Advance Data Structure - B+ Tree

Author:

- **Name:** Yi-Ming Chang
- **UFID:** 83816537
- **Email:** yimingchang@ufl.edu

Category

- A. Introduction
- B. Functions ProtoTypes and Descriptions
  - *bPlusTree*
    - 1. Construct Tree and assigning property parameters.
    - 2. Search operation
    - 3. Insert operation
    - 4. Delete operation
  - *treeNode*
    - 6. Construct treeNode
    - 7. Search treeNode interiorly
    - 8. Insert Object into treeNode interiorly
    - 9. Delete Object from treeNode interiorly
- C. Design Details
  - *bPlusTree*
    - *tracePath*
    - *leafList*
  - *treeNode*
    - *keyPairs*
    - *childPointers*
- D. Appendix

A. Introduction

1. *Run this program*

Firstly, we'll going to show the usage on executing the program.
After execute the makefile, we can run the program by the command below.

```
./bplustree [input_file name]
```

The result will be store into ***output_file.txt***.

2. *File structures*

This project builds a B+ tree by using C++. So, the plan of the project is majorly separate into three part

- Maintain B+ Tree property
- Arrangement of the tree node's interior values
- Command execution

Accordingly, we seperate the project into 3 parts

- ***bPlusTree.cpp*** - build to maintain the B+ tree structure.
- ***treeNode.cpp*** - focusing on the interior of a tree node.
- ***main.cpp*** - execute the input command, and output the result.

Moreover, the project files only exist in the first directory, with ***header files showing the parameters and functions of each class***. The file list is showed as below:

```
📦 BplusTree
 ┣ 📜 bPlusTree.cpp
 ┣ 📜 bPlusTree.hpp
 ┣ 📜 treeNode.cpp
 ┣ 📜 treeNode.hpp
 ┣ 📜 main.cpp
 ------
 ┣ 📜 constant.hpp // constant stored
 ┣ 📜 bplustree // exec file
 ┗  📜 makefile
```

B. Functions ProtoTypes and Descriptions

This section show the functions input output parameters and simply describe how the process works to use these functions to build the B+ tree.

bPlusTree

1. Construct Tree and assigning property parameters.

	Function Names	input	return	description
a	bPlusTree	int degree	None	tree constructure
b	init	int degree	None	init tree parameters

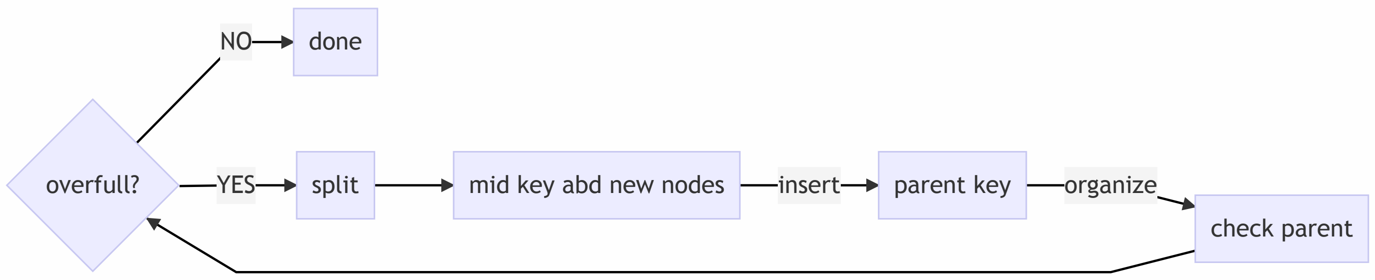
2. Search operation

	Function Names	input	return	description
a	searchLeaf	int key	treeNode*	traverse to the leaf contains the key
b	search	int key	int success	output the value of the key
c	searchRange	int start, int finish	int success	output the values in the range

3. Insert operation

	Function Names	input	return	description
a	insertion	int key, double value	int success	insert data {key, value}

The progress of this function is using *2-(a) searchLeaf* to find the spot to insert the data. After the node insert *8-(b) insertLeafNode)* or *8-(a) insertIndexNode*, there might cause a node ***overfull***, a new sibling nod will be created (See detailed in 8.) which will trigger a propagation insert up to the upper level node. . To simplify the propagation. We shown the flow chart as below.

