B+ Tree Implementation

Fulps, Patrick
prfy9b@mst.edu

Ward, George
gcwzf4@mst.edu

November 14, 2018

Abstract

This paper describes the algorithms used for insertion and deletion in a B^+ Tree of order 4. There are many implementations published on the internet for both of these functions. Our implementation has been written in C++ and we have provided the pseudocode in this document to more easily follow along with the processes of both insertion and deletion into the B^+ Tree.

1 Introduction

The B+ Tree is used for dynamic multilevel indexing. It is designed to break data entries into nodes that hold keys represented by K and pointers represented by P. Keys are the values that are inserted and deleted from the tree, while the pointers are used for traversal through the tree. From Elmasri and Navathe (2016), we learned that there are internal nodes that hold key pointer pairs in the form $< P_1, K_1, P_2, K_2, ..., P_q1, K_{q-1}, P_q >$ and the order $K_1 < K_2 < .... < K_{q-1}$, and leaf nodes whose key pointer pairs are in the form $< K_1, Pr_1 >, < K_2, Pr_2 >, ... < K_{q-1}, Pr_{q-1} >, P_{next} >$, where Pr is a data pointer. In our implementation, we are using Pr as an identifier for a leaf, since all of our leaf pointers

point to NULL, and each leaf nodes keys are ordered $K_1 < K_2 < .... < K_{q-1}$. With this base idea, we are able to recursively search through the tree until a leaf node is reached. Leaf nodes are the final nodes in the tree and where the data is stored. We can then find where the key that has been given to perform insertion or deletion is.
After insertion or deletion is performed on the B^+ Tree, the tree may be left with oversized or empty nodes. This is an issue because the B in B^+ Tree stands for "Balanced". If nodes at height H are empty, then nodes from height H+1 must split, which will propagate upwards until the tree has returned to a balanced state. Similarly, when an insertion happens into a node that is full then the node must split to make room for more data to be inserted into the tree. If a Tree splits at height H, then the two children nodes will remain at height H. However, the

parent node will move up to H+1. This will continue until the tree is fully balanced.
This report covers the implementation of both insertion and deletion in the B^+ Tree, the logic and reasoning behind the choice made and a more abstract view for higher level thinking on the implementations chosen.

2 Approach

The approach chosen to make the program was to start with the high level thinking about the B^+ Tree structure. After careful thought and consideration we starting by building the classes of the Node and BPlusTree, then with ways of storing the data that would be given then there was a decision of how to search through the tree, we used the formula given by Elmasri and Navathe (2016) to perform this operation at the beginning of insertion and deletion functions. The priority of the search was to find a leaf node that contains the data that is being inserted or deleted. After finding a node then we worked on an insertion function, being as the deletion of any value is strictly dependent on a value actually being there insertion was the second function to naturally work on. After values have been inserted then the deletion was able to be performed. The order of writing our functions was first to implement insertion and then deletion. They are presented in this report in that order, with the pseudo and hard code being provided for each function and a high level overview of how we implemented our functions.

3 Insertion

For the insertion into a B^+-Tree is performed on the leafs of the tree. Firstly the input is compared with the keys that are currently in the node, using the formula $K1 < K2 < .... < K_{q-1}$ until a leaf is reached. Then the if the value is already inserted into the tree cannot be reinserted into the tree, if a value is passed into insert that is already in the tree there will be no change.If the leaf node is not full then the value is inserted the end of the node and then then node is sorted. If values the leaf that is being inserted into is full then the leaf then the splitLeaf(). This is the high level overview of insertion into a B^+ Tree. Different splits that can arise from insertion are shown in Apendies 6.1(Insertion into leaf), 6.2(Insertion Splitting a Leaf), 6.3(Insertion Splitting and Internal Node), and 6.7(Example of Program using Test Data).

3.1 Pseudo Code Insertion

```
insert( node, input)

  Node* node ← root (First node)

  while(node is not leaf)
    node ← node->pointer based on
    input to node->keys
    Look for input in node

  if(input already in node)
    output error
    return;
```

if(node is not full)
Move all values in node after position of input right
Insert input in space created

If(node is full)
splitLeaf(node, input)

return

3.2 Pseudo Code splitLeaf

Splitting a leaf is different that splitting an internal node, due to the fact that a leaf contains only one block pointer that points to the leaf next leaf on the same level. The function after a split requires a merge with an internal node, we did not write specific function for this instead we implemented the merging of the internal node in the splitLeaf function. This is how to perform the operation of split leaf. Example can be seen in Appendix 6.2(Insertion Splitting a Leaf)
splitLeaf(passed node, input)

splitLeaf(passed node, input)

Create newNode
put right 2 values in newNode, keep left 2 in node
If(parent of node is NULL)
Create new node
First key is last value of node
First 2 pointers are node and newNode
Root is this node

Else
Find position of node in pointers of parent

If parent is full
splitInternalNode(node->parent, input)

Else
Move parent pointers after child index to the right
Move parent keys after child index to the right
Insert left value of node in created key space
Insert pointer to newNode in created pointer space
return

3.3 Pseudo Code for splitInternalNode

Splitting an internal node requires special attention to the pointers due to the fact that the pointers are what leads the program to the leafs where the data is to be inserted into the tree. The pointers and key values must be handled meticulously to make sure that the traversal of the tree is correct. Present below is the pseudo code for writing an internal node split operation. Example can be seen in Appendix 6.3(Insertion Splitting and Internal Node)

splitInternalNode(node, input, child node, index)

```
Create newNode
Put right 2 values and right 3 pointers in
newNode

If(parent of node is NULL)
Create new node
  First key is last value of node
  First 2 pointers are node and newNode
  Root is this node

Else
  Find position of node in pointers of
  parent

If parent is full
  splitInternalNode(node->parent, input)

Else
  Move parent pointers after child index
  to the right
  Move parent keys after child index
  to the right
  Insert left value of node in created key
  space
  Insert pointer to newNode in created
  pointer space

return
```

4 Deletion

The function of deletion is to remove a key from the B^+ Tree. If there are any keys of the same value as input in the internal nodes of the tree, then that value must be replaced with the new rightmost value in the node. If input is not found in the node, the function will end since it cannot be removed. After the removal of a key it must check and see if there is an underflow in the leaf. If it is, then the leaf must be either merged with a sibling or take a value from one. We have constructed functions for the merging of leaves and for the merging of internal nodes. The merge operation is performed when a node has an underflow. Shown below is the pseudo code for both delete, mergeLeaf, and mergeInternalNode. Examples of deletion can be seen in Appendices 6.4(Delete from Leaf), 6.5(Delete from Internal Node), 6.6(Delete and Merge TwoLeafs), and 6.7(Example of Program using Test Data).

