COP 5536: Advanced Data Structures, Fall 2017

Programming Project Report – B Plus Trees

Saranya Vatti

29842706 | saranyavatti@ufl.edu

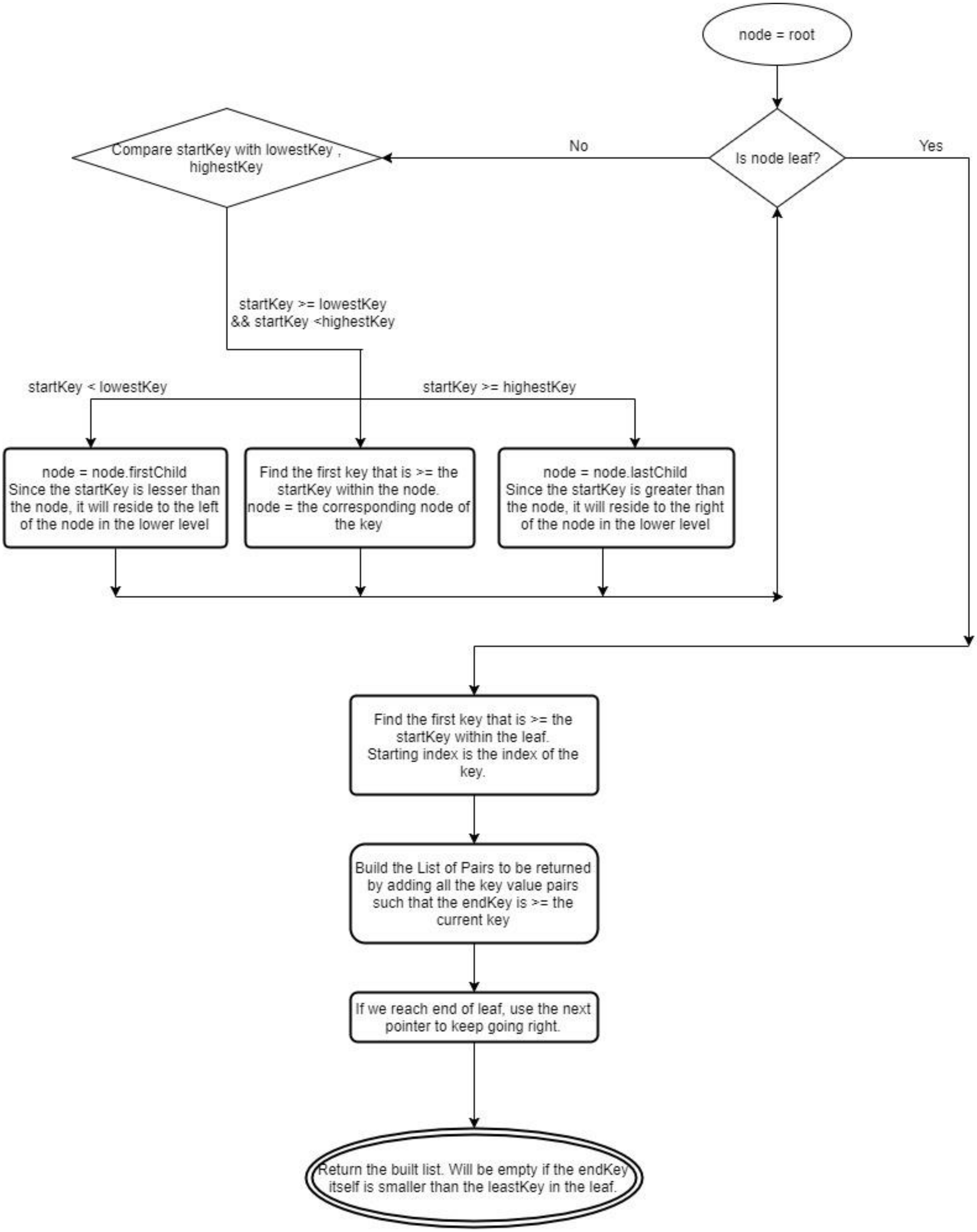
Table of Contents

Running “treesearch”

