**ECS 36c** **Programming Assignment #2 (50 points, 2 hours)** **Spring 2021**

Due: April 21st at 11:59pm in the p2 handin directory of cs36c.
The only files to be handed in: authors.csv, BTree.h, BTree.cpp, InternalNode.h, InternalNode.cpp, LeafNode.h, LeafNode.cpp.

Your program will read a file that contains a series of integers that it will insert into or delete from a BTree, and then print to the screen a level order traversal of the BTree. This program involves many files: BTreeDriverDebug.cpp, BTree.h, BTree.cpp, BTreeNode.h, BTreeNode.cpp, InternalNode.h, InternalNode.cpp, LeafNode.h, LeafNode.cpp, QueueAr.cpp, QueueAr.h, dsexceptions.h, and Makefile. You are only allowed to modify those files listed above that you will handin. BTreeDriverDebug.cpp contains main(), and prints the tree after each operation. I have already provided the insertion implementation code. Your task will be to provide the implementation code for remove() functions in all of the classes except BTreeNode, which has remove() as a pure virtual function anyway. You may add as many auxiliary functions as you deem necessary. I added six functions other than the original remove()'s already provided.

The program will take three command line parameters: 1) file name; 2) M, the number of children that internal nodes have; and 3) L, the number of integers each leaf node holds. The first number in the file name is the number of insertions, and the second number is the number of deletions. The input file will have the same format as the timetest .txt files. There is one change in the data structure from the book's; the internal nodes should keep track of the minimum value for every child. You will find this easier than not having the minimum for the first child.

I wrote the insertion routine so that when a node splits, the new node created will always contain at least as many entries as the remaining old node. For example, with an L of 4, the old node would hold 2 and the new node would hold 3. Since only integers will be stored in the BTree class, I did not implement it as a template. The output must match that shown in the sample.

For deletion, when the node drops below its minimum size and it has a left sibling then try to borrow from left sibling, and if that fails then merge with the left sibling. If the too small node does not have a left sibling, then try to borrow from its right sibling, and if that fails then merge with the right sibling. If the node has no siblings (and thus the root) and if it is an internal node with only one child it should notify the BTree which should take corrective action. You may assume that both L and M will be greater than two, so there may always be a sibling during the borrowing phase.

You will find all of the BTree implementation files, data files, and my executable in ~ssdavis/36c/p2.

Suggestions: Print out all of the header files, and become familiar with what the non-insertion methods do. You'll be referring to them often. Develop your program iteratively. Get the LeafNode class working, and then start on the more difficult InternalNode class using your LeafNode code as a starting point. Here is a suggested order of development:

1. Add LeafNode code to deal with removal that doesn't involve borrowing. Test with the five-insertion data set and a leaf size of at least 5, i.e., BTree5_1.txt 2 6, BTree5_1a.txt 2 6, BTree5_1b.txt 2 6, BTree5_error.txt 2 6, BTree5_2.txt 2 6, BTree5_3.txt 2 5, BTree5_4.txt 2 5 BTree5_5.txt 2 5.
2. Add LeafNode code to deal with borrowing, and minimal code to InternalNode::remove() to call LeafNode::remove(). When you want to call resetMinimum() of a parent because the minimum value in a node has change, check to make sure the node has a parent first. After all the root has no parent, and thus its parent would be NULL causing a seg fault if you tried to use it to call resetMinimum(). Test with smaller leaf sizes so that will have to borrow, e.g. BTree5_1 2 4, BTree5_1a.txt 2 4.
3. Add LeafNode code to deal with merging, ignoring InternalNode issues, and test accordingly, BTree10_6.txt 4 3
4. Add code to InternalNode::remove to handle when it is a root, and has only one child, and test accordingly, e.g. BTree5_3.txt 2 4, BTree5_5.txt 2 3.
5. Add code to InternalNode::remove to handle borrowing and merging, and test accordingly. Note that when dealing with siblings you will need to cast them to InternalNode*. Pay particular attention to maintaining the sibling and parent pointers. BTree10_6.txt 2 2, BTree18_18.txt 3 2

```
[ssdavis@lect1 p2]$ cat BTree5_1.txt            Leaf: 17 20 29 40
This file inserts 5 values and then
deletes 1
i20 i40 i17 i29 i18 d18
[ssdavis@lect1 p2]$ BTree BTree5_1.txt 2
6
After all insertions.
Leaf: 17 18 20 29 40
Deleting 18.
```

```
[ssdavis@lect1 p2]$ BTree BTree5_5.txt 2
5
After all insertions.
Leaf: 17 18 20 29 40
Deleting 18.
Leaf: 17 20 29 40
Deleting 40.
Leaf: 17 20 29
Deleting 17.
Leaf: 20 29
Deleting 20.
Leaf: 29
Deleting 29.
Leaf:
```

```
[ssdavis@lect1 p2]$ BTree
BTree18_18.txt 3 2
After all insertions.
Internal: 1 7 13
Internal: 1 3 5
Internal: 7 9 11
Internal: 13 15 17
Leaf: 1 2
Leaf: 3 4
Leaf: 5 6
Leaf: 7 8
Leaf: 9 10
Leaf: 11 12
Leaf: 13 14
Leaf: 15 16
Leaf: 17 18
Deleting 1.
Internal: 2 7 13
Internal: 2 3 5
Internal: 7 9 11
Internal: 13 15 17
Leaf: 2
Leaf: 3 4
Leaf: 5 6
Leaf: 7 8
Leaf: 9 10
Leaf: 11 12
Leaf: 13 14
Leaf: 15 16
Leaf: 17 18
Deleting 2.
Internal: 3 7 13
Internal: 3 4 5
Internal: 7 9 11
Internal: 13 15 17
Leaf: 3
Leaf: 4
Leaf: 5 6
Leaf: 7 8
Leaf: 9 10
Leaf: 11 12
Leaf: 13 14
Leaf: 15 16
Leaf: 17 18
Deleting 3.
Internal: 4 7 13
Internal: 4 5
Internal: 7 9 11
Internal: 13 15 17
Leaf: 4
Leaf: 5 6
Leaf: 7 8
Leaf: 9 10
Leaf: 11 12
Leaf: 13 14
Leaf: 15 16
Leaf: 17 18
Deleting 4.
Internal: 5 7 13
Internal: 5 6
Internal: 7 9 11
Internal: 13 15 17
Leaf: 5
Leaf: 6
Leaf: 7 8
Leaf: 9 10
Leaf: 11 12
Leaf: 13 14
Leaf: 15 16
Leaf: 17 18
Deleting 5.
Internal: 6 9 13
Internal: 6 7
Internal: 9 11
Internal: 13 15 17
Leaf: 6
Leaf: 7 8
Leaf: 9 10
Leaf: 11 12
Leaf: 13 14
Leaf: 15 16
Leaf: 17 18
Deleting 6.
Internal: 7 9 13
Internal: 7 8
Internal: 9 11
Internal: 13 15 17
Leaf: 7
Leaf: 8
Leaf: 9 10
Leaf: 11 12
Leaf: 13 14
Leaf: 15 16
Leaf: 17 18
Deleting 7.
Internal: 8 13
Internal: 8 9 11
Internal: 13 15 17
Leaf: 8
Leaf: 9 10
Leaf: 11 12
Leaf: 13 14
Leaf: 15 16
Leaf: 17 18
Deleting 8.
Internal: 9 13