4.1 Pseudo Code Deletion

```
delete(input)

  Node* node = root (First node)
  while(node is not leaf)
    node = node->pointer based on
    comparing input to node->keys
  Look for input in node

  If(input not in node)
    output error
    return;

  If(node has more than 1 value)
    Remove input from node
    Move all values to the right left 1 space
    to close gap
```

```
If(node has 1 value)

  If(parent of node is NULL)
    Remove last value, to create
  empty root

  Else if(node is rightmost leaf)
    leafMerge(left leaf, node, first node
    key, nodeIndex)

  Else
    leafMerge(node, next leaf, first node
    key, nodeIndex)

print out tree
return
```

4.2 Pseudo Code mergeLeaf

Merging of a leaf is performed when there is an underflow in a leaf node. The merging of leaf will check its siblings to see if they have an extra value that can be moved. If so, that value will be shifted down the tree to the once empty node. If not, merge will be performed. After the merge is performed it will update the parent node with the last value in the node. The pseudo-code for the operation is shown below.

```
mergeLeaf(leftNode, rightNode,
input,indexToNode)

  Node* temp = whichever node contains
indent input, left or right
  Loop through all siblings to the right

If(sibling has more than 1 value)
  Use that sibling
  break loop;
Loop through all siblings to the left

If(sibling has more than 1 value)
  Use that sibling
  break loop;

If(sibling was found to the right)
  Loop through  of nodes between temp
  and sibling
  Move first value of sibling to node
  left of sibling
  Update respective parent key to that
  value

If(sibling was found to the left)
  Loop through  of nodes between temp
  and sibling
  Move last value of sibling to node right
  of sibling
  Update respective parent key to new
  sibling last value

If(no sibling found)
  Move values in right node to left node
  Remove parent key for right node,
  shift keys left
  Remove parent pointer to right node,
  shift pointers left
  If(parent is empty)
  internalMerge(parent)

delete right
return
```

4.3 Pseudo Code mergeInternalNode

Merging of an internal node is performed when there is an underflow in an internal node. The merging of internal node will check its siblings for values to be shifted. If none are found, merge will be performed. The merge and sibling shifts must account for pointers, unlike mergeLeaf. After the merge is performed it will update the parent node with the last value in the node. This function has a recursive call depending if the parent is empty for balancing purposes. The pseudo-code for the operation is shown below.

```
mergeInternalNode( node)

  If(parent is NULL)
    root = first pointer of node
    delete node
    return;
  Find indexToNode
  Loop through all siblings to the right

  If(sibling has more than 1 value)
    Use that sibling
    break loop;

  Loop through all siblings to the left
  If(sibling has more than 1 value)
    Use that sibling
    break loop;

  If(sibling was found to the right)
    Loop through  of nodes between temp
    and sibling
      Move first value of sibling to node left
      of sibling
      Move first pointer of sibling to node left
      of sibling
      Update respective parent key to that
      value


  If(sibling was found to the left)
    Loop through  of nodes between temp
and sibling
      Move last value of sibling to node
      right of sibling
      Move last pointer of sibling to node
      right of sibling
      Update respective parent key to new
      sibling last value


  If(no sibling found)
    Find right or left node to be merged
    with
    Left/right = node, whichever is not
    already assigned
    Move all values and pointers into left
    node
    Remove respective parent key and
    pointer to right
    Shift all other keys and pointers to the
    left


    If(parent is empty)
      mergeInternalNode(parent)
  delete right
  return
```

5 Conclusion

The B^+ Tree data structure is very efficient for retrieval of data, but implementation of the data structure takes a lot of time and thought to properly implement. If the order of the tree was higher, it would allow for extremely high fan-out. However, since p is only equal to 4, there is no real benefit to this B+ Tree compared to a B tree. In this paper we have demonstrated how to implement a B+Tree and given pseudocode for each of the major functions and a brief description of what the purpose of the function is. Attached to the document is the hard code for our program, and pictures of operations are shown in the Appendix.

