Yousef Jarrar, Jose Perez

CSE 461

Lab 2

20 Total Points

Lab 2: B-Tree

1. B-Tree.cpp

```
/* Yousef Jarrar, Jose Perez
Lab 2 - bTrees and remove Function (nonleaf nodes)
We are to implement the remove() function and to test it against the paragraph
provided to use from Lab 2.

The outputs of the data, are to be similar to that of Lab 2.
Comments have been added to keep track of what is being done.

*/


// C++ program for B-Tree insertion
// For simplicity, assume order m = 2 * t
#include <iostream>
#include <string>
#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

private:
    // Removes key at specific index from this node if it's leaf node
    void removeFromLeaf (int index);

    // Removes key at specific index from this node if it's not a leaf node
    void removeFromNonLeaf (int index);

    // Returns the predecessor of keys[index]
keyType getPred(int index);

    // Returns the successor of keys[index]
keyType getSucc(int index);

    // Merges index node with the next one
void merge (int index);

    // A function to fill child node that has less than t-1 keys after deletion
void fill (int index);

    // Removes key k from the sub-tree rooted at this node
void remove ( keyType k );

    // Returns the index of the first key that is >= k
int findKey( keyType k );

    // Promotes key from C[index - 1] to C[index]
void promoteFromPrev(int index);

    // Promotes key from C[index + 1] to C[index]
void promoteFromNext(int index);
 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.
```

```
// 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);

    // Removes key k from this B-tree
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
```

[redacted]

```
    // 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++;
}
```

```
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--;
}

// Removes key at specific index from this node if it's not a leaf
node 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]
```

```
// Removes key at specific index from this node if it's leaf node
```

```
else
















  {
        merge(index);
        // remove k from C[index]
        C[index]->remove(k);
    }
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     if (cur->nKeys > 0)         return cur-
>keys[0];     else
        return NULL;
}

// Merges the nodes 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 < 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] =
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];
    // Similarly, move children after index + 1
for (int i = index + 2; i <= nKeys; ++i)
        C[i - 1] = C[i];

    // Update sizes     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;
}

// Returns the index of the first key that is >= k
```

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

// Removes key k from the sub-tree rooted at 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 (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;
}

// Removes key k from this B-tree
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;
}

// Promotes key from C[index-1] to C[index]
template<class keyType> void
Node<keyType>::promoteFromPrev(int index)
{

    Node<keyType> *destination = C[index];
    Node<keyType> *source = C[index - 1];

    // Greatest key from C[index - 1] goes to parent
    // key[index - 1] from parent goes as first to C[index]
    // Moving all keys in C[index] one step forward
for (int i = destination->nKeys - 1; i >= 0; --i)
destination->keys[i + 1] = destination->keys[i];
    // If C[index] is not a leaf, move all its children one step forward
if (!destination->isLeaf)
    {          for (int i = destination->nKeys; i >=
0; --i)              destination->C[i + 1] =
destination->C[i];     }

    // Set C[index] first key to key[index - 1]     destination-
>keys[0] = keys[index - 1];

    // Set C[index] first child to last child of C[index -
1]     if(!destination->isLeaf)         destination->C[0] =
source->C[source->nKeys];

    // Move the greatest key from C[index - 1] to the parent
keys[index - 1] = source->keys[source->nKeys - 1];

    // Update key counts     destination-
>nKeys++;     source->nKeys--;
}

// Promotes key from C[index + 1] to C[index]
template<class keyType>
void Node<keyType>::promoteFromNext(int index)
{

    Node<keyType> *destination = C[index];
    Node<keyType> *source = C[index + 1];

    // keys[index] is inserted as the last key in
C[index]     destination->keys[destination->nKeys] =
keys[index];
    // Insert C[index + 1]'s first child as the last child of
C[index]     if (!destination->isLeaf)         destination-
>C[(destination->nKeys) + 1] = source->C[0];
    // Insert first key from C[index + 1] as last key of C[index]
    keys[index] = source->keys[0];
    // Move keys in C[index + 1] one step back
for (int i = 1; i < source->nKeys; ++i)
source->keys[i - 1] = source->keys[i];

    // Move children one step back
if (!source->isLeaf)
    {          for(int i = 1; i <= source-
>nKeys; ++i)             source->C[i - 1] =
source->C[i];
    }

    // Update key counts     destination-
>nKeys++;     source->nKeys--;
}

// program to test removal functions int
main()
{
    // Sample text data from the handout     string input = "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 predefined
\
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.";
    BTree<string> t(3); // A B-Tree with degree 3

    // Build tree with unique words from input
cout<< endl << endl;
     stringstream
inputStream(input);     while
(!inputStream.eof())
    {         string word;
inputStream >> word;         if
(t.search(word) == NULL)
            t.insert(word);
    }

    // Print current tree state     cout << "Traversal of
the consticted tree is:" << endl;     t.traverse();
    cout << endl << endl;
    // Remove fixed words     string words[15] = {"B-trees,", "nodes.",
"node,", "range.", "tree),",
"trees,", "changes.", "space,",
        "data,", "example,", "data,", "example,", "searches,", "range,",
"insertions,"};     for (int i
= 0; i < 15; i++)
    {
        t.remove(words[i]);
    }      cout << endl
<< endl;

    // Print new tree state     cout << "Traversal
of the new tree is:" << endl;
    t.traverse();
    cout << endl << endl;
     return
0;
}
```

2. a. Remove Functions