Internal: 9 10 11
Internal: 13 15 17
Leaf: 9
Leaf: 10
Leaf: 11 12
Leaf: 13 14
Leaf: 15 16
Leaf: 17 18
Deleting 9.
Internal: 10 13
Internal: 10 11
Internal: 13 15 17
Leaf: 10
Leaf: 11 12
Leaf: 13 14
Leaf: 15 16
Leaf: 17 18
Deleting 10.
Internal: 11 13
Internal: 11 12
Internal: 13 15 17
Leaf: 11
Leaf: 12
Leaf: 13 14
Leaf: 15 16
Leaf: 17 18
Deleting 11.
Internal: 12 15
Internal: 12 13
Internal: 15 17
Leaf: 12
Leaf: 13 14
Leaf: 15 16
Leaf: 17 18
Deleting 12.
Internal: 13 15
Internal: 13 14
Internal: 15 17
Leaf: 13
Leaf: 14
Leaf: 15 16
Leaf: 17 18
Deleting 13.
Internal: 14 15 17
Leaf: 14
Leaf: 15 16
Leaf: 17 18
Deleting 14.
Internal: 15 16 17
Leaf: 15
Leaf: 16
Leaf: 17 18
Deleting 15.
Internal: 16 17
Leaf: 16
Leaf: 17 18
Deleting 16.
Internal: 17 18
Leaf: 17
Leaf: 18
Deleting 17.
Leaf: 18
Deleting 18.
Leaf:
[ssdavis@lect1 p2]$
```

```
class BTree
{
  BTreeNode *root;
  int internalSize;
  int leafSize;
public:
  BTree(int ISize, int LSize);
  void insert(int value);
  void print();
  void remove(int value);
}; // BTree class

class BTreeNode
{
protected:
  int count;
  int leafSize;
  InternalNode *parent;
  BTreeNode *leftSibling;
  BTreeNode *rightSibling;
public:
  BTreeNode(int LSize, InternalNode *p, BTreeNode *left, BTreeNode *right);
  virtual ~BTreeNode() {}
  int getCount() const;
  BTreeNode* getLeftSibling();
  virtual int getMaximum()const = 0;
  virtual int getMinimum()const = 0;
  BTreeNode* getRightSibling();
  virtual BTreeNode* insert(int value) = 0;
  virtual void print(Queue <BTreeNode*> &queue) = 0;
  virtual BTreeNode* remove(int value) = 0; // NULL == no merge
  void setLeftSibling(BTreeNode *left);
  void setParent(InternalNode *p);
  void setRightSibling(BTreeNode *right);
}; //BTreeNode class

class InternalNode:public BTreeNode
{
  int internalSize;
  BTreeNode **children;
  int *keys;
public:
  InternalNode(int ISize, int LSize, InternalNode *p, BTreeNode *left, BTreeNode *right);
  BTreeNode* addPtr(BTreeNode *ptr, int pos);
  void addToLeft(BTreeNode *last);
  void addToRight(BTreeNode *ptr, BTreeNode *last);
  void addToThis(BTreeNode *ptr, int pos); // pos is where the node should go
  int getMaximum() const;
  int getMinimum() const;
  InternalNode* insert(int value); // returns pointer to new InternalNode if it splits else NULL
  void insert(BTreeNode *oldRoot, BTreeNode *node2); // if root splits use this
  void insert(BTreeNode *newNode); // from a sibling
  void print(Queue <BTreeNode*> &queue);
  BTreeNode* remove(int value);
  void resetMinimum(const BTreeNode* child);
  InternalNode* split(BTreeNode *last);
}; // InternalNode class
```

```
class LeafNode:public BTreeNode
{
  int *values;
public:
  LeafNode(int LSize, InternalNode *p, BTreeNode *left, BTreeNode *right);
  void addToLeft(int value, int last);
  void addToRight(int value, int last);
  void addToThis(int value);
  void addValue(int value, int &last);
  int getMaximum() const;
  int getMinimum() const;
  LeafNode* insert(int value); // returns pointer to new Leaf if splits else NULL
  LeafNode* remove(int value); // NULL == no merge
  void print(Queue <BTreeNode*> &queue);
  LeafNode* split(int value, int last);
}; //LeafNode class
```