Salim Chaouqi
COP3530-Project2
4/13/15

Project 2 Documentation

Code Organization:

Main Idea behind my implementation:

For my implementation I decided to use a 2 class system, were you have a class for the Tree named bTree() and a class for the node named node(). For the data structures I used due simply to my liking of how arrays work, and how easy they are to work with.

Constructor/Deconstructor's

bTree(int sizeOfNodes)

The bTree constructor includes two variables, one which keeps track of the size which in our case is the number of links, and a pointer root that points to the root node.

~bTree()

Simply checks if root has been created, and if so deletes it.

node(int sizeOfNodes, bool leaf)

The node constructor includes 6 variables, 3 of which are Arrays. The first two variables size and leaf are both taken in as parameters. Size works as a tracker of the size of the nodes, and leaf works as a tracker to signify if the node is a leaf or not. The next private variable numElements works as a counter, it is properly incremented/decrement on each insert/delete. The last three private variables are all array data structures. The first data structure keys, is a string array that holds the keys of each node. The second data structure values, is a string array that holds the values of the nodes. The last data structure children, is a data structure that acts as a node pointer to every child connected to this node, note a leaf node will never have any children.

~node()

A simple destructor that deletes all three of the arrays (values, keys, children)

Insert Methods

For insert I have an insert method in both node and bTree, and I also have a method called splitChild.

bTree::insert(string key, string value)

A bool method that first checks if the root is NULL, if the root is NULL, it creates a new node and then calls on insert in the node class on root, after it returns true.

If the root is not NULL, it first checks if the root is full, if it is, it creates a new node that will essentially replace the current root, and make the current root a child of the new node, and then split the new root. After it will insert the key and value into the new child by using the insert method in node. If it did insert properly it will return true, else it will return false. Also if the root node is not full then it will just insert it into the root node, the above operation is only preformed if root is full. When a proper insert is preformed numElements is properly incremented (+1).

Node::insert(string key, string value)

A bool method that first check if the node is a leaf, if it is a leaf it uses a while look to find the location of the new key and value to be inserted, and inserts them in also properly incrementing numElements. If the current node is not a leaf, it will first look for the child the new key and value should be inserted too, if the child is full, then it will first split the child, if the child is not full it will just properly insert the key and value in the proper array and positions, increment numElements, and return true.

node::splitChild(int I, node *x)

First it will create a new node, then it will put half of the key and values of node x into the new node y and properly increment/decrement numElements of y and x. If x is not a leaf then we copy over the last half children of x. Next the method will shift the children of the current node to create space, and link the new node y. Last we move a key of x to the current node and find the location of a new key, and move all the keys and values one space head. At the end of the split operation we copy the middle key/value of x to the current node and increment numElements of the current node.

Find

bTree:find(string key, string *value)

If the root is null then it means that the tree is empty and to return false, otherwise it calls on node to first search the root for they and value. Please note that this is a bool method and it will return true if it finds the element, and false if it does not, also *value at the end of the element if it does find the key, it *value will hold the value associated with that key.

Node::find(string key, string *value)

First it looks for a key that is greater than or equal to the parameter key, if the key is found it will return true and store the value in *value parameter, if not it will first check if this is a leaf, if this is a leaf then it will return false, otherwise it will go to the correct child node of this node and attempt to find it.

toStr() : Preforms inorder traversal of the tree and returns a string with keys in the correct order

bTree::toStr()

A string method that if checks if there is a root, if a root exists then it will the toStr() method in node on the node root, if not it will return an empty string.

node::toStr()

First I create a new variable called returnStr, that will work as the return string, then a for loop is run that first checks if this I a leaf, if it is not a leaf then it goes to the children of the current node (note it will always start at the root), each time adding the key to the returnStr. At the end of the method it will add the last child to the string, and return returnStr which will include every key in tree in inorder traversal.

