ENEE 150: Intermediate Programming Concepts for Engineers

Spring 2019 **Handout #32**

Project #4: Binary Search Tree ADT: Due May 13 at 11:59pm

In this project, you will build a binary search tree (BST) abstract data type. Your ADT will permit users to perform insert and search operations, and allow users to examine data in sorted order. Your BST ADT will also be polymorphic, permitting the type of data to be defined by the user, and permitting the user to control sort order by a user-provided function accessed from the ADT via function pointers. In addition to building the BST ADT, you will also rewrite the tree-search.c and parse.c examples from lecture to use your new ADT. tree-search.c will use the ADT to store integers while parse.c will use the ADT to store dictionary entries, thus demonstrating the polymorphic nature of the ADT.

1 Binary Search Tree

As described in lecture, a BST is a binary tree that maintains entries in sorted order, allowing users to perform insert and search operations efficiently. At any given node in a BST, the left sub-tree contains nodes that are "less than" the given node while the right sub-tree contains nodes that are "greater than" the given node. This property holds for all nodes in the BST (*i.e.*, all sub-trees in the BST are themselves BSTs).

In this project, you will associate a "key" with each BST node which uniquely identifies the data stored at the node. This key will be used to order nodes in the BST during insert operations, and to search for data stored in the nodes. For each insert and search operation, you will perform a tree traversal. This is done starting at the root of the BST, with traversal proceeding downwards either left or right at each node depending on whether the key being inserted or searched is less than or greater than the key associated with the currently traversed node. Your BST will only hold unique keys. Attempts to insert data with a key already in the BST will not succeed (the BST will not be changed in this case).

Your ADT will also keep track of the number of items inserted into the BST as well as the depth of the BST. At any time, the user can call a function to retrieve these statistics from the ADT.

1.1 Polymorphism

Your BST ADT will be *polymorphic*, allowing users to define their own data stored in the BST, as well as the keys associated with that data. Your ADT will not place any constraints on the size or composition of the user's keys or data: they can be single values

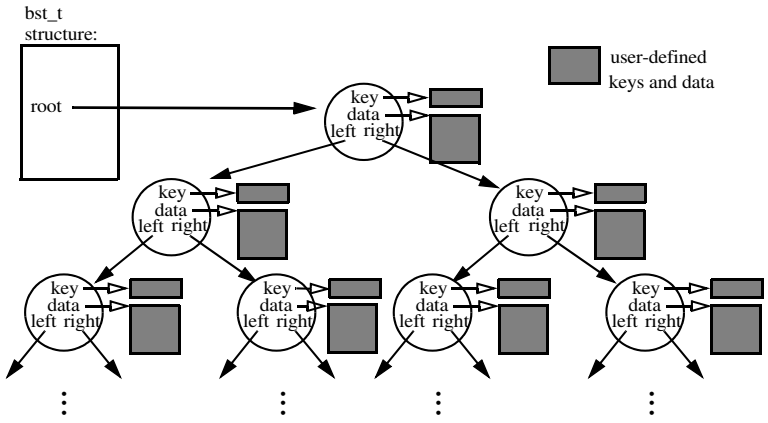

Figure 1: Facilitating user-defined keys and data (shaded boxes) by pointing to them from the BST nodes using generic pointers (white arrowheads).

(*e.g.*, a single int), arrays (*e.g.*, a string), or structures with a large number of fields. (This will permit your ADT to be very flexible, and usable in a large number of applications). To facilitate this, each BST node will employ a self-referencing structure defined in your ADT, but rather than embed the user's key and data in each struct, your BST node will instead point to the user-defined key and data through *generic pointers*.

Figure 1 illustrates the components of the BST. In Figure 1, the user-defined keys and data are denoted by shaded boxes while ADT structures are denoted by white boxes and circles. Furthermore, normal pointers are denoted by arrows with filled arrowheads while generic pointers are denoted by arrows with white arrowheads. Figure 1 shows the main structure of the ADT of type "bst_t" which contains a pointer to the BST root node (there are other fields in this structure as well, which we will describe later). The root node–and every other BST node–contains "left" and "right" pointers that point to the left and right sub-trees underneath the node. It also contains a "key" and "data" field which are generic pointers that point to the user-defined key and data for that node.

