Assignment 3: Binary Search Tree
Due at 11:59 pm on Sunday, 2/25/2024
Demo due by 11:59 pm on Friday 3/8/2024

This assignment is intended to have you start to explore non-linear data structures by implementing a binary search tree (BST). After implementing the BST, you'll solve some BST-based puzzle problems. The requirements for the assignment are described below.

Part 0. Download the skeleton code and unzip

You may download the skeleton code for this assignment using wget command:
wget https://classes.engr.oregonstate.edu/eecs/winter2024/cs261-020/assignments/assignment3.zip

To unzip the file, use the following command:
`unzip assignment3.zip`

Part 1. Implement a binary search tree

Your main task for this assignment is to implement a binary search tree(BST). A BST is a tree-structured data type that allows fast insertions, lookups, and removals by structuring itself in a way that encodes the behavior of binary search. Specifically, each node in a BST has at most two children, a left child and a right child, and every node satisfies the BST property, which requires that all values stored in a node's left subtree must be less than that node's value, while all values stored in a node's right subtree must be greater than or equal to that node's value.

For this assignment, the interface for the BST you'll implement (i.e. the structures and the prototypes of functions a user of the BST interacts with) is already defined for you in the file `bst.h`. Your first task in this assignment is to implement definitions for the functions that comprise this interface in `bst.c`.

Note that you may not modify the interface definition with which you are provided. Specifically, do not modify any of the already-defined BST function prototypes. We will use a set of tests to verify your implementation, and if you change the BST interface, it will break these tests, thereby (negatively) affecting your grade. You may also not modify any of the existing structures defined in the starter code (i.e. `struct bst` and `struct bst_node`). Beyond these things, though, feel free to add any additional functions or structures your BST implementation needs.
(Hint: you might want to add helper functions that takes `struct bst_node` as parameter, so you can apply recursion!)

The BST functions you'll need to implement are outlined briefly below. All of these functions use a type called `struct bst`, which is defined in `bst.c` and represents the BST itself. For more details, including information on function parameters and expected return values, see the documentation provided in `bst.c`.

- `bst_create()` – This function should allocate, initialize, and return a pointer to a new BST structure.

- `bst_free()` – This function should free the memory held within a BST structure created by `bst_create()`. Note that this function only needs to free the memory held by the BST itself. It does not need to free the individual elements stored in the BST. This is the responsibility of the calling function.

- `bst_size()` – This function should return the total number of elements stored in a given BST. Importantly, because you can't modify the fields of `struct bst` or `struct bst_node`, you'll have to calculate a BST's size each time this function is called. **It could be helpful to think recursively here**. Feel free to write any helper functions you need to make this work.

- `bst_insert()` – This function should insert a new key/value pair into a given BST. The BST should be ordered based on the specified key value. In other words, your BST must always maintain the BST property among all keys stored in the tree.

- `bst_remove()` – This function should remove the value with a specified key from a given BST. If multiple values are stored in the tree with the same key, the first one encountered (i.e. the one closest to the root of the tree) should be removed.

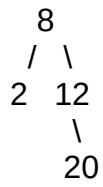
- `bst_get()` – This function should return the value associated with a specified key in a given BST. If multiple values are stored in the tree with the same key, the first one encountered (i.e. the one closest to the root of the tree) should be returned.

Part 2. Solve some BST puzzles

Now that you have the basic BST itself implemented, it's time so solve a few job interview-type BST "puzzles". Each of the puzzles you'll solve is listed below. For each one, you'll implement a function that's already stubbed out for you in `bst.c`. **Note that some of these are hard puzzles!** If you can't immediately figure out a solution to all of them, don't worry. I strongly encourage you to discuss how to solve these problems with your fellow classmates, especially on Discord.

1. ***Compute the height of a BST***. The first puzzle problem is to implement the function `bst_height()` to compute the height of a given BST. Remember, the height of a tree is the maximum depth of any node in the tree (i.e. the number of edges in the path from the root to that node). **Like with `bst_size()` above, it could be helpful to think recursively to solve this problem**. Feel free to write any helper functions you need to make this work.

2. ***Check if a path sum is valid in a BST***. The next puzzle problem involves path sums. A path sum is the sum of all the keys in a path from the BST root to one of the BST leaves. For example, the following BST (with only its keys visualized) has two path sums, 10 and 40:



   For this problem, you'll implement the function `bst_path_sum()` to determine whether a given value is a valid path sum within a given BST. In other words, you should check whether the BST contains any path from the root to a leaf where the keys sum to the specified value. **Again, it could be helpful to think recursively here**, and you can again feel free to write any helper functions you need.

3. ***Compute a range sum in a BST***. The last puzzle problem involves computing the sum of all the keys in a BST within a given range. Specifically, you should implement the function `bst_range_sum()` to compute the sum of all keys in a BST between a given lower bound and a given upper bound. For example, in the BST above, the sum for the range [2, 15] is 22 (i.e. 2 + 8 + 12), and the sum for the range [5, 10] is 8 (since 8 is the only key within that range). **As with the problems above, it could be helpful to think recursively here.** Feel free again to implement any helper functions you need here.

