For this programming lab, you are to implement the seven Dynamic Set Operators using a Binary Search Tree (BST). The name of your program (i.e., the name of your project) MUST be BST.exe

The nodes for your tree will contain a `string` (the "key"), an `int` (a piece of "Satellite data," to count the number of times the `string` has been seen), and of course, left and right child pointers. Implementing the parent pointer, is *strongly* recommended.

Your program will consist of a loop within `main`, which will accept commands from the keyboard, and execute them one-at-a-time. The commands your program ***must*** support are:

`insert <string>`	If <`string`> is already IN the set, increment the count (stored in the node containing <`string`>) If <`string`> is NOT already in the set, add it to the set (and set the count to 1) In either case, send <`string`> <`count`> to `cout`
`delete <string>`	If <`string`> is NOT in the set, output <`string`> <`-1`>. If <`string`> IS in the set, and has a count of more than 1, decrement the count corresponding to the string, and output <`string`> <`nnn`>, where `nnn` is the new count If <`string`> IS in the set, and has a count of exactly 1, delete it from the set, and output <`string`> <`0`>
`search <string>`	If <`string`> is in the set, output <`string`> <`nnn`>, where <`nnn`> is the count for the string. If <`string`> is not in the set, output <`string`> <`0`>
`min`	Output <`string`>, where <`string`> is the minimum value of the set. If the set is empty, output *only* a blank line. Do not output the count
`max`	Just like `min`, except output the maximum value of the set (or a blank line)
`next <string>`	If <`string`> is in the set, output its successor <`string`>. If <`string`> doesn't have a successor, or if it isn't in the set, output a blank line. Don't output the count.
`prev <string>`	Just like **`next`**, except output the predecessor of <`string`>
`list`	Does a (recursive) in-order traversal, listing all of the strings in the tree in ascending order, along with the number of times each appears (see below)
`parent <string>`	Output the <`string`> contained in the parent of the node whose key is <`string`>. If there is no node containing <`string`>, or it does not have a parent, output *only* a blank line.
`child <string>`	Similar to parent, except it will output the strings contained in the left and right child nodes of the node containing <`string`>. If <`string`> is not in the set, output a blank line; otherwise, output <`string`><comma><space><`string`>, where the two <`string`> values are those contained in the left and right children, respectively. If either child does not exist, output `NULL` as its <`string`>.
`help`	List the commands available to the user
`quit`	Terminate the program

Note: the angle brackets shown above (`<>`) are ***not*** to be used for input or output; they're shown only to indicate that a string `<string>` or a number (`<nnn>`) is to be input or output (see sample input/output below). All of your lines of output should end in a new line (`<< endl`, `'\n'`, or `"\n"`).

Your input *commands* will be case-*in*sensitive, but *not* the strings on which the commands operate (i.e., "`cat`" and "`Cat`" are two different strings, but "`insert  cat`" and "`InSeRt  cat`" would be treated identically, and would do the same thing). The strings will all be single words, so you don't have to worry about parsing entire lines.

You must implement the BST as a class (called BST), using an `#include` file for the interface (in a `.h` file, obviously), and a `.cpp` file for the implementation (see Savitch pp. 476-488, and the linked list demo I provided). If you implement the nodes as their own class (a `struct` will suffice, so you don't have to), then you'll need another `.h` / `.cpp` pair for that, too. Creating a node class is really overkill; I recommend just using a `struct` inside the tree class for the nodes, like I did in the LL Demo.

Your class implements an ADT – it must *not* expose (make public) any of its internal workings. That means that nodes cannot be parameters to any of those public methods, nor can any of those public methods return a node (or pointer to a node). ADTs expose *functionality*; not *implementation details*. Your parameters (and public method return types) must all be strings and/or or integers.

For the `list` command, if the tree is empty, output `"Set is empty"` and a newline; if the tree is not empty, then output `"Set contains: "`, followed by a numbered list separated by commas. There must *not* be a comma after the last item listed, and the list should be numbered from 1 to *n*, where *n* is the number of items in the tree. The item numbers must be in parentheses, but the counts must not. Each key listed is to be followed by its corresponding count. After the last key is displayed, append a newline to the output (see the example below).

Normally, all of your class's methods would return (to `main`, in this case) the values to be output; it's usually considered poor programming practice to have a data structure ADT doing I/O (input/output). For example, `search(string s)` would normally be of type `int`, so that the caller can decide what to DO with that `int`. For this project, I'm going to relax this particular "best practice" – it is fine if you have `search` as a `void` method, and then have `search` actually output `<string> <nnnn>` itself.