6 Appendix

6.1 Image of Insertion to Leaf

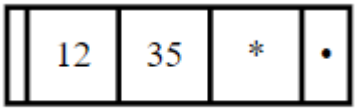

Figure 1: Node before insertion of 3

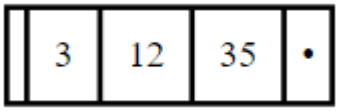

Figure 2: Node after insertion of 3

6.2 Image of Insertion Splitting a Leaf

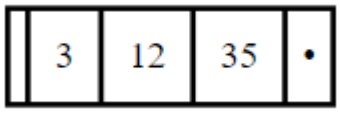

Figure 3: Node before insertion of 20

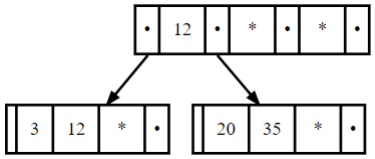

Figure 4: Node after insertion of 20

6.3 Image of Insertion Splitting an Internal Node

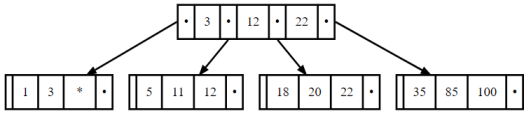

Figure 5: Node before insertion of 55

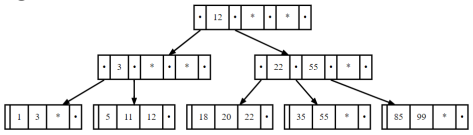

Figure 6: Node after insertion of 55

6.4 Image of Delete from Leaf

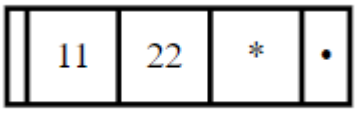

Figure 7: Node before deletion of 22

Figure 8: Node after deletion of 22

6.5 Image of Delete from Internal Node

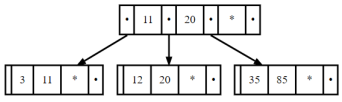

Figure 9: Node before deletion of 22

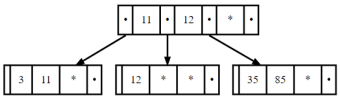

Figure 10: Node after deletion of 22

6.6 Image of Delete Key and Merge Two Leafs

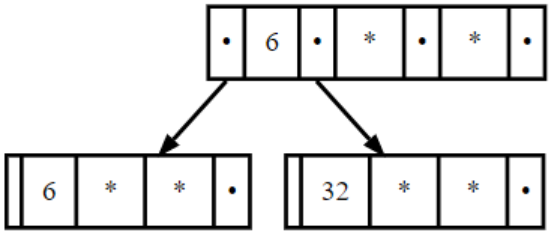

Figure 11: Node before deletion of 6

Figure 12: Node after deletion of 6

6.7 Example of Program Using Test Data

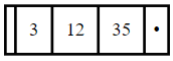

Figure 13: inserted 12,35,3

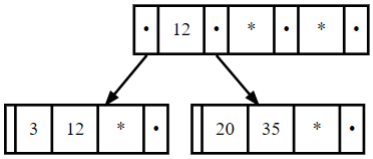

Figure 14: Inserted 20

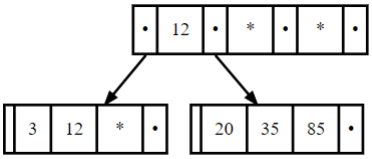