1.2 User-Defined Sort Order

Since keys are user-defined, so must be the sort order of keys. Your ADT will facilitate this by allowing users to define their own compare function. When allocating a BST ADT, the user will provide a function pointer that points to their compare function. This function compares two user-defined keys, and returns an integer that is a negative number, 0, or a positive number depending on whether the first key is less than, equal to, or greater than the second key. (The definition of smaller, equal, or larger is up to the user). You will record this function pointer in your ADT (in the main BST structure in Figure 1). Whenever your ADT needs to perform a comparison between two keys (*i.e.*, during BST traversal for insert and search operations), you will call the user-defined compare function through its function pointer.

1.3 Examining Data in Sorted Order

In addition to insert and search, your BST ADT will also permit users to examine the data stored in the BST in sorted order. Specifically, users can command the BST ADT to "walk" the sorted data and present it back to the user one at a time in sorted order. Your ADT will support a "start walk" function that will initiate the walk and present the first piece of data back to the user. Your ADT will also support a "next walk" function; successive calls to "next walk" will return the next piece of data in sorted order until all data in the BST have been examined.

1.4 ADT Interface

The interface for the BST ADT consists of a structure template, "bst_t," along with 7 public functions that implement the operations defined on the BST. The structure template and public function forward declarations are defined in "bst.h" which can be downloaded from the course website (just follow the "Project 4 Files" hyperlink). *You must implement these structure and functions in your ADT.* (You may add other structures–for example a "node" structure–and private functions, but you cannot omit any of the ones defined in bst.h).

You should fill in the structure template in bst.h (the version provided is empty). Also, you should create a file, "bst.c," that implements the 7 required public functions along with any other private functions you need. Together, the bst.h and bst.c files make up the ADT. The following describes the 7 public functions you must implement:

1. `Pbst_t new_bst(int (*cmp_func)(void *, void *))`

This is the ADT constructor. This function allocates a BST structure of type bst_t on the heap, and returns a pointer to it. The function takes a single argument, the function pointer to the user-defined compare function. The allocated BST should be empty (*i.e.*, the root pointer in the BST structure points to NULL). The statistics for the ADT (number of inserted items and height) should be initialized to zero.

2. `void free_bst(Pbst_t bst)`

This is the ADT destructor. This function deallocates the BST ADT pointed to by the argument passed into the function. All structures dynamically allocated as part of the ADT should be freed after this function is called. In particular, if the BST is not empty, the function should traverse every node in the BST and deallocate the node along with the key and data pointed to by the generic pointers in the node. (Assume each user-defined key and data can be deallocated by simply calling "free" on the generic pointer that points to it). Any heap objects associated with walking the BST (see below) should also be freed.

3. `void insert_bst(Pbst_t bst, void *key, void *data)`

This function inserts a new piece of data into the BST. The function takes 3 arguments: a pointer to the BST ADT, a generic pointer to the user-defined key associated with the new data, and a generic pointer to the user-defined data to be inserted. As mentioned earlier, to find the proper insertion point, you must traverse the BST starting from the root. You should use the user-provided compare function to determine the direction of traversal (left or right) at each traversed node. If at any point your traversal encounters a node with exactly the same key (*i.e.*, the compare function returns "0"), you should abort the insert operation and leave the BST unmodified. If the insert operation succeeds, you should update the statistics (number of items and height) associated with the BST.

4. `void *find_bst(Pbst_t bst, void *key, int *depth)`

This function searches for a key in the BST, and if found, returns the generic pointer to the data associated with the key. The function takes 3 arguments: a pointer to the BST ADT, a generic pointer to the user-defined key, and the depth in the BST where the key was found (passed by reference). As mentioned earlier, to find the key, you must traverse the BST starting from the root. You should use the user-provided compare function to determine the direction of traversal (left or right) at each traversed node. If the function finds the key in the BST, it returns a pointer to the user-defined data associated with the key, and sets the "depth" parameter accordingly. If the key was not found in the BST, the function returns NULL and the "depth" parameter is not set.

5. `void stat_bst(Pbst_t bst, int *num_items, int *height)`

This function returns 2 statistics associated with a BST: the total number of data items stored in the BST, and the height of the BST. The function takes 3 arguments: a pointer to the BST ADT, and the 2 statistics passed in by reference. The function should return the 2 statistic values to the user through the pass-by-reference arguments.

6. `void *start_bst_walk(Pbst_t bst)`

This function commands a BST ADT to initiate an ordered walk of the data stored in the BST, and to return the first data item in the sorted order. The function takes a single argument which is a pointer to the BST ADT. This function can be called any number of times on the same BST ADT. Each time it is called, a new ordered walk is initiated, and the first data item in the sorted order is returned.