   Note that for full credit on this problem, you should not explore/process any subtree whose keys cannot be included in the range sum. For example, when computing the sum for the range [5, 10] in the tree above, the subtree rooted at 12 should not be explored/processed, since none of the keys in that subtree fall within the specified range.

Extra credit: implement an in-order BST iterator

For up to 10 points of extra credit, you can implement an iterator for your BST that returns keys/values from the BST in the same order they would be visited during an in-order traversal of the tree. In particular, for a BST, this means the iterator should return keys in ascending sorted order.

The type you'll use to implement the iterator, struct `bst_iterator`, is declared in `bst.h` and defined in `bst.c`. You may not change the definition of this structure by adding or modifying its fields. The one field it contains represents a stack, which means that you'll have to use a stack to help you order and store the BST's nodes during the in-order iteration (the stack implementation is provided in `stack.{h,c}` and `list.{h,c}`).

You'll also have to implement the following functions, which are defined in `bst.c` (with further documentation in that file):

- `bst_iterator_create()` – This function should allocate, initialize, and return a pointer to a new BST iterator. The function will be passed a specific BST over which to perform the iteration.

- `bst_iterator_free()` – This function should free the memory associated with a BST iterator created by `bst_create()`. It should not free any memory associated with the BST itself. That is the responsibility of the caller.

- `bst_iterator_has_next()` – This function should return a 0/1 value that indicates whether or not the iterator has nodes left to visit.

- `bst_iterator_next()` – This function should return both the key and value associated with the current node pointed to by the iterator and then advance the iterator to point to the next node in an in-order traversal. Note that the value associated with the current node must be returned via an argument to this function. See the documentation in `bst.c` for more about this.

To be able to earn this extra credit, your BST insertion function must be working correctly. **Importantly, to earn the full 10 points of extra credit, your iterator must have worst-case space complexity of O(h), where h is the height of the BST**. In other words, to earn full credit, you can't just implement a normal in-order traversal of the BST that stores all of the tree's nodes in the correct order in the iterator's stack. You'll have to be more clever.

Hint: To implement this iterator so that it has O(h) space complexity, try to mimic the way a recursive in-order traversal works. In particular, think about the way that each node in a BST "waits" to be visited/processed while its entire left subtree is explored. Then after that node is visited/processed, its entire right subtree is explored. Can you use the stack with which you're provided to implement this "waiting" behavior? In particular, can you put BST nodes onto the stack in such a way that the top of the stack always represents the next node to be visited/processed in an in-order traversal? To see how this might work, think about the way function calls are added and removed to/from the call stack during a recursive in-order BST traversal.

To test your iterator implementation, a testing application `test_bst_iterator.c` is provided. This application will be compiled automatically when you run `make`, and you can run it like so: `./test_bst_iterator`. The expected output for this application is provided in the `example_output/` directory.

Additional Information: Storing key/value pairs

It is important to note that each data element stored in your BST will actually consist of two parts: a key and a value. Under this scheme, the key will serve as a unique identifier for the data element, while the value will represent the rest of the data associated with that element. For example, if you were storing information about OSU students in your BST, the key for each student element might be that student's OSU ID number, while the corresponding value might be a struct containing all other data related to that student (e.g. name, email address, GPA, etc.). Storing data as key/value pairs is a common approach that we'll see in other data structures we explore in this course.

For your BST implementation, each data element's key will be represented as an integer value, while the associated value will be a void pointer. This is reflected in the structure you must use to represent a single node in your BST:

```
struct bst_node {
    int key;
    void* value;
    struct bst_node* left;
    struct bst_node* right;
};
```

Your BST should be organized *based on the keys* of the elements. In other words, the BST property must always hold among all *keys* in the tree. For example, when a new data element is inserted into your BST, decisions about whether to insert that element within a node's left subtree or its right subtree should be based on comparisons between the key of the element being inserted and the keys stored in the tree. Similarly, when a user wants to lookup or remove data elements stored in your BST, they will do so by specifying the key to be found/removed.

Testing your work

In addition to the skeleton code provided here, you are also provided with some application code in `test_bst.c` to help verify that your BST implementation, is behaving the way you want it to. In particular, the testing code calls the functions from `bst.c`, passing them appropriate arguments, and then prints the results. You can use the provided `Makefile` to compile all of the code in the project together, and then you can run the testing code as follows:

```
make
./test_bst
```

Example output of the testing program using a correct BST implementation is provided in the `example_output/` directory.

In order to verify that your memory freeing functions work correctly, it will be helpful to run the testing application through `valgrind`.

Submission

In order to submit your homework assignment, you must create a **zip file** that contains `assignment3/` folder with your implementation. This zip file will be submitted to TEACH . In order to create the zip file, go to the directory where you can access the `assignment3/`, and use the following command:

```
zip assignment3.zip assignment3 -r
```

Remember to sign up with a TA to demo your assignment. The deadline of demoing this assignment without penalties is 3/8/2024.