Figure 15: inserted 85

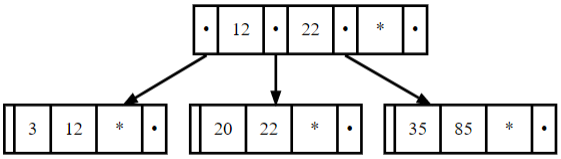

Figure 16: Inserted 22

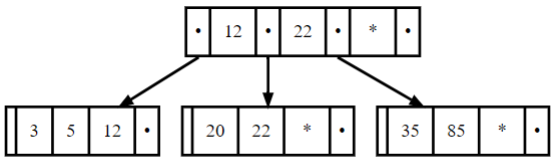

Figure 17: inserted 5

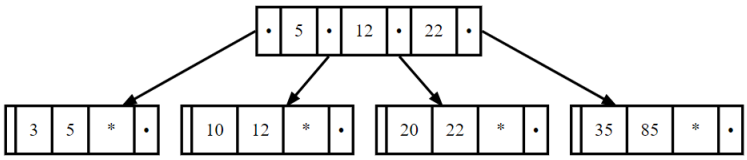

Figure 18: Inserted 10

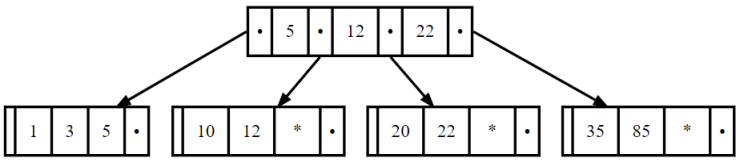

Figure 19: Inserted 1

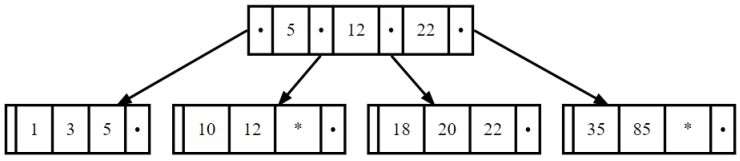

Figure 20: Inserted 18

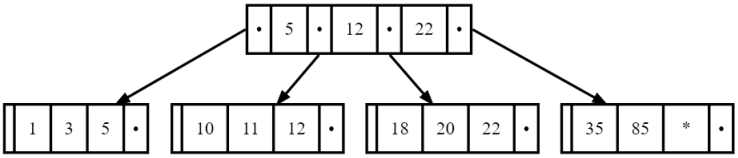

Figure 21: Inserted 11

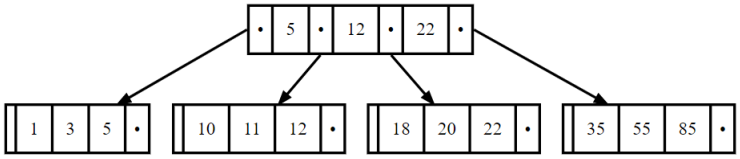

Figure 22: Inserted 55

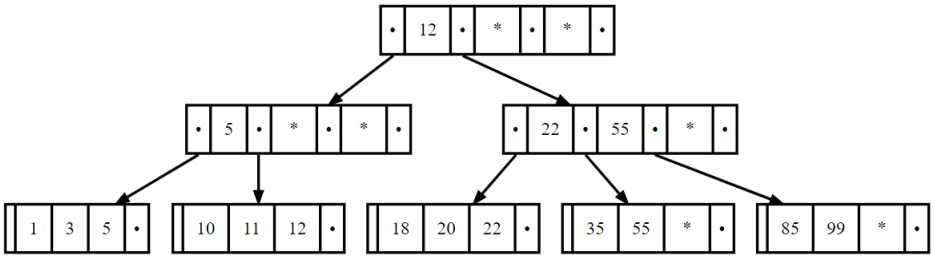

Figure 23: Inserted 99

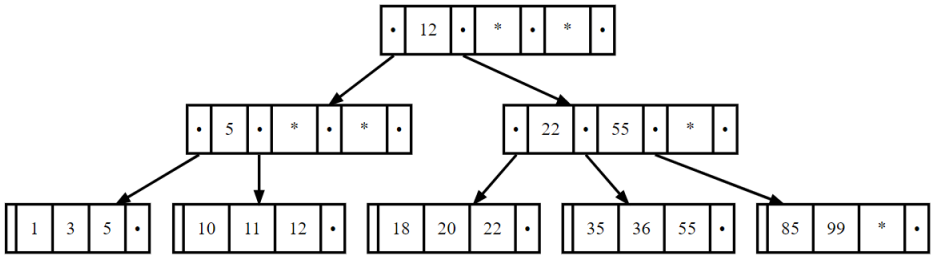

Figure 24: Inserted 36

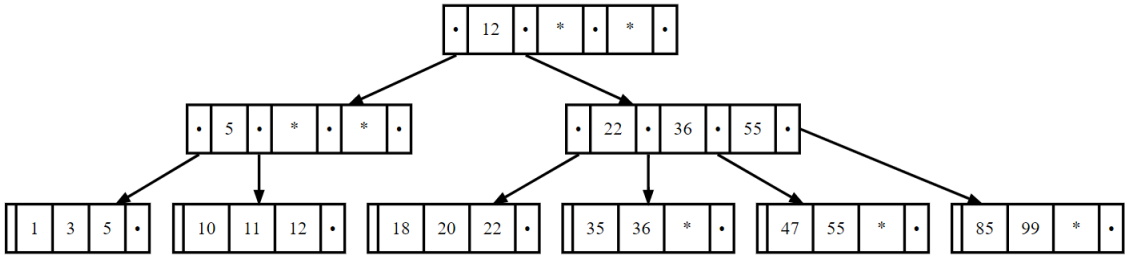

Figure 25: Inserted 47

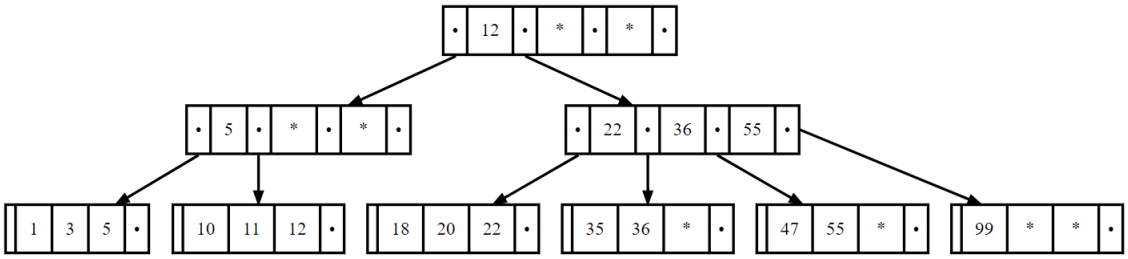

Figure 26: Deleted 85

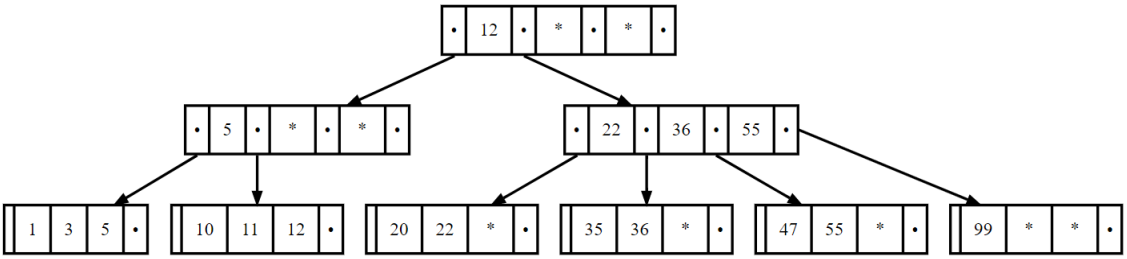

Figure 27: Deleted 18

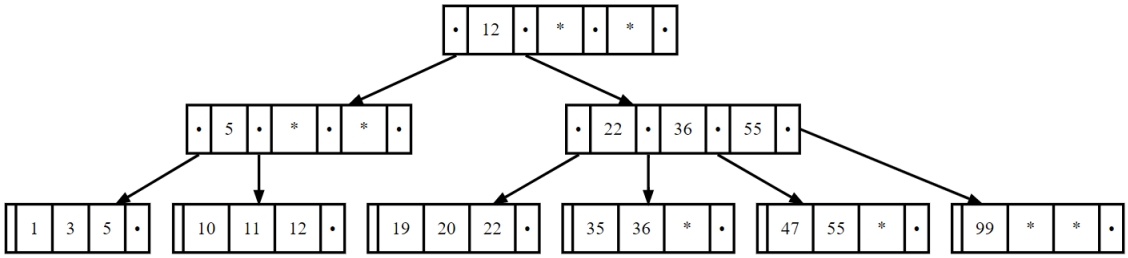

Figure 28: Inserted 19

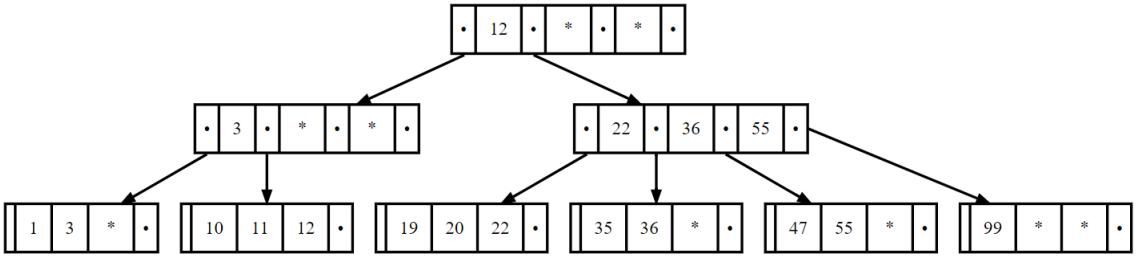

Figure 29: Deleted 5

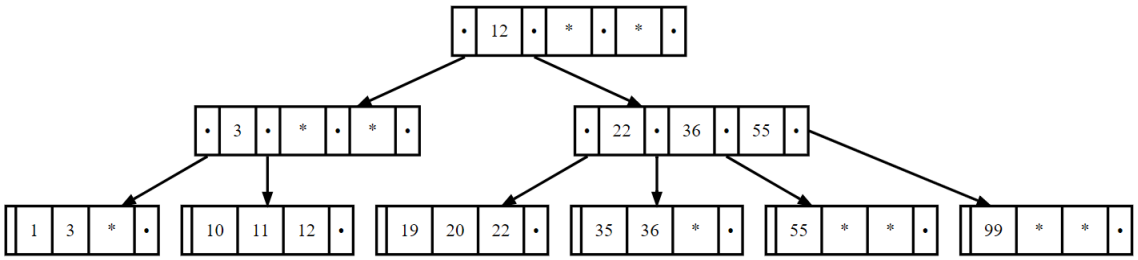

Figure 30: Deleted 47

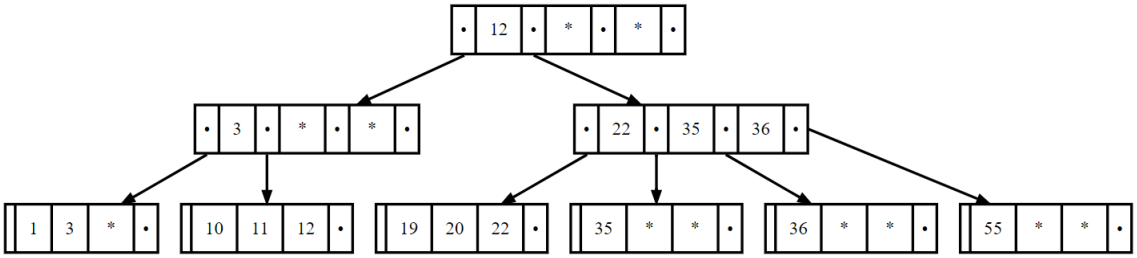

Figure 31: Deleted 99

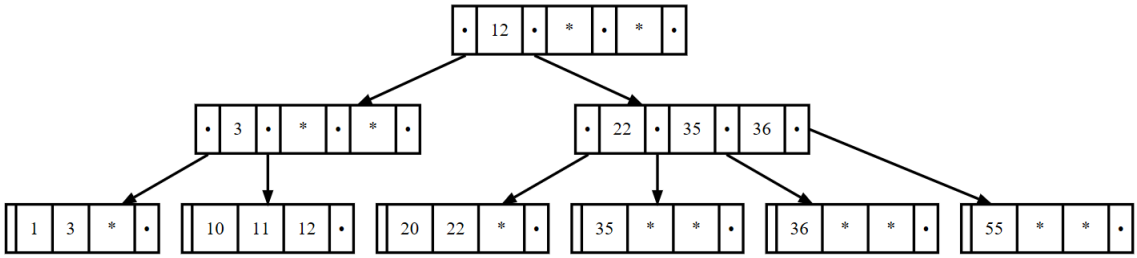

Figure 32: Deleted 19

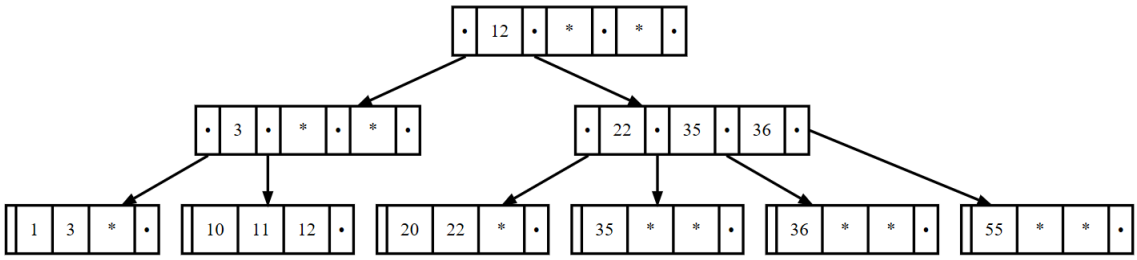

Figure 33: Inserted 11

References

[1] Ramez Elmazri and Shamkant B. Navathe. *Fundamentals of Database Systems*, University of Texas Arlington, 7nd edition, 2016.

