public interface bTree<string>

extends array <string>

This interface extends the array interface to create a tree structure that balances itself when adding or removing elements.

Uses:

Test Cases:

Date: 04/13/15		**Code Version: Project 2**		
Test Case	**Test Description**	**Expected Result**	**Observed Result**	**Pass/Fail**
T01 (TC1.txt)	Tree of size 3. Add elements then search for a key, delete that key, and then search for it again.	OP1.txt	Same as expected	Pass
T02 (TC2.txt)	Add elements to the tree of size 3 then call toStr().	OP2.txt	Same as expected	Pass
T03 (input1)	Adds multiple elements to a tree of size 4. Prints the tree. Searches for a few keys and then prints the tree again.	output1	Same as expected	Pass
T04 (input2)	Adds multiple elements to a tree of size 4. Prints the tree. Deletes a few keys and then prints the tree again.	output2	Same as expected	Pass
T05 (input3)	Creates a tree of size 5, adds elements and prints out tree.	output3	Same as expected	Pass
T06 (input4)	Create a tree of size 3. Add elements, print the tree. Delete one element and print the tree again.	output4	Same as expected	Pass

T07 (input5)	Creates a tree of size 4. Prints tree. Search for a few keys. Delete some keys. Print tree.	output5	Same as expected	Pass
T08 (input6)	Creates a tree of size 5. Adds elements, prints tree, searches for elements that are and are not in the tree. Prints tree again.	output6	Same as expected	Pass
T09 (input7)	Creates a tree of size 3. Adds elements, prints tree. Searches for keys. Deletes elements. Prints tree. Deletes 3 more elements and then prints tree one more time.	output7	Same as expected	Pass
T10 (input8)	Creates a tree of size 4. Adds elements, prints tree. Deletes two elements, and prints tree again.	output8	Same as expected	Pass

Test cases are attached. These files do not have extensions and can be opened in gedit. There are no known bugs.

Modifier and Type	Method and Description
public class bTree	
void	**insert**(string key, string value) Description: This function inserts a key and a value into the bTree. The first part of the method checks to see if the tree is empty and then the second part inserts the values when the tree is not empty. Variables Used: bTreeNode *root, bTreeNode *newNode, bTreeNode *child_ptr, int min_deg, string *keys, string *values, string key, string value, Methods called: insertNode(key, value)
bool	**find**(string key, string *value) Description: Searches for a key in the bTree and then if the key has been found, the value is assigned to the value parameter. Calls the search method which returns the value that coincides with the key value if the key is found. Variables Used: bTreeNode *root, string answer Methods called: search(key)
bool	**delete_key**(string key) Description: Takes in a key and attempts to delete it. If the key is not found the method returns false. If the key is found, it is deleted and then the method returns true. Variables Used: bTreeNode *root, bTreeNode *tmp, bool isLeaf, int num_keys Methods called: remove(key)
string	**toStr**() Description: Concatenates the contents of the tree inorder so that the output is sorted alphabetically as it is in the data structure.

	Variables Used: bTreeNode *root, string output Methods called: traverse()
public class bTreeNode	
void	**traverse**() Description: The traverse method is used to assist the toStr in the class bTree to recursively iterate through the tree. Variables Used: bTreeNode *child_ptr, bTreeNode *keys, bTreeNode *values, bool isLeaf, int num_keys Methods called: traverse() Called by: bTree::toStr()
bTreeNode*	**search**(string key) Description: Very similar to the traverse method, this method iterates through the tree and attempts to check if the key is equal to the key at the current node. Variables Used: bTreeNode *child_ptr, bTreeNode *keys, bTreeNode *values, bool isLeaf, int num_keys Methods called: search(key) Called by: bTree::find(key, *value)
int	**findKey**(string key) Description: Finds where in the array the first key is greater than or equal to the key that you are removing. Variables Used: int index, int num_keys, bTreeNode *keys Methods called: None Called by: remove(key)
void	**insertNode**(string key) Description: This is a helper method to use a current node instead of constantly having to keep track of which new node you are looking at, insertNode recursively calls itself on different nodes until the key and value have found the appropriate empty space to stay. Variables Used: int num_keys, bool isLeaf, bTreeNode *keys, bTreeNode *values, bTreeNode *child_ptr Methods called: split(int, bTreeNode*) , insertNode(key, value) Called by: insertNode(key, value), bTree::insert(key, value)
void	**split**(int index, bTreeNode* fullNode) Description: This method is called when the tree needs to insert a value but the current node is full so it must split what is in the node to create space for another insertion. First the method creates a new node. Then, the method copies the min_deg-1 keys and values into the new node. Once this is complete, the method checks to see if the fullNode was a leaf and if it was not then we know it was a parent so we must copy the children from the full node to the new one. After the children have been copied, the method changes the number of keys left in the full node and copies the rest of the child pointers. Variables Used: bTreeNode *newNode, bTreeNode *fullNode,