Delete

For delete I had around 11 helper methods due to the difficulty of debugging which comes from splitting and merging.

bTree:delete_key(string key)

A bool method that first check if there is a root, if a root is not there it returns false. After it preforms the delete and stores the bool value from that delete in rtrBool, if the numElements in root now becomes 0, it first checks if the root is a leaf, if root is a leaf it deletes the root and makes the root node in bTree NULL, otherwise it makes the root one of its children. At the end the method returns rtrBool which if the delete was successful will be true, otherwise it will be false.

node:delete_key(string key)

A bool method that will first it will run a helper method called findKey(key) that will return an int of the closest index or the index of that key. Next it checks if the key to be removed is in the current node, if it is it checks if the current node is a leaf, if the current node is a leaf then it will run the deleteFromLeaf(index) method that is a helper method designed to simply delete that element from the leaf, and check if restructuring is needed. If it is not a leaf it will run the deleteFromNotLeaf(index) helper method that is designed to delete from a nonleaf and restructure that nodes children and the tree if needed. Next it check if this is a leaf node, if this is a leaf node and it did not find the key, then it returns false since the key does not exist. If not we will check if the key is present in the sub-tree rooted with the last child of this node. If the child where the key is found has half the size of elements, then it will run the helper method fill(index) to fill that node. At the end it will return a bool signifying if the delete was successful.

node::merge(int index)

Merge is a helper method that will properly merge two nodes and preform checking using some of the simpler helper methods if restructuring is needed. First two new nodes are created; one is child, and the other sibling. Child is the child node of the current node in the given index, and sibling is the node after child. First it will get the keys and values from the current node and insert them into the index position of the child node; it will then copy the keys and values from sibling to child. It will move the keys and values from the index in the current node one step before to fill the gap created from moving the keys and values to the child. It will next move the child pointers after index + 1 in the current node and one step before, it will finish by properly decrementing/incrementing numElements in the current node and child, and delete the sibling node.

Salim Chaouqi
COP3530-Project2
4/13/15

Testing:

Please note: I created a generator that I did not include to extensively test the code, if you would like to see the generator please send me an email.

Date: **4/10/2015** **Code Version:** **1.1**

Test Case	**Test Description**	**Expected Result**	**Observed Result**	**Pass/Fail**
T01	Insert an Element	Inserts without crashing	Inserted without crashing	Pass
T02	Insert and Element then test toStr() method	Element gets inserted and toStr() properly prints	Element got inserted and toStr() properly prints	Pass
T03	insert multiple elements and toStr() prints properly	Elements that got inserted properly print in inorder traversal	Elements got properly inserted and toStr() prineted things in inorder traversal	Pass
T04	Test split when adding more then size elements then run toStr to check that its inorder	ToStr prints properly signifying that split was run properly	ToStr() printed properly meaning that split ran properly	Pass
T05	Test inserts from smallest to greatest i.e a,b,c,d	Tostr prints properly inorder traversal	ToStr() printed properly signifying that it was proper	Pass
T06	Test inserts from greatest to smallest i.e a,b,c,d	Tostr prints properly inorder traversal	ToStr() printed properly signifying that it was proper	Pass
T07	Test search	Insert and Element then search it and find it	The inserted element was found	Pass
T08	Test search on a missing element	Does not find the element	Did not find the element	Pass
T09	Insert an element, then delete it, then search it	It should not find the element	It did not find the element	Pass
T10	Create a generator to test large inserts	It should handle it and not crash	inserted properly with no crashes	Pass
T11	Test large amounts of inserts and deletes one after another	should insert and delete properly	inserted and deleted properly without crashing	Pass
T12	Test merge by inserting a lot of elements then deleting them	Merge works and arrays stay properly	Merge works properly and array stayed correct tested by calling toStr	Pass
T13	Create a generator to test large amounts of splits/merges to check stability	Should be stable with no crashes and elements should be correct	After many bug fixes splits and merges are stable and works properly	Pass
T14	Use a generator to do multiple tests of insert delete and search	Test up to 10000 elements, generator will report any incorrect searches	All the searches were proper and no bugs were found	Pass

T15	Test for multiple different sizes using the generator	Size should not cause any problems to the logic and output should stay correct	Output is correct with multiple size changes	Pass

Bugs:

No known bugs.