7 Actual Program

```
#include <iostream>
#include <cmath>
#include "fstream"
#include <string>
using namespace std;


template <typename T>
void swap_val(T& val1,T& val2){
    T temp = val1;
    val1 = val2;
    val2 = temp;

    return;
}



class Node{
    public:

        //Variable used for telling whether or not
        //the node you're at is a leaf
        bool   leaf;

        //Array of pointer  to used for internal nodes
        Node** pointer;
        Node* next;
        Node* parent;

        //Dynamic array of elements that are contained
        //within the B+ Tree nodes
        int*   key;

        //Variables representing the maximum number
        //of pointers and elements able to be stored
        //in a particular node
        int    maxVals;
        int    maxPtrs;

        //Variables representing the actual number
        //of pointer  and elements that are currently
        //in the node
```

```
    int    actVals;
    int    actPtrs;

//This is a constructor for both a standard
//node of order 3 and an oversized node, to
//be used when the split function is called
Node(int input = 3){

    //Defaulting the key for a B+ Tree of p=num_vals
    leaf = true;
    next = NULL;
    maxVals = input;
    maxPtrs = maxVals+1;
    key = new int[maxVals];
    pointer = new Node*[maxVals+1];
    parent = NULL;
    actVals = 0;
    actPtrs = 0;

    //Default key to -1
    for(int i = 0; i < maxVals; i++){
        key[i] = -1;
    }

    //Default pointer  to NULL
    for(int i = 0; i < maxPtrs; i++){
        pointer [i] = NULL;
    }
}

Node(Node* src){

    this->leaf = true;
    this->maxVals = src->maxVals;
    this->actVals = src->actVals;
    this->maxPtrs = src->maxPtrs;
    this->actPtrs = src->actPtrs;
    this->key = new int[this->maxVals];
    this->pointer  = new Node*[this->maxPtrs];
    this->parent = NULL;

    for(int i = 0; i < src->maxVals; i++){
        this->key[i] = src->key[i];
        this->pointer [i] = src->pointer[i];
    }
```

```
        this->pointer[src->maxVals] = src->pointer[src->maxVals];

    }

   /* ~Node(){
            delete[] pointer;
            delete parent;
            delete next;
            delete key;

    }*/

};

void insertion_sort(Node* node)
{
    int i = 1;
    int j = 0;
    while(i < node->actVals)
    {
        j = i - 1;
        while((node->key[j] > node->key[j+1]) && (j >= 0))
        {
            swap_val(node->key[j], node->key[j+1]);
            j--;
        }
        i++;
    }
}

class BPlusTree
{
public:
    int numNodes;
    Node* root;

    BPlusTree(){
        numNodes = 1;
        root = new Node();
    }

    ~BPlusTree(){
       // root->ClearTree(root);
        delete root->key;
    }
```

```
void printLeaves()
{
    Node* node = root;
    cout << "Root Values: " << root->actVals << endl;
    cout << "Root:" << endl;
    for(int i = 0; i < root->actVals; i++)
        cout << "Value: " << root->key[i] << endl;
    while(!node->leaf)
        node = node->pointer[0];
    while(node != NULL)
    {
        cout << "Node:" << endl;
        for(int i = 0; i < node->actVals; i++)
            cout << "Value " << i << ": " << node->key[i] << endl;
        if(numNodes !=1)
        cout << endl;
        node = node->next;
    }
    cout << endl;
    return;
}

Node* findNextLeaf(Node* node)
{
    int value = node->key[node->actVals - 1];
    if(node->parent == NULL)
    {
        return NULL;
    }
    Node* temp = node->parent;
    int index;
    for(int i = 0; i < temp->actPtrs; i++){
        if(temp->pointer[i]->key[temp->pointer[i]->actVals-1] == value)
            index = i;
    }
    if(index == temp->actPtrs - 1)
    {
        return findNextLeaf(temp);
    }
    else
    {
        temp = temp->pointer[index+1];
        while(!temp->leaf)
            temp = temp->pointer[0];
```

```
        return temp;
    }
}
Node* splitLeaf(Node* node, int input)
{
    Node* parent = node->parent;
    int childPosition;
    Node* newNode    = new Node();
    newNode->parent = node->parent;
    node->next = newNode;

    //Splits the values between the 2 nodes
    if(input < node->key[0])
    {
        newNode->key[1] = node->key[2];
        newNode->key[0] = node->key[1];
        node->key[1] = node->key[0];
        node->key[0] = input;
    }
    else if(input < node->key[1])
    {
        newNode->key[1] = node->key[2];
        newNode->key[0] = node->key[1];
        node->key[1] = input;
    }
    else if(input < node->key[2])
    {
        newNode->key[1] = node->key[2];
        newNode->key[0] = input;
    }
    else
    {
        newNode->key[1] = input;
        newNode->key[0] = node->key[2];
    }
    node->key[2] =-1;
    node->actVals = 2;
    newNode->actVals = 2;

    //Makes new level if no parent
    if(node->parent == NULL)
    {
        Node* parent = new Node();
        parent->key[0] = node->key[1];
```

```
        parent->pointer[0] = node;
        parent->pointer[1] = newNode;
        parent->actVals = 1;
        parent->actPtrs = 2;
        parent->leaf = false;
        node->parent = parent;
        newNode->parent = parent;
        numNodes++;
        root = parent;
    }
    else
    {
        for(int i = 0; i < parent->actPtrs; i++)
            if(node->parent->pointer[i] == node)
                childPosition = i;

        //Adding mid val to parent, splitting if parent is full
        if(parent->actVals == parent->maxVals)
        {
            splitInternalNode(parent, node->key[1], newNode, childPosition + 1);
        }
        else
        {
            for(int i = parent->maxVals - 1; i > childPosition; i--)
                parent->key[i] = parent->key[i-1];
            parent->key[childPosition] = node->key[1];
            for(int i = parent->maxPtrs - 1; i > childPosition + 1; i--)
                parent->pointer[i] = parent->pointer[i-1];
            parent->pointer[childPosition + 1] = newNode;
            parent->actVals++;
            parent->actPtrs++;
        }
    }

    //Get index of original node in parent
    //cout << node->parent->key[0] << endl;
    numNodes++;
    newNode->next = findNextLeaf(newNode);
    return node;
}

Node* splitInternalNode(Node* node, int input, Node* childNode, int index)
{
    Node* parent = node->parent;
    int childPosition;
```

```
int mid;
Node* newNode    = new Node();