If you go with this approach (which I recommend), then `main` becomes basically a command-and-parameter-accepting loop that dispatches the appropriate call. This might be your basic structure:

```
int main()
{
    instantiate a BST object
    loop forever
    {
        Get a word (command string) from the console
        If it's one of our one-word commands, execute it
        else, if it's a command that takes an argument, then
           Get a word (the parameter string) from the console
           Execute the command with the parameter
        else, it's an invalid command
    }
}
```

Some of you may have had occasion to code in C (or in "C-Style" C++) somewhere along the way. You should NOT use the old-style C-Strings (`char` arrays); rather, use the new standard `string` class (see chapter 9, Savitch). The new class functions much more like Java's `String` class, and in some ways is even more flexible (for example, you can use the == operator to compare the *contents* of two strings, rather than their *references*). You can also use >, <, >= and <= to compare strings on a character-by-character basis. Comparisons are made using the ASCII values of the characters in the strings (see http://www.ASCIIChart.com).

You already have snippets of code (in the lecture slides) for some of the operations you will need, and you have pseudocode for all of the set operators. You may need to code some other methods, but the main ones should be rather straightforward. Note: In the lecture slides, because they came straight from the Cormen text, the set, *S*, is a parameter to most of the dynamic set operators. In this case, all of your methods will be inside the BST class, so they will know what *S* is – that means you don't need to worry about passing *S*. Cormen's operator definitions also referred to "items" using *x* (which, in the lecture I said would be a node pointer), but you will be using *k*, the key instead.

All code you write for this assignment must be yours and yours alone (except for what is in the lecture slides and the Savitch text, of course). You may *not* use any code from any other source, including the Internet, even as a reference. Using code other than what you write (or are explicitly permitted to use) will be considered academic dishonesty, and will be dealt with in most severe terms. You may not post any of this assignment on the internet, seeking help.

Sample input and output:

Input	Output
`search cat`	`cat 0`
`min`	`<blank line>`
`max`	`<blank line>`
`next cat`	`<blank line>`
`insert cat`	`cat 1`
`insert cat`	`cat 2`
`search cat`	`cat 2`
`delete cat`	`cat 1`
`insert dog`	`dog 1`
`min`	`cat`
`max`	`dog`
`list`	`Set contains: (1) cat 1, (2) dog 1`
`child cat`	`NULL, dog`
`next cat`	`dog`
`prev dog`	`cat`
`parent dog`	`cat`
`parent cat`	`<blank line>`
`quit`	`<program ends>`

So, on the screen, the interleaved typed input and displayed output would look like this:

```
search cat
cat 0
min

max

next cat

insert cat
cat 1
Insert cat
cat 2
Search cat
cat 2
delete cat
cat 1
insert dog
dog 1
min
cat
max
dog
list
Set contains: (1) cat 1, (2) dog 1
child cat
NULL, dog
next cat
dog
prev dog
cat
parent dog
cat
parent cat

quit
```

the program would end at this point, with no other output

Hard requirements for your program:

1. It *must* be named BST (i.e., your compiled program must be called BST.exe)
2. You *must* have a class called BST (declared in a `.h` file, and implemented a `.cpp` file), and another `.cpp` file to hold `main()`.
3. The methods your BST class *must* expose are (in addition to a constructor and a destructor):

   ```
   insert(string word)          parent(string word)
   remove(string word)          child(string word)
   search(string word)          min()
   next(string word)            max()
   prev(string word)            list()
   ```

   Note: Although the *command* is "delete", the *method*'s name will be "remove" (`delete` is a C++ reserved word)

   You may create a method for `help` if you like, or you can implement that in `main()`.

   Quitting the program should be done from within `main` (as soon as you realize you have the "quit" command, just `delete` your tree and `return 0` from `main`!

   Note that none of these methods accepts any tree-specific parameters (no node pointers); nor can any of your public methods *return* a node pointer (just `int`s, `string`s, or they can be `void`).

   Your destructor must delete every node in the tree (recall the discussion about using a pointer after what it points *at* has been deleted – it *may* happen to work, but it's a no-no). You must find a way of deleting the nodes in a *single* traversal; the destructor should make only *one* pass through the tree. You may not make repeated passes through the tree, deleting the leaves each time (that's too inefficient).

   None of your code may be recursive except the list traversal and a traversal related to the destructor.

4. You are categorically *not* allowed to use anything from the C++ STL (Standard Template Library). This includes, but is not limited to, `vector`, `bitmap`, `map`, `list`, `set`, etc. If there's something you want to use beyond standard I/O or strings, check with me first for permission. Even if not explicitly stated in future assignments, it applies to all of your programming labs this semester. The idea is for you to learn how those data structures work *internally*; not for you to just pick up the pre-made structures and apply them.
5. Your program must be able to do each of the following (each should work correctly, regardless of how much [or how little] data is already in the BST, and how the nodes in the tree are arranged):
   - Adding a new item
   - Listing the contents of the tree
   - Finding the minimum and maximum values
   - Finding the predecessor or successor of any value (whether that value is in the tree or not)
   - Deleting any value from the tree with a count > 1 (which should leave the tree unchanged, except for decrementing the count in the appropriate node)
   - Deleting any value from the tree with a count of 1 (which should cause (only) the appropriate node

to be removed, without disconnecting any other branches in the tree, or creating any pointer problems)

