!root) {

cout <<"Tree empty\n";

return;

}

// Call the remove function for root node

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 the old root

delete tmp;

}

return;

}

// Driver program to test above functions

int main()

{

/\*

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

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 constucted tree is ";

t.traverse();

int k = 6;

(t.search(k) != NULL)? cout << "\nPresent" : cout << "\nNot Present";

k = 15;

(t.search(k) != NULL)? cout << "\nPresent" : cout << "\nNot Present";

\*/

/\*

if (!search(input)){

}

\*/

BTree<string> t(3);

//prasing inputs into tokens

char inputs[] = "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.";

char \*token;

token = strtok(inputs, " ");

//making sure we only insert if the token isn't already in the tree

while (token != NULL){

if (t.search(token) == NULL){

t.insert(token);

}

token = strtok(NULL, " ");

}

cout << "Traversal of the constucted tree is:\n";

t.traverse();

char to\_remove[] = "B-trees, nodes. node, range. tree), trees, changes. space, data, example, data, example, searches, range, insertions,";

token = strtok(to\_remove, " ");

while (token != NULL){

t.remove(token);

token = strtok(NULL, " ");

}

cout << "Traversal of the tree after removing certain keys is:\n";

t.traverse();

return 0;

}