William Montgomery
Spring 2013

tmap

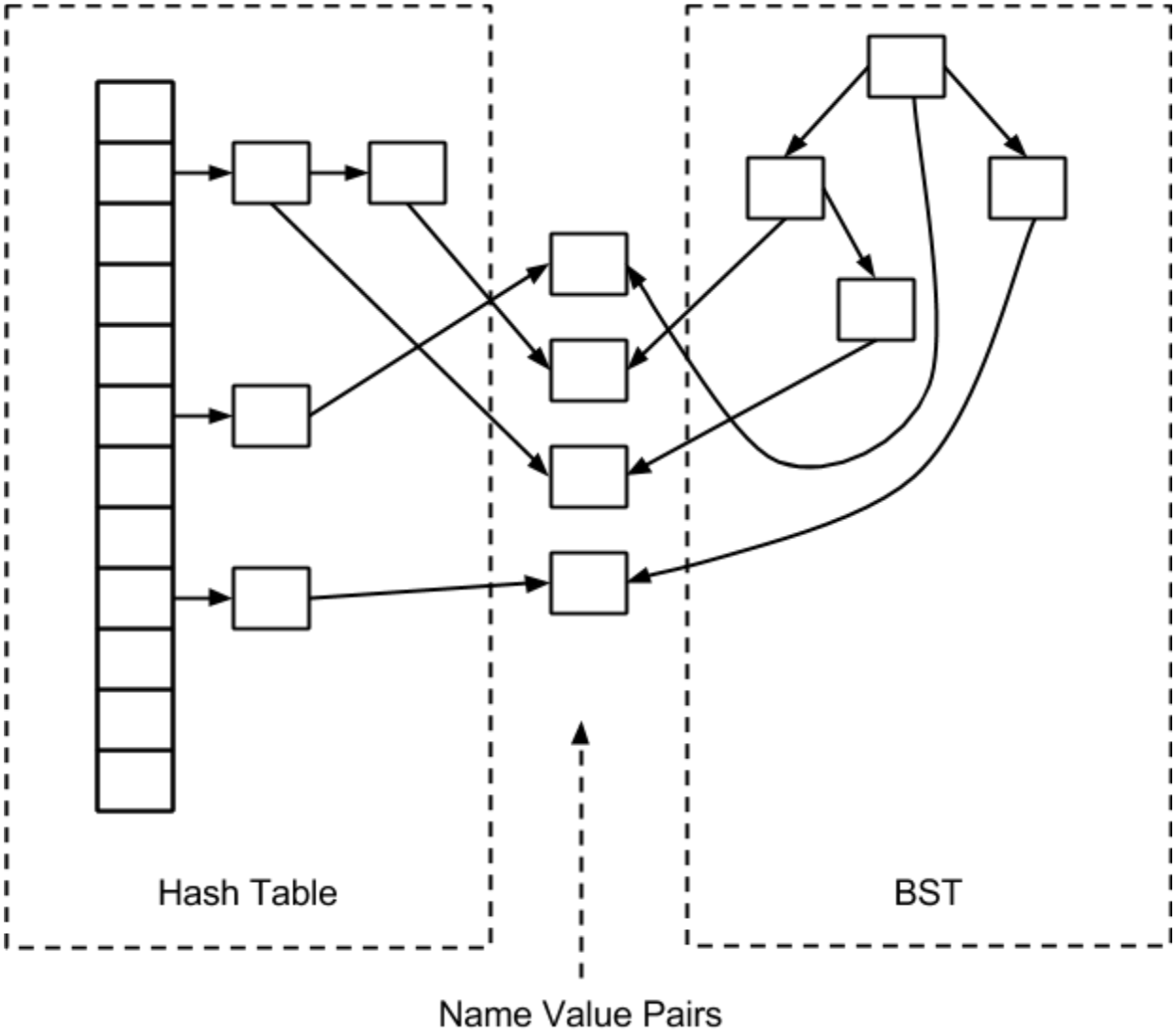
This tmap maps strings to double values. It uses a combined hash table and binary sorted tree to support the following operations (with the runtime requirements if any):

Function	Runtime	Comments
tmap_create		
tmap_insert	O(log n) (ammortized)	Updates the associated value and modifies the tree.
tmap_size	O(1)	
tmap_height	O(1)	
tmap_get_ith	O(log n)	Returns the ith member.
tmap_range_size	O(log n)	Returns the number of elements within a range.
tmap_extract_range	O(m + log n)	Returns the elements in a range in an array.
tmap_stats		
tmap_destroy		
tmap_name2value	O(1)	Bonus.
tmap_remove_by_name	O(log n)	Bonus.