	int min_deg, int num_keys, bTreeNode *keys, bTreeNode *values, bool isLeaf, bTreeNode *child_ptr, int index Methods called: none Called by: insertNode(key, value)
void	**remove**(string key) Description: Finds the index of the key and then attempts to remove the key and value by using many helper functions. Variables Used: bTreeNode *newNode, bTreeNode *fullNode, int min_deg, int num_keys, bTreeNode *keys, bTreeNode *values, bool isLeaf, bTreeNode *child_ptr, int index Methods called: findKey(key), getPred(index), getPredVal(index), remove(pred), getSucc(index), getSuccVal(index), merge(index), fill(index) Called by: bTree::delete_key(key)
string	**getPred**(int index) Description: This method is used to re-shuffle the tree after a key is deleted from the node. This is a helper method to keep the tree balanced. Variables Used: bTreeNode *cur, bTreeNode child_ptr, int index, int num_keys Methods called: None Called by: remove(key)
string	**getPredVal**(int index) Description: This method does the same thing that getPred does but it balances the values in the tree instead of the keys as the getPred function does. Variables Used: bTreeNode *cur, bTreeNode child_ptr, int index, int num_keys Methods called: None Called by: remove(key)
string	**getSucc**(int index) Description: This method is similar to the getSucc method as it balances but it balances the left half of the tree after deleteing whereas the getPred balances the rightmost part of the tree. Variables Used: bTreeNode *cur, bTreeNode child_ptr, int index, int num_keys Methods called: None Called by: remove(key)
string	**getSuccVal**(int index) Description: This method balances the values on the left half of the tree. Variables Used: bTreeNode *cur, bTreeNode child_ptr, int index, int num_keys Methods called: None Called by: remove(key)
void	**fill**(int index)

	Description: This method fills in the gap that is created when deleting a node. It is called at the end of the remove method to help with balancing around the given index position. Variables Used: bTreeNode *child_ptr, int num_keys, int min_deg, int index Methods called: borrowPrev(index), borrowNext(index) Called by: remove(key)
void	**borrowPrev**(int index) Description: This function shifts the pointers in the child_ptr array to the right one index position. Then the function checks if the newly created child is a leaf and if it is then it shifts the child pointers in this array as well. Variables Used: bTreeNode *child, bTreeNode *sibling, bTreeNode *keys, bTreeNode *values, bool isLeaf, int num_keys Methods called: None Called by: remove(key)
void	**borrowNext**(int index) Description: This function moves the child_ptrs within the array to the left one index position to account for when a node is deleted from the tree. Variables Used: bTreeNode *child, bTreeNode *sibling, bTreeNode *keys, bTreeNode *values, bool isLeaf, int num_keys Methods called: None Called by: remove(key)
void	**merge**(int index) Description: Combines the child pointers in the array at position index and index + 1 so that they are only taking up the location at the index position. This is used since a node has already been deleted and using the second memory location is no longer necessary. Variables Used: bTreeNode *child, bTreeNode *sibling, bTreeNode *keys, bTreeNode *values, bool isLeaf, int num_keys Methods called: None Called by: remove(key)

Code Organization:

class bTree

private:

```
  bTreeNode *root;
  int size, min_deg;
```

public:

Constructors:
bTree(int s)
~bTree()
Methods
insert(string key, string value) Called when the user would like to add a key and a value to the tree. First this function checks to see if the root is empty. If it is, the function simply inserts the key and value into their respective arrays at the first index position. Since the function is only inserting, there is no need to return a value.
delete_key(string key) This function is called when the user would like to delete a key and a value from the tree and does so by calling many other helper methods to avoid creating one method that is extremely long. By delegating certain tasks to other functions, delete key is able to maintain a simple structure. Once the helper methods have been accessed, delete_key returns a bool value. Returns: The method returns true if the object was found and has been deleted. Otherwise, the function will return false.
find(string key) Searches for the given key within the tree and returns the value stored with a certain key value if the key is found. Returns: The value stored at the same position as the key that was found in the array.
toStr() Concatenates the keys in the tree by traversing the tree in order. Therefore, the keys will be in alphabetical order with all keys starting with a capital letter preceeding keys that begin with a lowercase letter. Returns: String representation of the contents of the tree.