newNode->leaf = false;
newNode->parent = node->parent;

//Splits the values and pointers between the 2 nodes,
//including new node from last split
if(input < node->key[0])
{
    newNode->key[1] = node->key[2];
    newNode->key[0] = node->key[1];
    node->key[1] = node->key[0];
    node->key[0] = input;
    newNode->pointer[2] = node->pointer[3];
    newNode->pointer[1] = node->pointer[2];
    newNode->pointer[0] = node->pointer[1];
    node->pointer[1] = childNode;
}
else if(input < node->key[1])
{
    newNode->key[1] = node->key[2];
    newNode->key[0] = node->key[1];
    node->key[1] = input;
    newNode->pointer[2] = node->pointer[3];
    newNode->pointer[1] = node->pointer[2];
    newNode->pointer[0] = childNode;
}
else if(input < node->key[2])
{
    newNode->key[1] = node->key[2];
    newNode->key[0] = input;
    newNode->pointer[2] = node->pointer[3];
    newNode->pointer[0] = node->pointer[2];
    newNode->pointer[1] = childNode;
}
else
{
    newNode->key[1] = input;
    newNode->key[0] = node->key[2];
    newNode->pointer[2] = childNode;
    newNode->pointer[1] = node->pointer[3];
    newNode->pointer[0] = node->pointer[2];
}
mid = node->key[1];
node->key[1] = -1;
```

```
node->key[2] =-1;
node->pointer[3] = NULL;
node->pointer[2] = NULL;
node->actVals = 1;
node->actPtrs = 2;
newNode->actPtrs = 3;
newNode->actVals = 2;
newNode->pointer[0]->parent = newNode;
newNode->pointer[1]->parent = newNode;
newNode->pointer[2]->parent = newNode;
//Makes new level if no parent
if(parent == NULL)
{
    Node* parent = new Node();
    parent->key[0] = mid;
    parent->pointer[0] = node;
    parent->pointer[1] = newNode;
    parent->actVals = 1;
    parent->actPtrs = 2;
    parent->leaf = false;
    node->parent = parent;
    newNode->parent = parent;
    numNodes++;
    root = parent;
}
//Inserts mid value and pointer to new node to parent if not full
else
{
    //Get index of original node in parent
    for(int i = 0; i < node->parent->actPtrs; i++)
        if(node->parent->pointer[i] == node)
            childPosition = i;

    //Adding mid val to parent, splitting if parent is full
    if(parent->actVals == parent->maxVals)
    {
        splitInternalNode(parent, mid, newNode, childPosition + 1);
    }
    else
    {
        for(int i = parent->maxVals - 1; i > childPosition; i--)
            parent->key[i] = parent->key[i-1];
        parent->key[childPosition] = mid;
        for(int i = parent->maxPtrs - 1; i > childPosition + 1; i--)
            parent->pointer[i] = parent->pointer[i-1];
```

```
            parent->pointer[childPosition + 1] = newNode;
            parent->actVals++;
            parent->actPtrs++;
        }
    }

    numNodes++;
    return node;
}
Node* insert(int input)
{
    Node* node = root;
    int actual;
    bool found = false;
  //  int maximum;
    Node* temp = node;
    //Find leaf node
    while(!temp->leaf) {
        if(input < temp->key[0])
            temp = temp->pointer[0];
        else if(input < temp->key[1] || temp->key[1] == -1)
            temp = temp->pointer[1];
        else if(input < temp->key[2] || temp->key[2] == -1)
            temp = temp->pointer[2];
        else
            temp = temp->pointer[3];
    }

    for(int i = 0; i < temp->actVals; i++)
        if(temp->key[i] == input)
            found = true;
    if(found)
    {
        cout << "Value already in the tree, cannot insert " << input << endl;
        return node;
    }
    if(temp->maxVals == temp->actVals){
        splitLeaf(temp, input);
    }
    else {
        int position = temp->actVals;
        for(int i = temp->actVals - 1; i >= 0; i--)
        {
            if(input < temp->key[i])
                position = i;
```

```
        }
        for(int i = temp->actVals; i > position; i--)
            temp->key[i] = temp->key[i-1];
        temp->key[position] = input;
        temp->actVals++;
    }
    printLeaves();

    return temp;
}

void leafMerge(Node* left, Node* right, int input, int indexToNode)
{
    Node* sibling;
    Node* temp;
    bool siblingFound = false;
    bool toTheRight;
    int nodeGap = 0;
    if(left->key[0] == input)
        temp = left;
    else
        temp = right;
    for(int i = indexToNode + 1; i < temp->parent->actPtrs; i++)
    {
        sibling = temp->parent->pointer[i];
        if(sibling->actVals > 1)
        {
            siblingFound = true;
            toTheRight = true;
            nodeGap = i - indexToNode;
            break;
        }
    }
    sibling = temp->parent->pointer[indexToNode + nodeGap];
    if(!siblingFound)
    {
        for(int i = indexToNode - 1; i >= 0; i--)
        {
            sibling = temp->parent->pointer[i];
            if(sibling->actVals > 1)
            {
                siblingFound = true;
                toTheRight = false;
                nodeGap = indexToNode - i;
                break;
```

```
        }
    }
    sibling = temp->parent->pointer[indexToNode-nodeGap];
}
if(siblingFound && toTheRight)
{
    Node* leftSibling;
    for(int i = 0; i < nodeGap; i++)
    {
        leftSibling = temp;
        while(leftSibling->next != sibling)
            leftSibling = leftSibling->next;
        leftSibling->key[1] = sibling->key[0];
        leftSibling->actVals++;
        for(int j = 0; j < sibling->actVals - 1; j++)
            sibling->key[j] = sibling->key[j+1];
        sibling->key[sibling->actVals-1] = -1;
        sibling->actVals--;
        temp->parent->key[indexToNode + nodeGap - i - 1]
           = leftSibling->key[1];
        sibling = leftSibling;
    }
    temp->key[0] = temp->key[1];
    temp->key[1] = -1;
    temp->actVals--;
}
else if(siblingFound && !toTheRight)
{
    Node* rightSibling;
    for(int i = 0; i < nodeGap; i++)
    {
        rightSibling = sibling->next;
        rightSibling->key[1] = rightSibling->key[0];
        rightSibling->key[0] = sibling->key[sibling->actVals-1];
        rightSibling->actVals++;
        sibling->key[sibling->actVals-1] = -1;
        sibling->actVals--;
        temp->parent->key[i + indexToNode - nodeGap] =
                sibling->key[sibling->actVals-1];
        sibling = rightSibling;
    }
    temp->key[1] = -1;
    temp->actVals--;
    cout << temp->actVals << endl;
}
```

```
    else
    {
        if(left->key[0] == input)
            left->key[0] = right->key[0];