6. Note the rules on case-sensitivity (or case-insensitivity, as the case may be) – see above.
7. ***<u>You must not prompt the user for any input, nor label any output beyond what is specified</u>*** (no "enter command", or "min is" messages.
8. You may not use Cormen's TRANSPLANT helper function; implement `delete` from the explanation in the lecture slides.
9. If your source code won't compile, it will get a zero.
10. Your project must be built in Visual Studio 2019 (not 2017 or an earlier version). If you are using PCs on the main campus, and VS 2017 is all that's installed, please contact me before you submit your project.
11. Your project must use the Visual Studio 2019 Platform Toolset v142 (right-click on your project in the "Solution Explorer" window to check / change it.

Suggestions:

1. It may be helpful for you to create some (private) utility functions to help you with coding and debugging of your BST class. Feel free to use the following:

```
isLeaf(node* p) {return p->lch == nullptr && p->rch == nullptr;}

isRoot(node* p) {return p == root;} // assumes p points at a valid node

getChildCount(node* p)
{
    if (p == nullptr) return -1; // can't count children of no node!
    if (isLeaf(p)) return 0;     // leaves have no child nodes
    if (p->lch != nullptr && p->rch != nullptr) return 2; // 2 children
    if (p->lch != nullptr && p->rch == nullptr) return 1; // 1 (l child)
    if (p->lch == nullptr && p->rch != nullptr) return 1; // 1 (r child)
    // it should not be possible to get here - all nodes have 0, 1, or 2 children
    cout << "Internal error in getChildCount\n"; // practice defensive
    exit(1);                                     // programming!
}

// The next two functions presuppose p is not null, and doesn't point at the root
isLeftChild(node* p)  {return p == p->parent->lch);}

isRightChild(node* p) {return p == p->parent->rch);}
```

Your program should utilize good programming practices – information hiding, etc. Your public methods should not give away HOW the BST works. For example, `main()` should never see your BST's root pointer or know about nodes. You will have to create some public methods that, in turn, call private methods to get the job done. Your "dynamic set" could just as easily be implemented with arrays, a linked list, or something else. What gives it its behavior is its *interface*; *<u>not its implementation</u>*!

Your code must be well-documented:

- Use block comments at the start of each method, briefly explaining *what* it does, and *how* it does what it does.
- Use line comments liberally to explain what's going on
- Use internal documentation, like descriptive variable names where appropriate
- Make SURE you have a suitable block header on ALL of your source files WITH YOUR NAME, THE COURSE, AND THE DATE!

Some helpful starting points:

- Create your program as a Win32 Console application within VS 2019
- Your program should use the `std` namespace; NOT the `System` namespace
- To get access to `cin` and `cout`, use "`#include <iostream>`"
- To get access to the `string` data type and its related functionality, "`#include <string>`"

Consequently, you should create a number of test scenarios that test various configurations of the tree, particularly with respect to delete – as we have seen / will see, delete has multiple sub-cases, so make sure your code handles them all by creating tests that exemplify them all! If your program crashes in the middle of a test, then it fails the test; I won't debug your code, or try to figure out how to get it to work after you turn it in; that's your job *before* you turn it in.

Your program is NOT to prompt the user for any input, nor should it produce any output other than what is called for above. It should start and just wait for the first command, and when the command is "quit", it should just end. There must NOT be a "press any key to exit" or other "to end the program, …" prompt.

To submit your program to Blackboard, use 7-Zip to create a compressed archive of your entire Visual Studio workspace folder (the one that contains the `.sln` file, and the other folders within the solution folder – do not just submit your source and/or `.exe` files). You MUST use 7-zip (not zip, rar, or any other compression format). Name your project "<LastName><comma><space><FirstName>.7z", as in "`Smith, Bob.7z`". If you don't have 7-zip, you can get it free at www.7-zip.org.

You are expressly forbidden from using anything from the C++ STL (Standard Template Library). Components like `map`, `treemap`, `vector`, `bitset`, etc., are ALL off-limits for the entire semester, even if I happen to not say so in subsequent assignments. All you should need for this one are standard I/O and `string`s. The nodes for your tree will be `struct`s that you create.

None of your code should be recursive, except for the traversals – use loops to navigate the tree for all non-traversal operations. For the `delete` operation, code the deletion manually, using the procedure in the slides. Make sure you test for ALL cases!

This is not the time to try to find obscure or esoteric features of the language to use (no need for inheritance, overloading, pointers to pointers, etc.) – this should be very straightforward code. The most complex language feature you should need is most likely the `?:` operator, and even that one is not really required (though it can make your code a bit shorter, and once you're used to it, easier to read).

All of your code must be yours and yours alone. You may not copy code or non-code positions of the solution from each other, or from any online source. You may not post on any online forum seeking help. Write all of this from scratch. See the syllabus for more details.