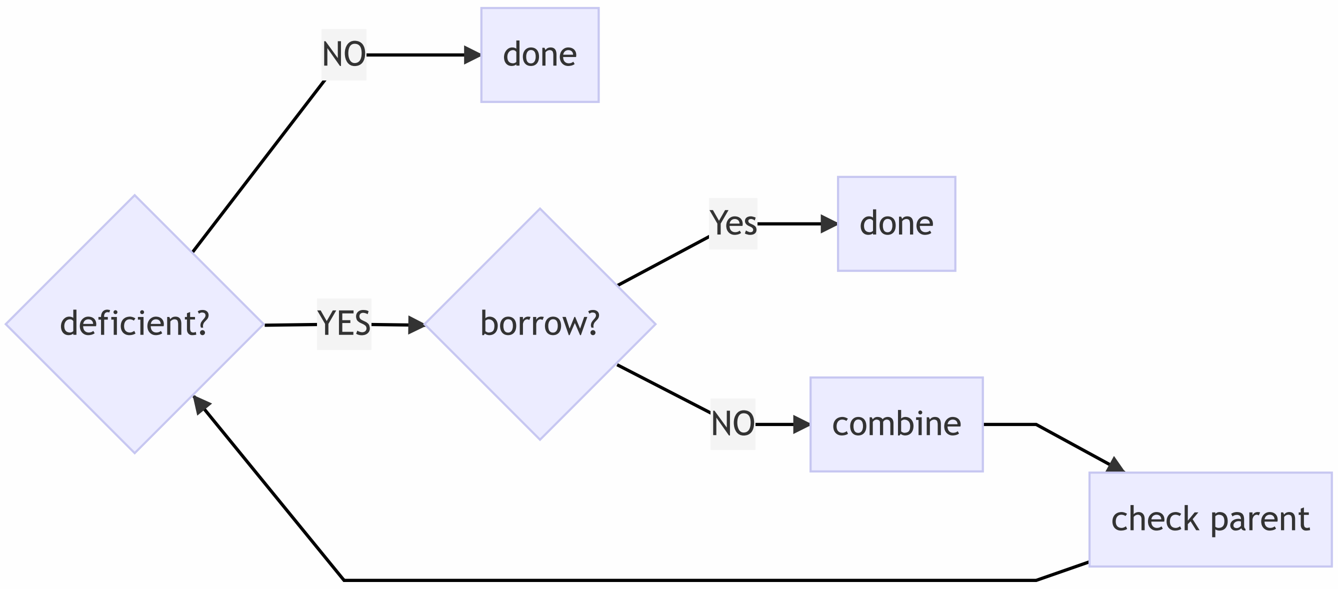


As the flow char shown, the ***middle key*** will be ***insert*** into the parent. However, if the parent node is missing, parent node will have to be created. According to the overfull situation, the ***organize*** contains key and child pointers re-arrangement in the parent node. So, if the split node is INDEX node, we not only have to organize the key but also have to set some of the children pointer correctly.

4. Delete operation

	Function Names	input	return	description
a	deletion	int key, double value	int	delete data {key, value}
b	borrowFromIndex	treeNode* parent, treeNode* deficient	bool	borrow key from sibling index node
c	borrowFromLeaf	treeNode* parent, treeNode* deficient	bool	borrow key from sibling leaf node
d	combineWithIndex	treeNode* parent, treeNode* deficient	bool	combine with index node
e	combineWithLeaf	treeNode* parent, treeNode* deficient	bool	combine with sibling leaf node
f	getInvalidParentKeyIdx	treeNode* parent, iterator changNodeIt	int distance	return the deficient index location of the key and child

Also using *2-(a) searchLeaf* to find the spot to delete the data.
After the interior LEAF node delete *9-(a) deleteLeafNode*, the LEAF node might be ***deficient***, which will trigger the a ***propagation process*** fixing the deficient node. The flow chart shows a brief of the fixing process.



Most importantly, combining with sibling might cause the parent becoming deficient, the process might propogates from leaf to root.

5. Test functions

	Function Names	input	return	description
a	getTreeDegree	None	int degree	return tree degree
b	printLeafList	None	None	print out leaf double link list
c	printTree	None	None	print out the whole tree by DSF
d	getRoot	None	treeNode*	return tree root

treeNode

6. Construct treeNode

	Function Names	input	return	description
a	treeNode	int degree, int key, bool insert	None	construct INDEX node
b	treeNode	int degree, int key, double value, bool insert	None	construct LEAF node