        for(int i = indexToNode; i < left->parent->actVals-1; i++)
            left->parent->key[i] = left->parent->key[i+1];
        left->parent->key[left->parent->actVals-1] = -1;
        for(int i = indexToNode + 1; i < left->parent->actPtrs-1; i++)
            left->parent->pointer[i] = left->parent->pointer[i+1];
        left->parent->pointer[left->parent->actPtrs-1] = NULL;
        left->parent->actVals--;
        left->parent->actPtrs--;

        left->next = right->next;
        if(left->parent->actVals == 0)
        {
            internalMerge(left->parent);
        }
        delete right;
        return;
    }
}

void internalMerge(Node* node)
{
    Node* sibling;
    bool siblingFound = false;
    bool toTheRight;
    int nodeGap = 0;
    int indexToNode;
    int siblingIndex;
    if(node->parent == NULL)
    {
        root = node->pointer[0];
        delete node;
        root->parent = NULL;
        cout << "Test" << endl;
        return;
    }
    for(int i = 0; i < node->parent->actPtrs; i++)
        if(node->parent->pointer[i] == node)
            indexToNode = i;
    for(int i = indexToNode + 1; i < node->parent->actPtrs; i++)
    {
        sibling = node->parent->pointer[i];
```

```
    if(sibling->actVals > 1)
    {
        siblingFound = true;
        toTheRight = true;
        siblingIndex = i;
        nodeGap = i - indexToNode;
        break;
    }
}
sibling = node->parent->pointer[indexToNode + nodeGap];
if(!siblingFound)
{
    for(int i = indexToNode - 1; i >= 0; i--)
    {
        sibling = node->parent->pointer[i];
        if(sibling->actVals > 1)
        {
            siblingFound = true;
            toTheRight = false;
            nodeGap = indexToNode - i;
            siblingIndex = i;
            break;
        }
    }
    sibling = node->parent->pointer[indexToNode-nodeGap];
}
if(siblingFound && toTheRight)
{
    Node* leftSibling;
    for(int i = 0; i < nodeGap; i++)
    {
        leftSibling = node->parent->pointer[siblingIndex-1];
        leftSibling->key[1] = sibling->parent->key[siblingIndex-1];
        sibling->parent->key[siblingIndex-1] = sibling->key[0];
        leftSibling->pointer[2] = sibling->pointer[0];
        sibling->pointer[0]->parent = leftSibling;
        leftSibling->actVals++;
        leftSibling->actPtrs++;
        for(int j = 0; j < sibling->actVals - 1; j++)
            sibling->key[j] = sibling->key[j+1];
        for(int j = 0; j < sibling->actPtrs - 1; j++)
            sibling->pointer[j] = sibling->pointer[j+1];
        sibling->key[sibling->actVals-1] = -1;
        sibling->actVals--;
        sibling->pointer[sibling->actPtrs-1] = NULL;
```

```
        sibling->actPtrs--;
        sibling = leftSibling;
        siblingIndex--;
    }
    node->key[0] = node->key[1];
    node->key[1] = -1;
    node->pointer[1] = node->pointer[2];
    node->pointer[2] = NULL;
}
else if(siblingFound && !toTheRight)
{
    Node* rightSibling;
    for(int i = 0; i < nodeGap; i++)
    {
        rightSibling = node->parent->pointer[siblingIndex+1];
        rightSibling->key[1] = rightSibling->key[0];
        rightSibling->key[0] = node->parent->key[siblingIndex];
        node->parent->key[siblingIndex] = sibling->key[sibling->actVals-1];
        rightSibling->actVals++;
        rightSibling->pointer[2] = rightSibling->pointer[1];
        rightSibling->pointer[1] = rightSibling->pointer[0];
        rightSibling->pointer[0] = sibling->pointer[sibling->actPtrs-1];
        sibling->pointer[sibling->actPtrs-1]->parent = rightSibling;
        rightSibling->actPtrs++;
        sibling->key[sibling->actVals-1] = -1;
        sibling->actVals--;
        sibling->actPtrs--;
        sibling = rightSibling;
        siblingIndex++;
    }
    node->key[node->actVals-1] = -1;
    node->pointer[2] = NULL;
}
else
{
    Node* left;
    Node* right;
    if(indexToNode == node->parent->actPtrs-1)
    {
        left = node->parent->pointer[indexToNode-1];
        right = node;
        left->key[1] = left->parent->key[indexToNode-1];
        left->pointer[2] = right->pointer[0];
        right->pointer[0]->parent = left;
        left->actVals = 2;
```

```
            left->actPtrs = 3;
            indexToNode--;
        }
        else
        {
            left = node;
            right = node->parent->pointer[indexToNode+1];
            left->key[0] = left->parent->key[indexToNode];
            left->key[1] = right->key[0];
            left->pointer[1] = right->pointer[0];
            right->pointer[0]->parent = left;
            left->pointer[2] = right->pointer[1];
            right->pointer[1]->parent = left;
            left->actVals = 2;
            left->actPtrs = 3;
        }

        for(int i = indexToNode; i < left->parent->actVals-1; i++)
        {
            left->parent->key[i] = left->parent->key[i+1];
            left->parent->pointer[i+1] = left->parent->pointer[i+2];
        }
        left->parent->key[left->parent->actVals-1] = -1;
        left->parent->pointer[left->parent->actPtrs-1] = NULL;
        left->parent->actVals--;
        left->parent->actPtrs--;
        if(left->parent->actVals == 0)
            internalMerge(left->parent);
        delete right;
    }
}

Node* remove(int input){
    Node* node = root;
    int actual;
    int internalIndex = -1;
    int leafIndex = -1;
    int indexToNode;
    Node* nodeWithValue;
    Node* temp = node;
    //Find leaf node
    while(!temp->leaf) {
        for(int i = 0; i < temp->actVals; i++)
            if(input == temp->key[i])
            {
```

```
            nodeWithValue = temp;
            internalIndex = i;
        }
    cout << temp->key[1] << " " << temp->leaf << endl;
    if(input <= temp->key[0])
        indexToNode = 0;
    else if(input <= temp->key[1] || temp->key[1] == -1)
        indexToNode = 1;
    else if(input <= temp->key[2] || temp->key[2] == -1)
        indexToNode = 2;
    else
        indexToNode = 3;
    cout << "Index: " << indexToNode << endl;
    temp = temp->pointer[indexToNode];
}
for(int i = 0; i < temp->actVals; i++)
    if(temp->key[i] == input)
        leafIndex = i;
if(leafIndex == -1)
{
    cout << "Value not in the tree, cannot delete " << input << endl;
    return node;
}
if(temp->actVals > 1){
    for(int i = leafIndex; i < temp->actVals - 1; i++)
        temp->key[i] = temp->key[i+1];