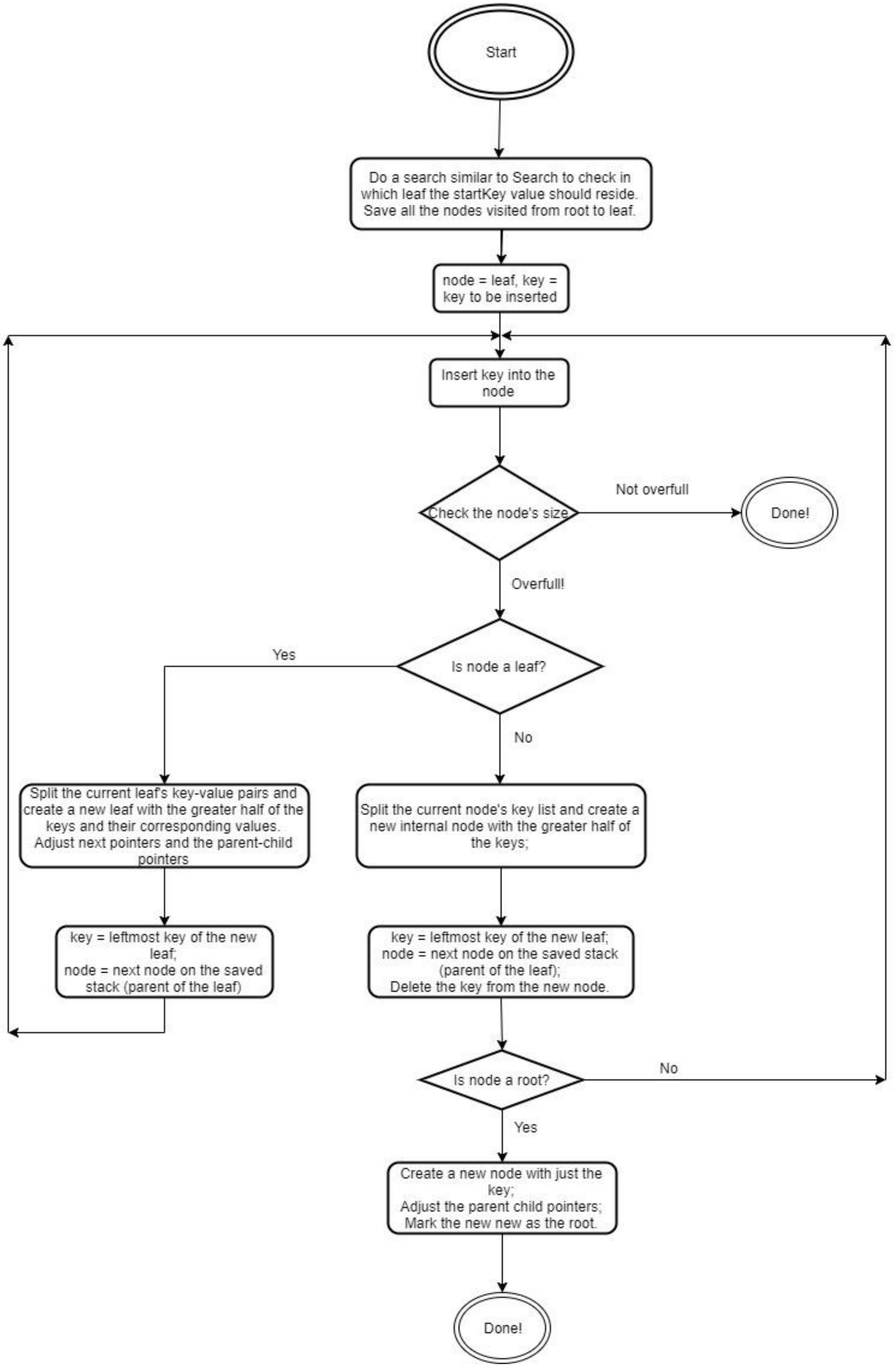
Contents of the zip

Code outline

Search(startKey, endKey)


node = root
Is node leaf?
No
Yes
Compare startKey with lowestKey , highestKey
startKey >= lowestKey && startKey <highestKey
startKey < lowestKey
startKey >= highestKey
node = node.firstChild Since the startKey is lesser than the node, it will reside to the left of the node in the lower level
Find the first key that is >= the startKey within the node. node = the corresponding node of the key
node = node.lastChild Since the startKey is greater than the node, it will reside to the right of the node in the lower level
Find the first key that is >= the startKey within the leaf. Starting index is the index of the key.
Build the List of Pairs to be returned by adding all the key value pairs such that the endKey is >= the current key
If we reach end of leaf, use the next pointer to keep going right.
Return the built list. Will be empty if the endKey itself is smaller than the leastKey in the leaf.