While there are potentially many ways to implement the BST walk, one way (recommended) is to create an array of generic pointers, one for each data item stored in the BST, and to initialize the generic pointers so that they point to the data items in the BST in sorted order in the pointer array. That way, start_bst_walk can simply return the first pointer in this pointer array while each subsequent call to next_bst_walk can return the next pointer in this pointer array. Initialization of the generic pointers can be done by performing a left-to-right traversal of the BST which will visit the nodes in sorted order (see the tree-search2.c example from lecture), and setting successive pointers in the pointer array at each traversed node.

7. `void *next_bst_walk(Pbst_t bst)`

This function returns the next data item in the sorted order immediately following the data item returned most recently by a call to either next_bst_walk or start_bst_walk–*i.e.*, it continues a "sorted walk" of the data items stored in the BST. The function takes a single argument which is a pointer to the BST ADT. After returning the last data item in the sorted order, subsequent calls to next_bst_walk should return NULL until another call is made to start_bst_walk.

2 ADT Applications

In addition to creating the BST ADT, you will also modify two examples from lecture, tree-search.c and parse.c, to use your ADT. Download these examples from the course website, and modify them according to the discussion below. Note, both of these examples *must use a single version of the BST ADT code* that you have created, thus demonstrating the polymorphic feature of your ADT.

2.1 tree-search.c

You are to rewrite this program to use your BST ADT to implement the tree data structure used in the program. In this example, both the user-defined key and data will be a single integer value per BST node. And the compare function is simply the "<", "==", and ">" operators on integers.

Specifically, you are to make the following changes to the tree-search.c program. Remove the typedef of "data_el" at the top of the file, and remove the "insert" and "search" functions. To add your BST ADT, you should insert a call at the beginning of "main" to "new_bst" to create an ADT. You will need to create a comparison function that simply does the integer comparison mentioned above, and returns a negative number, 0, or a positive number accordingly. (Note, the keys are passed into this function as generic pointers. You will need to cast these generic pointers to pointers to ints before performing the comparisons).

Next, you should modify the loop inside the "create_data" function to malloc a single integer instead of a data_el struct each loop iteration, and to only set the malloc'd integer to the randomly generated number. Then, call "insert_bst" to insert the new value into your BST ADT rather than calling "insert". At the end of the "create_data" function, you should also call "stat_bst" and print the number of nodes in the BST as well as its height. (Note, because your BST ADT will discard duplicates, the number of nodes will be less than 10,000).

Finally, modify the infinite loop in "main" to call "find_bst" instead of "search". Also, move the printf code that was in "search" into main to print the result–whether or not you found the value in the BST, and if you found it, the number of tries it took. Note,

the number of tries needed to find the value is simply the "depth" parameter passed by reference into "find_bst".

2.2 parse.c

You are to rewrite this program to use your BST ADT to implement the dictionary used in the program. In this example, each user-defined data item will be a struct that includes two items: the word in the dictionary (a string of length WORD_SIZE characters), and a single counter that keeps track of the number of times (frequency) the associated word has been encountered. The word stored in each data struct is also the key for that piece of data. The compare function on keys is simply "strcmp". This will keep words in the dictionary sorted in alphabetical order.

Specifically, you are to make the following changes to the parse.c program. Remove the "dictionary" and "frequency" arrays declared in "main". To add your BST ADT, you should insert a call at the beginning of "main" to "new_bst" to create an ADT. You will need to create a compare function that calls "strcmp", and returns its return value (strcmp already returns a negative number, 0, or a positive number). (Note, the keys are passed into this function as generic pointers. You will need to cast these generic pointers to pointers to chars before calling strcmp).

Once you have created the BST ADT, you should modify the parse.c program to use it. In the parse loop of "main", instead of calling "insert_word," you will call "insert_bst." Instead of calling "find_word," you will call "find_bst." Lastly, in the "dump_dictionary" function, you should first call "stat_bst" and print the number of nodes in the BST as well as its height. Then, you should call "start_bst_walk" to initiate a walk of all of the dictionary entries, and then call "next_bst_walk" to get the dictionary words in sorted order, printing the word and frequency value each time. (Note, the original parse.c program only printed words encountered more than once; in this new parse.c program, you should print *all* words encountered regardless of how many times they were encountered). This will dump the dictionary with all the words appearing in *alphabetical order*.