```
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--;
}  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]
```

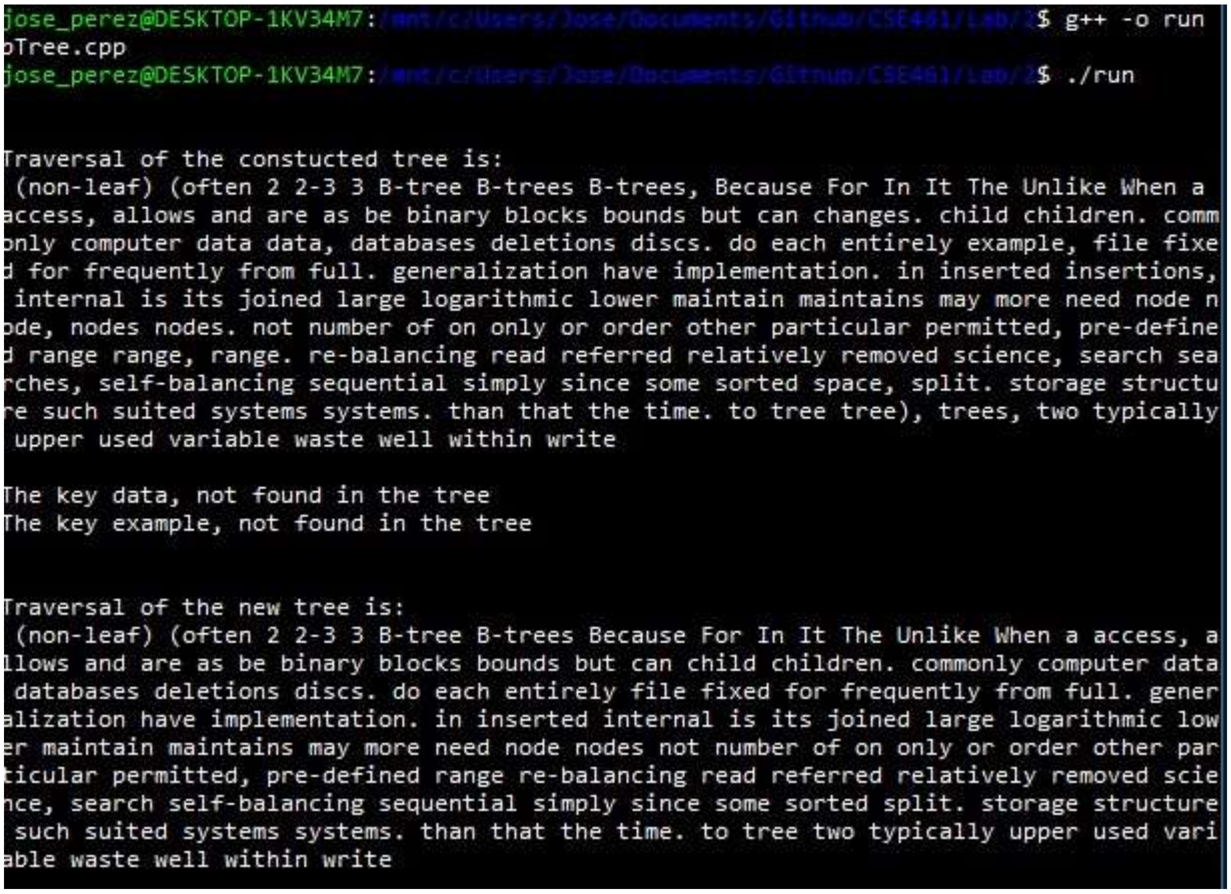
```
    // into C[index]
    // Now C[index] contains 2t-1 keys
    // Free C[index+1] and remove k from
C[index]     else     {          merge(index);
        // remove k from C[index]
        C[index]->remove(k);
    }
return;
}

// Removes key k from the sub-tree rooted at 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 (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;
}

// Removes key k from this B-tree
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;
}
```

c & d: Output screenshot shows both Traversing the tree and traversing the tree after deleting 15 keys.



```
jose_perez@DESKTOP-1KV34M7:/mnt/c/Users/Jose/Documents/Github/CSE461/Lab/2$ g++ -o run
bTree.cpp
jose_perez@DESKTOP-1KV34M7:/mnt/c/Users/Jose/Documents/Github/CSE461/Lab/2$ ./run


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. comm
only computer data data, databases deletions discs. do each entirely example, file fixe
d for frequently from full. generalization have implementation. in inserted insertions,
 internal is its joined large logarithmic lower maintain maintains may more need node n
ode, nodes nodes. not number of on only or order other particular permitted, pre-define
d range range, range. re-balancing read referred relatively removed science, search sea
rches, self-balancing sequential simply since some sorted space, split. storage structu
re 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 new tree is:
 (non-leaf) (often 2 2-3 3 B-tree B-trees Because For In It The Unlike When a access, a
llows 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. gener
alization have implementation. in inserted internal is its joined large logarithmic low
er maintain maintains may more need node nodes not number of on only or order other par
ticular permitted, pre-defined range re-balancing read referred relatively removed scie
nce, 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 vari
able waste well within write
```

Self-Evaluation:

We believe we should get 20 out of 20 points for this lab. We completed all that was requested from us during the lab. Within the lab report, we show our source code with the code given to us by the professor and a output that shows what is required. There was some trouble due to both of us not knowing what exactly what a b-tree, but after a bit of research and notes from the professor, we finished. The instructions were pretty clear and the file containing the classes allowed us to finish the lab.