Insert(key, val)


Start
Do a search similar to Search to check in which leaf the startKey value should reside. Save all the nodes visited from root to leaf.
node = leaf, key = key to be inserted
Insert key into the node
Check the node's size
Not overfull
Done!
Overfull!
Is node a leaf?
Yes
No
Split the current leaf's key-value pairs and create a new leaf with the greater half of the keys and their corresponding values. Adjust next pointers and the parent-child pointers
Split the current node's key list and create a new internal node with the greater half of the keys;
key = leftmost key of the new leaf; node = next node on the saved stack (parent of the leaf)
key = leftmost key of the new leaf; node = next node on the saved stack (parent of the leaf); Delete the key from the new node.
Is node a root?
No
Yes
Create a new node with just the key; Adjust the parent child pointers; Mark the new new as the root.
Done!

Code Structure and description

Driver class

class Main

`main()`

Method summary

Driver class to parse the input, create a BPlusTree and run the insert and search methods. Also outputs the result of the search queries to a file. The program always outputs to "output_file.txt" created in the folder that the program runs and overwrites, if anything is present.

Tree class

class BPlusTree

Class description

Class that implements the search and inserts. The BPlusTree is responsible to maintain a root and an order. Root may change after inserts but order never changes.

`BPlusTree(order)`

Method summary

Constructor. Creates and empty BPlusTree with order and root properties.

Parameters

`order {int}`
Integer greater than two. This dictates the maximum number of keys per node.

Returns

The created BPlusTree.

`search(node, startKey, endKey)`

Method summary

Recursively searches for the range between the starting key and the ending key, both inclusive in the BPlusTree. A normal search for a single key will have both the parameters same and the single key search is also implemented with range search with same starting and ending keys. When we reach the leaf, we start at the end of the leaf and look through the keys in the leaf to get the first key where the starting key matches. Once we get that, we keep iterating through the node keys and then the next leaf keys (via leaf pointers) until the ending key satisfies. Time complexity is the height of the tree times the order of the tree. Worst case scenario is when we always have to follow the path that is the second child of the node till the leaf.

Parameters

`node {Node}` Internal node or a leaf where the search will start. Initially, root, then in recursive calls, we follow the path down to the leaf where the starting key is supposed to reside.
`startKey {double}` Starting key where the range search starts.
`endKey {double}` Ending key where the range search ends.

Returns

A list of key, value dictionary pairs that satisfy the range search. A Pair is created for each key and value and then returned. Since the tree inserts keys in sorted order, the returned list is also sorted by key. If there are multiple pairs with same key, they might not be sorted since the values are never compared.

`search(key)`

Method summary

Parent search method called by Main to search for a given key. This simply calls the above search method with starting node as root and the starting and ending keys in the range same as the given key.

Parameters

`key {double}` The single key, all the values of which we should find.

Returns

A list of key, value dictionary pairs that satisfy the key search, which are returned by the above search method.

`search(startKey, endKey)`

Method summary

Parent search method called by Main to search for range between two given keys. This simply calls the above search method with starting node as root and the starting and ending keys as the given starting and ending keys.

Parameters

`startKey {double}` The starting key from which we should find all the matching keys and their corresponding values.
`endKey {double}` The ending key to which we should find all the matching keys and their corresponding values.

Returns

A list of key, value dictionary pairs that satisfy the range search, which are returned by the above search method.

`insert(key, value)`

Method summary

Insert method called by the main method to insert a given key-value pair into the tree.
If the tree is empty, we simply create a new leaf, update the root pointer and end the execution.
If the root is not empty, we recursively trickle down the tree until we find the leaf where the key is supposed to be inserted. Here we handle the case where the leaf becomes overfull and subsequently the internal node becomes overfull too. While we are trickling down the path from root to leaf, we store the nodes in a stack and when we handle an overfull node, we recursively pop the nodes from the stack to handle overfull internal nodes too. If root is overfull, we handle that differently.
When a leaf is overfull, we split the key and value lists both and assign half of them to the new leaf while the rest remain in the old leaf. In addition, we push the smallest key from the new leaf into the parent internal node that we popped out of the stack.
When the internal node becomes overfull, we split the keys into half just like we did with the leaf and then push the smallest key from new internal node to the parent. In addition, we delete the smallest key from the new internal node since we do not need that key to search anymore. We also handle the child lists so that both the internal nodes get the appropriate child lists.
When the root becomes overfull, we split it just like the internal node and push the key into a new root node (which will be an internal node) instead of to a parent internal node.