class bTreeNode

private:

string *keys, *values, output;

bTreeNode **child_ptr;

int num_keys, min_deg, size;

bool isLeaf;

public:

Constructors:
bTreeNode(int min, bool leaf)
~bTreeNode()
Methods
insertNode(string key, string value) This is a helper method to use a current node instead of constantly having to keep track of which new node you are looking at, insertNode recursively calls itself on different nodes until the key and value have found the appropriate empty space to stay. This method does not return

anything because it simply inserts the key and value to a node.

remove(string key)
This method is also void and does not return anything, it simply deletes the key and value stored in the tree. The method is very complex however as it requires many helper methods. In order to remove a node in a bTree after removing the key and value the table must rebalance itself.

search(string key)
This method traverses through the tree while the key is greater than the value at index position i of the keys array. Once it exits the while loop the code checks for equivalency of the keys. If the keys are equivalent then the value is stored.
Returns: True if the value was found in the bTree and also assigns the value to be used for later references. If the key is not found, the method returns false

traverse()
Performs an inorder move through the tree so that an output string can be returned to the toStr method.
Returns: The string representation and traversal of the tree to the toStr method.

split(int, bTreeNode*)
This method is called when the tree needs to insert a value but the current node is full so it must split what is in the node to create space for another insertion. First the method creates a new node. Then, the method copies the min_deg-1 keys and values into the new node. Once this is complete, the method checks to see if the fullNode was a leaf and if it was not then we know it was a parent so we must copy the children from the full node to the new one. After the children have been copied, the method changes the number of keys left in the full node and copies the rest of the child pointers.
Returns: Nothing because the new node that was created has been added to the tree so there is no need to return anything.

getPred(int index)
This method is used to re-shuffle the tree after a key is deleted from the node. This is a helper method to keep the tree balanced. The method does this by creating a new node and then setting it equal to the child and the given index position. Then it checks to see if the node there is a leaf and if it is not, the new node becomes the value node that is the a child of the current node. The method recursively calls itself until the it reaches the last leaf
Returns: String which is the value in the keys array and is the rightmost leaf.

getPredVal(int index)
This method does the same thing that getPred does but it balances the values in the tree instead of the keys as the getPred function does. The method traverses through the tree in the same manner but in the end returns the value instead of a key.
Returns: The rightmost leaf’s value that has been stored with the key as a string.

getSucc(int index)
This method is similar to the getSucc method as it balances but it balances the left half of the tree after deleteing whereas the getPred balances the rightmost part of the tree. The method starts towards the center of the tree and then movesto the leftmost leaf to obtain the key value.
Returns: The leftmost leaf’s key value represented as a string.

getSuccVal(int index)
This method balances the values on the left half of the tree. Similarly, it finds the leftmost value that has been stored in the tree, not the key.

Returns: The leftmost leaf's value that has been stored with the key as a string.
fill(int index) This method takes in an index then ensures that the index is not zero so that it is able to go to index-1 of the child pointer array and access a valid node. If the method finds that the node in the child pointer array has the same number of keys as the minimum degree then the tree knows it will need to borrow from this child and position equal to the index value. Otherwise it will attempt to borrow from the node to the right of the index. If the function does not borrow from either of these it will perform a merge. Returns: Does not need to return anything because the method modifies the entire tree when necessary
borrowPrev(int index) This function shifts the pointers in the child_ptr array to the right one index position. Then the function checks if the newly created child is a leaf and if it is then it shifts the child pointers in this array as well. Returns: Does not need to return anything because the method modifies the entire tree when necessary
borrowNext(int index) This function moves the child_ptrs within the array to the left one index position to account for when a node is deleted from the tree. Returns: Does not need to return anything because the method modifies the entire tree when necessary
merge(int index) Combines the child pointers in the array at position index and index + 1 so that they are only taking up the location at the index position. This is used since a node has already been deleted and using the second memory location is no longer necessary. Returns: Does not need to return anything because the method modifies the entire tree when necessary