Jeremy Noble, Taylor Pedretti
CSE 461
Lab Report

Lab Report 2

Write Up:

After working on the B-Tree we can see why it is used in file systems and databases, and this reason is because there is only one root node and everything else is just a leaf of that root node, so searching, and removing things can be done easier. Maybe the B in B-Tree stands for Bush because the tree will look more like a bush when it starts to get full. Getting the remove to work was a little bit of a pain because the tree would a lot of the times need to be rebalanced out again and you want to make sure nodes are getting removed correctly or you'd have memory leaking. Overall this lab was fun to do.

We feel like we earned the full 20 points because everything is correct and every part of the lab is fulfilled. Along with having to write the findKey() function on our own, we also took the extra step to do file input for the tree keys instead of simply tediously adding each each using the insert() function. We think this lab is solid.

Source Code:

```
// C++ program for B-Tree implementation
#include<iostream>
#include<string>
#include<fstream>
#include<cassert>
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.

    void removeFromLeaf(int index);
    void removeFromNonLeaf (int index);
    keyType getPred(int index);
    keyType getSucc(int index);
    void merge(int index);
    void fill(int index);
    void promoteFromPrev(int index);
    void promoteFromNext(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(); } // recursive

    // 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++;
}

// delete key from leaf
// FINISHED
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];

        this->nKeys--; // reduce number of keys in node
        return;
}

// delete keys from non-leaf
// FINISHED
template<class keyType>
void Node<keyType>::removeFromNonLeaf (int index)
{
        keyType k = keys[index];

        if (C[index]->nKeys >= t) {
                keyType pred = getPred(index);
                keys[index] = pred;
                C[index]->remove(pred);
        }
        else if (C[index + 1]->nKeys >= t) {
                keyType succ = getSucc(index);
                keys[index] = succ;
                C[index+1]->remove(succ);
        }
        else {
                merge(index);
                C[index]->remove(k);
        }
        return;
}

// get predecessor
template<class keyType>
keyType Node<keyType>::getPred(int index)
{
        Node<keyType> *cur=C[index];

        while(!cur->isLeaf) {
                cur = cur->C[cur->nKeys];
```

```
	}

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

}

// get successor
template<class keyType>
keyType Node<keyType>::getSucc(int index)
{
	Node<keyType> *cur = C[index+1];

	while(!cur->isLeaf) {
		cur = cur->C[0];
	}

	return cur->keys[0];
}

//merge
template<class keyType>
void Node<keyType>::merge(int index)
{
	Node<keyType> *child = C[index];
	Node<keyType> *sibling = C[index+1];

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

	for(int i = 0; i < sibling->nKeys; ++i) {
		child->keys[i+t] = sibling->keys[i];
	}

	if(!child->isLeaf) {
		for(int i=0; i <=sibling->nKeys; ++i) {
			child->C[i+t] = sibling->C[i];
		}
	}

	for(int i = index+1; i < nKeys; ++i) {
		keys[i-1] = keys[i];
	}

	for (int i = index+2; i <= nKeys; ++i) {
		C[i-1] = C[i];
	}

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

```
        delete(sibling);
        return;
}

// fill
template<class keyType>
void Node<keyType>::fill(int index)
{

    if (index!=0 && C[index-1]->nKeys>=t) {
        promoteFromPrev(index);
        }
        else if (index!=nKeys && C[index+1]->nKeys>=t) {
        promoteFromNext(index);
        }
    else
    {
              if (index != nKeys)
                     merge(index);
        else
            merge(index-1);
    }

        return;
}

// promote from previous
template< class keyType>
void Node< keyType>::promoteFromPrev(int index)
{

    Node< keyType> *child=C[index];
    Node< keyType> *sibling=C[index-1];

    for (int i=child->nKeys-1; i>=0; --i)
        child->keys[i+1] = child->keys[i];

    if (!child->isLeaf)
    {
        for(int i=child->nKeys; i>=0; --i)
            child->C[i+1] = child->C[i];
    }

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

    if(!child->isLeaf)
        child->C[0] = sibling->C[sibling->nKeys];

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

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

    return;
}

// promote from next
template< class keyType>
void Node< keyType>::promoteFromNext(int index)
{

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

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

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

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

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

    if (!sibling->isLeaf)
    {
        for(int i=1; i <=sibling->nKeys; ++i)
            sibling->C[i-1] = sibling->C[i];
    }

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

    return;
}

// find key
template<class keyType>
int Node<keyType>::findKey(keyType k)
{
    int index = 0;
    while (index < nKeys && keys[index] < k)
        ++index;
    return index;
}

template<class keyType>
void Node<keyType>::remove (keyType k)
```

```
{
        int index = findKey(k);

    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);

          // 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;
}

// test b-tree
int main()
{
    BTree<string> st(3);
    fstream input;
    input.open("keys.txt");
    assert(input.is_open());
    string temp = "";

    while(input>>temp) {
              // prevent duplicate keys
              if(st.search(temp) == NULL) {
                     st.insert(temp);
              }
       }

       cout << "Traversal of the constucted tree is ";
    st.traverse();
    cout << endl << endl;

    st.remove("B-trees");
    st.remove("nodes.");
    st.remove("node,");
    st.remove("range.");
    st.remove("tree),");
    st.remove("trees,");
    st.remove("changes.");
    st.remove("space,");
    st.remove("data,");
```

```
    st.remove("example,");
    st.remove("searches,");
    st.remove("range,");
    st.remove("insertions,");
    st.remove("data,");
    st.remove("example,");

    cout << "\n\nTraversal of the constucted tree is ";
    st.traverse();

        return 0;
}
```

Script Output:

```
Script started on 2019-01-28 17:46:20-0800
jeremy@jeremy-XPS-15-9550~/Desktop/CSE 461/Lab 2 $ ./btree

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 write

The key data, not found in the tree
The key example, not found in the tree


Traversal of the constucted tree 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;jeremy@jeremy-XPS-15-9550: ~/Desktop/CSE 461/Lab
```

```
jeremy-XPS-15-9550~/Desktop/CSE 461/Lab 2 $ exit
exit

Script done on 2019-01-28 17:46:26-0800
```