Since we always maintain order during insertion, all the keys in every node are always in the increasing order.

Parameters

`key {double}` Key that is to be inserted
`value {String}` Value corresponding to the key .

`toString()`

Method summary

A debugger method written to print a tree. Every node of the tree is printed on a single line. Each line is prefixed with the level where the node belongs. If it is an internal node, only the key lists are printed in the array format of java. If it is a leaf node, key and value pairs are printed. This is never called in the current execution.
Example:

```
Level 1-> [7.07]
Level 2-> [3.03, 5.05]
Level 2-> [9.09, 11.011]
Level 3-> 1.01,Value1; 2.02,Value2;
Level 3-> 3.03,Value3; 4.04,Value4;
Level 3-> 5.05,Value5; 6.06,Value6;
Level 3-> 7.07,Value7; 8.08,Value8;
Level 3-> 9.09,Value9; 10.01,Value10;
Level 3-> 11.011,Value11; 12.012,Value12; 13.013,Value13;
```

Node class

```
class Node
```

Class description

A parent class that implements nulls/empty methods for methods that are specific to either InternalNode or Leaf classes. This is a parent class that only implements key lists and methods on key list as both InternalNode and Leaf have key lists.

Properties

`keys {ArrayList<Double>} A list of keys that belong to the current node.`

```
getKeys()
getKey(int index)
setKeys(ArrayList<Double> keys)
getNumOfKeys()
getLeastKey()
getHighestKey()
addKey()
removeKey()
```

Methods summary

All the setter and getter methods for key list. getLeastKey and getHighestKey help in skipping to the next level during a search or an insert when the key to be searched is lesser than the least or greater than the highest key in the current node respectively.

```
getChildNode(int index)
getLastNode()
removeChildNode(int index)
getChildren()
addChildNode (int index, Node node)
```

Methods summary

All the setter and getter methods that are actually implemented in the InternalNode class. Only null/ default values are returned from Node.

```
getValue(int index)
getNext()
setNext(Leaf next)
removeVal(int index)
addVal(int index, String val)
```

Methods summary

All the setter and getter methods that are actually implemented in the Leaf class. Only null/ default values are returned from Node.

```
isLeaf()
```

Method summary

Returns true by default, implemented as false in InternalNode and true in Leaf classes.

InternalNode class

```
class InternalNode
```

Class summary

Extends the Node class and uses the key list. In addition, also maintains a child list which contains the pointers to all the child nodes. The number of child nodes will always be one plus the number of keys in the node. The keys are always stored in the increasing order.

```
getChildNode(int index)
getLastNode()
removeChildNode(int index)
getChildren()
addChildNode (int index, Node node)
```

Methods summary

Getters and setters that are overridden from the parent class and implemented here.

```
InternalNode(ArrayList<Double> keys, ArrayList<Node> children)
```

Method summary

Constructor. References the parent's key list, sets them and also sets the current node's children. Both can be edited by the getter and setter of the parent Node class or the current class.

```
isLeaf()
```

Method summary

Returns false since this is an internal node.

```
toString()
```

Method summary

Debugger method that returns all the keys in the current node, prepended with an "IN" string.

Leaf class

```
class Leaf
```

Class summary

Extends the Node class and uses the key list. In addition, also maintains a value list that has the same size as the key list as these are the key-value pairs that the tree actually contains. These are always in the sorted order – increasing from left to right, sorted by keys. If there are multiple key-value pairs with the same key, the order will be the order in which they are inserted. All the leaves in the tree are on the bottom level and are also linked to each other with a "next" pointer. This is a singly linked list.

```
getValue(int index)
getNext()
setNext(Leaf next)
removeVal(int index)
addVal(int index, String val)
```

Methods summary

Getter and setter methods that are overridden from the parent class and implemented here.

```
Leaf (ArrayList<Double> keys, ArrayList<String> values)
```

Method summary

Constructor. Creates a new leaf with the given key list and the value list.

```
toString()
```

Method summary

Debugger method that returns all the key value pairs in the current Leaf, enclosed in round brackets and separated by commas.

Pair class

```
class Pair
```

Class description

Simple class with key and value. Used to print all the key value pairs that match the range in a tree. Only a toString method is defined and used after a Pair is created.