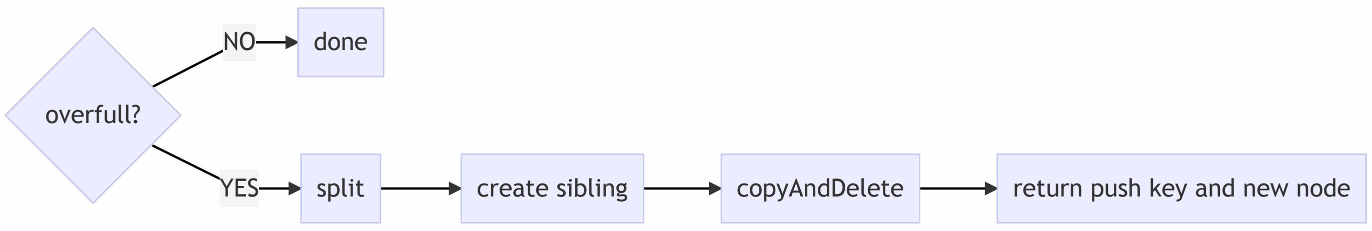
7. Search treeNode interiorly

	Function Names	input	return	description
a	searchIndexNode	int key	treeNode*	return the index node
b	searchLeafNode	int key	pair<bool, double>	return the data

8. Insert Object into treeNode interiorly

	Function Names	input	return	description
a	insertIndexNode	treeNode*, pair<int,double>	pair<int, treeNode*>	insert key into index node
b	insertLeafNode	treeNode, *pair<int,double>, list<treeNode>* &leafList	pair<int, treeNode*>	insert key into leaf node
c	getMiddleKey	None	map<int, double>	return the middle key-value after insert
d	getMiddleChild	None	vector<treeNode*>::iterator	return the middle child pointer after insert
e	copyAndDeleteKeys	treeNode *newNode, iterator start, iterator end	int success	split occurs, copy keys to new node, delete keys from old
f	copyAndDeleteChilds	int key	int success	split occurs, copy childs to new node, delete childs from old

When split occurs, a new sibling node will be create. Then we copy the proper keys and childs to the new node, and remove them from the old node.



9. Delete Object from treeNode interiorly

	Function Names	input	return	description
a	deleteLeafNode	int key	bool isDeficient	delete the key in data

10. Get Classes private variables

	Function Names	input	return	description
a	getIsLeaf	None	bool isLeaf	return tree node is LEAF or INDEX
b	getKeyPairs	None	map<int,double>&	return tree node's key-values
c	getChildPointers	None	vector<treeNode*>&	return tree node's child pointers

11. Test function

	Function Names	input	return	description
a	printNodeKeyValue	None	None	print the node key-values

C. Design Details

This secection I'm going to show the data structures I used in bPlusTree and treeNode classes and why I use it.

bPlusTree

```
class bPlusTree {
  private:
    int degree;
    int minPairsSize;
    treeNode *root;
    vector<treeNode*> tracePath;
    list<treeNode*> leafList;
}
```

- ***degree=m*** - tree is an m-way B+ tree
- ***minPairsSize*** - The minimum number of pairs for all nodes, which is $ceil \lceil m/2 \rceil - 1$
- root*** - point to the root of the tree.
- ### *tracePath*

  Use vector as stack, as we traverse down from root to leaf we push the nodes in the path into stack.
- ### *leafList*

  Use to construct double link list in the bottom of the tree.

treeNode

In this part, the crucial decision is the data structure on using ***map* for *keyPairs*** and ***vector*** for childPointers.

```
class treeNode {
  private:
    /**
     * Maximum keyPairs = degree-1;
     * Minimum keyPairs = ceil(degree/2)-1
    */
    bool isLeaf;
    int degree;
    int maxPairsSize;
    int minPairsSize;
    map<int,double> keyPairs;
    vector<treeNode*> childPointers;
```

- ***degree=m*** - set node degree to m way
- ***maxPairsSize*** - The maximum number of pairs in the map keyPairs
- ***minPairsSize*** - The minimum number of pairs in the map keyPairs

- ### *keyPairs*

  The reason to use map to store the the key-values

  1. To make the insert key order up, the complexity will be only ***$logn$***
  2. With N number insertions, the complexity will be $O(n)$

  Since the map in C++ is build in red-black tree, this result will be better than other std containers insertions, which most of them need to be sort in $O(nlogn)$.

- ### *childPointers*

  Using vector is easily to insert and erase and acces, since we will often have to re-organize the child pointers.

D. Appendix

Database management usage

Many database engine uses B+ tree as there structure, for example InnoDB, Microsoft SQL server, and SQLite. The reason the DBMS often uses B+ tree as the engine structure is because of the properties of the tree.

1. Balance tree with same tree height for all data nodes
2. Leaf node is using double link list, which supports random access as well as sequential access.
3. If we also update the index node value always match the data key, we also can easily to get a subtree of data by fast traverse.

- Refer

  1. InnoDB – Jeremy Cole
  2. B+ tree - Wikipedia