Washington State University Vancouver

CS 360 Precheck

Binary Search Tree (Using Arrays)

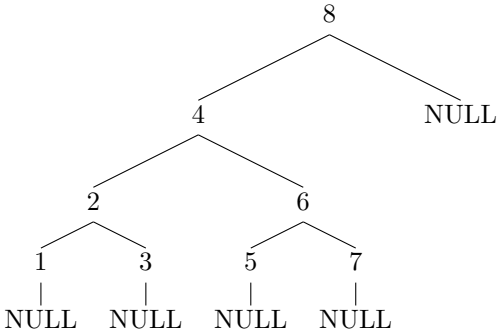
Professor:
Ben McCamish

Overall Assignment

Write a program (in C) targeted at the Linux platform which implements a Binary Search Tree (BST) using only an array. The tree is not balanced and there will be some wasted space. The array used to store the BST must be allocated from the heap. It will be an integer array where a 0 represents an empty node.

Example

Suppose we have the following BST:



This tree would be represented in an array as the following:

```
[8,4,0,2,6,0,0,1,3,5,7,0,0,0,0,]
```

There are a few things to notice about this setup. First, all the empty spots in the tree are represented as 0. We will need this information later on to detect if a node is "empty" or if a node has children. This does mean there will be some wasted space in the tree.

Also note the position of the nodes in the array. If we wanted to find the child of some node, we have to use a specific formula. For instance, the left child of the node containing 4, is a node containing 2. Node 4 is indexed in the array at position 1 and node 2 is indexed in the array at position 3. Thus, our formula for finding the left child would be $leftChildIndex = currentNodeIndex * 2 + 1$. Using our example, if $currentNodeIndex = 1$, then the index of the node containing 2 is $1 * 2 + 1 = 3$. It is a similar story for the right child with the formula being $rightChildIndex = currentNodeIndex * 2 + 2$.

Finally, we can take advantage of integer division to find the parent of a node. If we wanted to find the parent of either node 2 or 6, we first subtract 1 from the index and divide the result by 2. For instance, the node containing 6 is at index 4. Using our formula of $parentIndex = (currentNodeIndex - 1)/2$, we get $(4 - 1)/2 = 1$. Notice that we are using integer division, so the result is truncated from 1.5.

Program Interface

You will implement at least the following interface. Feel free to create additional function as needed, but do not modify the header file provided.

```
typedef struct _bst{
    int size;
    int *tree;
} BST;

BST* createEmptyTree(int size);
void growTree(BST* bst);
int insertValue(BST* bst, int value);
int findValue(BST* bst, int value);
int deleteValue(BST* bst, int value);
```

The Structure

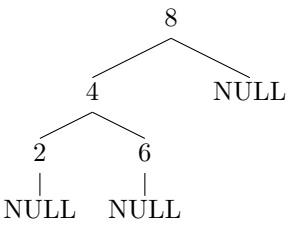
In the structure, there are two values, an integer called `size` and a pointer called `tree`. `size` will let us know how large the current tree is. Another way to look at this is the size of the current array. `tree` will be used to point to the array of integers where we store the values of the BST.

BST* createEmptyTree(int size)

This function will take as input a single integer indicating the size of the new tree. It should return a pointer to the new structure. The tree should be initialized to be empty (all 0s). **Hint:** `calloc` might be useful here.

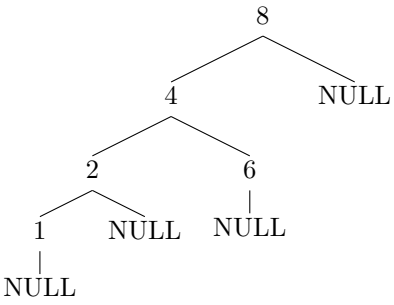
void growTree(BST* bst)

This function will take in an already existing BST structure. It should grow the tree by a single level. For example, suppose we have the following tree and array:



`[8,4,0,2,6,0,0]`

if we wanted to insert the value 1, we would have to grow the tree because there is no room in the array for this value. After growing and inserting the value 1, the tree and array are as follows:



`[8,4,0,2,6,0,0,1,0,0,0,0,0,0,0]`

Notice how the array grew by 8 integers, the size of the next level of the tree. All the new spaces that do not contain the new value are initialized to 0 to indicate they are empty nodes.

int insertValue(BST* bst, int value)

This function will take the parameter `value` and insert this value into the BST `bst` passed as the other parameter. The function should return 1 upon a successful insertion, otherwise return -1. There are a few reasons why the insertion might fail. First, we are not allowing duplicates. If the value already exists in the tree, then the function should return -1. Second, the BST should only accept values > 0, so any negative numbers or 0 should cause the insertion to return -1. Finally, the function may be required to grow the tree to insert the new value. This doesn't cause a return of -1, but should be checked when inserting a new value.

int findValue(BST* bst, int value)

This function will return the index of the value if it exists, otherwise it will return -1. You should use a binary search to reduce the time of searching. Remember the functions discussed earlier on how to navigate the array as if it were a tree. **Hint:** The process for finding a value is almost identical to inserting a new value.