    temp->key[temp->actVals - 1] = -1;
    temp->actVals--;
    if(internalIndex != -1)
        nodeWithValue->key[internalIndex] = temp->key[temp->actVals - 1];
}
else
{
    if(temp->parent == NULL)
    {
        temp->key[0] = -1;
        temp->actVals = 0;
    }
    else if(temp->next != NULL)
        leafMerge(temp, temp->next, temp->key[0], indexToNode);
    else
    {
        leafMerge(temp->parent->pointer[indexToNode-1], temp,
                  temp->key[0], indexToNode);
    }
```

```
        }
        printLeaves();
        Node* leftLeaf = root;
        while(!leftLeaf->leaf)
            leftLeaf = leftLeaf->pointer[0];
        if(leftLeaf->parent == NULL)
            return temp;
        while(leftLeaf != NULL)
        {
            int ptrs = leftLeaf->parent->actPtrs;
            for(int i = 0; i < ptrs; i++)
            {
                leftLeaf->parent->pointer[i] = leftLeaf;
                leftLeaf = leftLeaf->next;
            }
        }
        return temp;
    }

    int getIndexOf(Node* node, int val)
    {
            for(int i = 0; i < node->actVals; i++)
                if(node->key[i] == val)
                    return i;
            return -1;
    }

    int prettyPrint(ofstream& graphfile, Node* node, int count)
    {
        if (node == NULL)
            cout <<"";
        else
        {
           /* if(node->pointer[0] != NULL)
                prettyPrint(graphfile,node->pointer[0], position+1,level+1);
            if(node->pointer[1] != NULL)
                prettyPrint(graphfile,node->pointer[1], position+2,level+1);
            if(node->pointer[2] != NULL)
                prettyPrint(graphfile,node->pointer[2], position+3,level+1);
            if(node->pointer[3] != NULL)
                prettyPrint(graphfile,node->pointer[3], position+4,level+1);
*/
            if(node->pointer[0] == NULL){
                 graphfile << "node" << count << "[label = \" <f0> |  "
                           << node->key[0] << "  |";
```

```
            if(node->key[1] == -1){
                graphfile << "   "<<" * "<<"  |" ;
            }
            else
                graphfile << "   "<<node->key[1]<<"  |" ;
            if(node->key[2] == -1){
                graphfile <<"  "<< " * "<<" 
                           | <f3> &bull;"<<" \" ];\n";
            }
            else
                graphfile <<"  "<< node->key[2]<<" 
                       | <f3> &bull;"<<" \" ];\n";
        }
        else{
            graphfile <<"node"<<count<<"[label = \" <f0> &bull;  |  "
                      << node->key[0]<< "  |";
            if(node->key[1] == -1){
                graphfile << " <f1> &bull; |  "<<" * "<<"  |" ;
            }
            else
                graphfile << " <f1> &bull; |  "
                           <<node->key[1]<<"  |" ;

            if(node->key[2] == -1){
                graphfile <<"<f2> &bull; |  "
                           << " * "<<"  | <f3> &bull;"<<" \" ];\n";
            }
            else
                graphfile <<"<f2> &bull; |  "<<
                          node->key[2]<<"  | <f3> &bull;"<<" \" ];\n";

        }

    }
    for(int i = 0 ;i < node->actPtrs;i++){
            count = prettyPrint(graphfile,node->pointer[i],count+=1);
        }
    return count;
}

int connectGraph(ofstream& graphfile, Node* node, int count){
    int nodesInSubTree=0;
    if (node == NULL)
        cout <<endl;
    if(node->leaf == false){
```

```
        for(int i = 0; i < node->actPtrs; i++){
            if(i > 0 && node->parent == NULL && node->pointer[i]->leaf == false){
                nodesInSubTree+=countSubTree(node->pointer[i],nodesInSubTree);
                graphfile <<"node"<<count<<":f"<<i<<"->node"
                          <<nodesInSubTree+count <<";"<<endl;
            }
            else{
            graphfile <<"node"<<count<<":f"<<i<<"->node"
                      <<nodesInSubTree+count+i+1 <<";"<<endl;
            }
        }
    }
    for(int i = 0 ;i < node->actPtrs;i++){
        count = connectGraph(graphfile,node->pointer[i],count+=1);
    }
    return count;
}
int countSubTree(Node* node, int count){
    if (node == NULL)
        cout <<endl;
    for(int i = 0 ;i < node->actPtrs;i++){
        count = countSubTree(node->pointer[i],count+=1);
    }
    return count;
}

void dotGraphFile(Node* node){
    ofstream graphfile;
    Node *temp = node;
    int numNodes;

    graphfile.open("graph.dot");
    graphfile << "digraph {\n" <<"graph [margin=0,
                            splines=line];\n"<<"edge [penwidth=2];\n"
                <<"node [shape = record, margin=0.03,1.2,
                        penwidth=2, style=filled, fillcolor=white]
                        ;\n"<<endl;
    numNodes = prettyPrint(graphfile,temp,0);

    //Need to figure out how to recursively add all the pointers to sub trees
    connectGraph(graphfile,temp,0);

    graphfile << "}"<<endl;
    graphfile.close();
}
```

```
};

int isNum(string input)
{
    int result = 0;
    int count = 0;
    while(count < input.length())
    {
        if(input[count] < 48 or input[count] > 57 || count >= 3)
            return -1;
        result = result * 10 + input[count] - '0';
        count++;
    }
    return result;
}

int main()
{
    BPlusTree tree;
    bool inserting = true;
    int numNodes = 0;
    string input = "";
    int num;

    cout << "Welcome to the B+ tree creator!" << endl;
    cout << "Please enter value you want to insert.
            To switch to delete, type delete" << endl;
    while(input != "quit")
    {
        cin >> input;
        num = isNum(input);
        if(input == "delete")
        {
            cout << "Now in delete mode: To switch
                    to insert, type insert" << endl;
            inserting = false;
        }
        else if(input == "insert")
        {
            cout << "Now in insert mode: To switch
                    to delete, type delete" << endl;
            inserting = true;
        }
        else if(num < 0)
            cout << "Input must be an integer
```

```
                    of 3 digits or less" << endl;
        else if(inserting)
        {
            tree.insert(num);
        }
        else
        {
            tree.remove(num);
        }
        tree.dotGraphFile(tree.root);
    }
    cout << numNodes << endl;
    return 0;
}
```