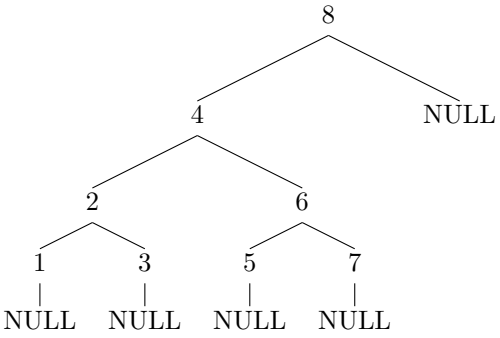
int deleteValue(BST* bst, int value)

This function will take the value and delete it from the tree if it exists. The function returns the index of the deleted value upon a successful deletion, otherwise it will return -1. For reference, we will be using the Hibbard Algorithm for deletion. You can find the full PDF of the original paper in the handout. There are three cases you must consider for deletion:

1. The node to be deleted has no children
2. The node to be deleted has only a single child
3. The node to be deleted has two children

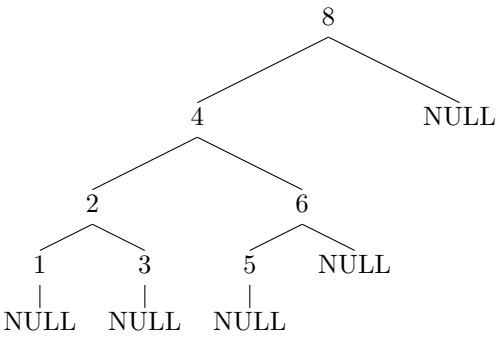
Case 1 The first case is easy. We would simply remove the value from the array. Remember, "deleting" in our case does not reclaim space, but instead we set the value of that spot in the array to 0. See the following example of the tree and array before and after deletion.

Before:



`[8,4,0,2,6,0,0,1,3,5,7,0,0,0,0]`

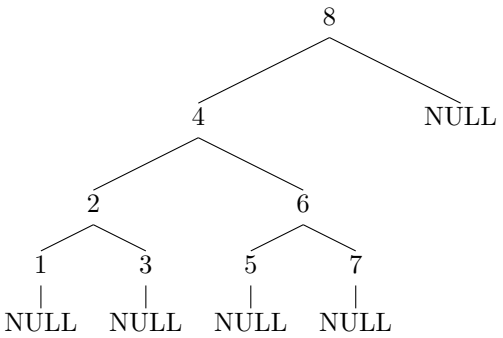
After deleting 7:



`[8,4,0,2,6,0,0,1,3,5,0,0,0,0,0]`

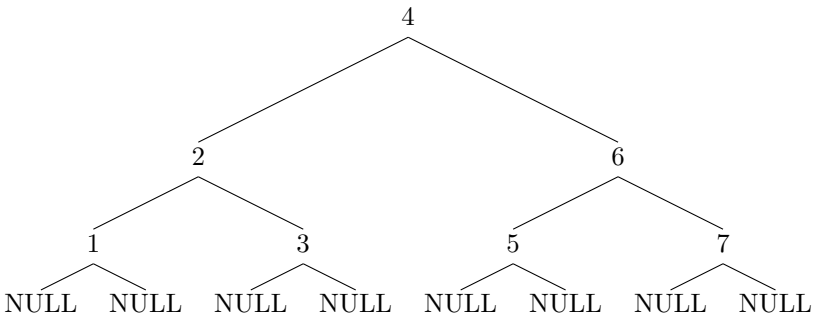
Case 2 The second case is still quite simple. We can remove the value that needs to be deleted, and replace it with the subtree of the only child. This does require moving around some of the other children to maintain the properties of the binary tree. See the following example of the tree and array before and after deletion.

Before:



`[8,4,0,2,6,0,0,1,3,5,7,0,0,0,0]`

After deleting 8:

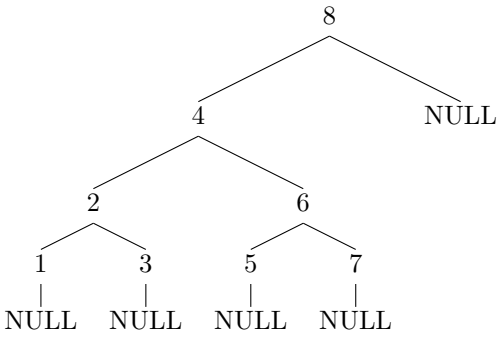


`[4,2,6,1,3,5,7,0,0,0,0,0,0,0,0]`

Notice how there is an entire empty level of the tree in the array. This is fine, we won't be reclaiming space once the tree has grown. You may retain the 0s to indicate those spaces are "empty" and ready to use.

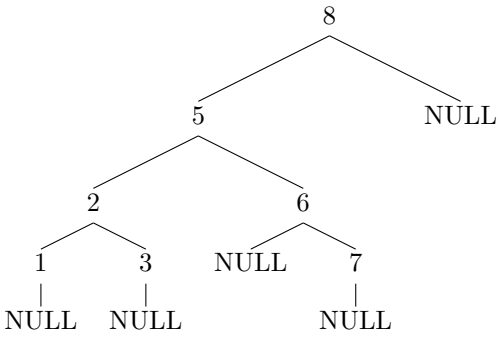
Case 3 This final case is a little bit tricky. First, we need to find what is called the "successor" node. The successor node is the node in the right subtree that has the minimum value. For example, if we wanted to delete 4 in the previous tree, then the successor would be 5. Finding the successor can be done by first going to the right child. Then following the left children until you get to a leaf node. Once a successor has been found, replace the node you want to delete with the value of the successor. Then you can perform either the case 1 or 2 deletion on the successor node. We know that the successor node will have at most one child, since it is the minimum value of this subtree. See the following example of the tree and array before and after deletion.

Before:



`[8,4,0,2,6,0,0,1,3,5,7,0,0,0,0]`

After deleting 4:



`[8,5,0,2,6,0,0,1,3,0,7,0,0,0,0]`

What to turn in (on Autolab):

- A single C file called "bst.c"