Data structures

The primary data structure is a tree, with an additional hash table to used for looking up by strings. The tree and the hash were built as abstract data types, with the data type specified in a header file. Function pointers were used heavily to support comparisons, printing, and freeing of the stored elements. The hash table uses separate chaining to insert keys that map to the same hash value.

Diagram



Files

In order to implement the structures as abstract data types, each data type was written in a separate .c file with an accompanying .h file. The binary search tree is contained in the bst.c and bst.h files. The hash table is contained in the ht.c and ht.h files. The tmap itself is contained in the tmap.c and the tmap.h files.

Testing

In order to test the functionality of the code, tests were written for every exposed interface. Data was filled in three ways, from an input file, from pseudo-random input data one by one, and from a sorted array (to test the BST only). There were up to 10000 nodes inserted using each of the methods. Valgrind was used to ensure that no memory leaks exist.
It was noted that the creation of a pseudo-random array, sorting of the array, and then building

the BST from that array was much faster than the single insertion of each element. Both operations are O(n log n) for the problem size of n elements. This is likely because the sorting of the array was much more efficient than the rebalancing of the binary tree. For large datasets, using an array to insert the data would increase performance dramatically, although care must be taken to remove duplicates.

Compilation

A Makefile was created to automate the compiling and linking of files. The compilation was tested on my MacBook and on bert.cs.uic.edu using gcc. The version of gcc on bert requires that the -std=c99 flag be set. This does not interfere with compilation on the MacBook. It is recommended that an optimization level of at least 2 be set, with the -O2 flag.

The makefile is included for convenience of compilation.

Range Functions

In order to support the range functions, the only augmentation was to add the size of the subtree at each node.

The runtime requirements to retrieve the number of elements between the range was O(log n). Initially, I chose to find the value by finding the minimum and traversing the tree and adding to the count for each value that was within the range. This approach was O(log n + m) and did not meet the runtime requirements. I ended up by implementing three recursive functions, all of which are O(log n). The first recursive function simply walks down the tree, until it finds the first node within the range. It then calls a recursive function on the left and a recursive function on the right. The function on the left follows these rules:

- If the node is within the range, return the number in the range in the left subtree (calculated in O(log n)) plus the size of the right subtree (this is O(1) because it is just a lookup) + 1
- If the node is not within the range, recursively return the number in the range of the right subtree.
- If the node is null, return 0.

A similar method is used for the right recursive function.

The runtime requirements to retrieve all of the elements in the range is O(log n + m). This function makes a recursive call based on these rules:

- If the node is within the range
    - Call the function on the left subnode
    - Add the node
    - Call the function on the right subnode
- If the node is less than the range
    - Call the function on the right subnode
- If the node is more than the range
    - Call the function on the left subnode

- If the node is null, return 0

The general idea is that the tree is walked down until a node in range is found. From there it walks down on the left and then on the right, adding all nodes within the range on the way down. The walking down is a O(log n) operation, while the compiling of the elements is O(m) where m is the number within the range. Thus the function is O(log n + m).

Bonus Operations

In order to support the bonus operations, a hash table was used. The hash table used a reasonably good hash function and reallocated when the fill rate was at 75%. Since this hash table module was used from the prior project, the table has been tested thoroughly. The statistics show that the average number of elements mapped to a non-empty bucket is around 1.2 and the longest chain was less than 10. This means that lookups from the hash table are O(1). For the update by name, a lookup to find the value is made (O(1)). Then a call to delete that node is made (O(log n)). A new node is then created and inserted (O(log n). Since all of these are O(log n), the function is O(log n).

Size Balanced Height Proof

A proof that any size-balanced binary tree has height `O(log n)`. By now this should be a pretty easy exercise in mathematical induction for most of you.

Definitions:

A binary tree is size-balanced if, for all nodes, the size of the left subtree is less than or equal to the size of the right subtree plus one and the size of the right subtree is less than or equal to the size of the left subtree times 2 plus one. Also, the size of a leaf node, is defined to be 0. Due to the nature of size-balanced subtrees, the height of the tree must be 1 + the height of the left subtree. Thus the recurrence relation is as follows:

$$H(n) = H(2/3\ n) + 1$$

Claim:

The claim is that:

$$H(n) \leq c \log n \ \forall\ n \geq n_0 \text{ where } c \text{ and } n_0 \text{ are constants}$$

WLOG, assume that the size of the left subnode is 2 times the size of the right subtree plus one.

Basis:

For the basis case, we will use a tree size of 1.

$H(1) = 0$ by definition.

$0 \leq c \log 1 \ \forall$ values of c

The basis case holds, therefore the proof can proceed.

Inductive Hypothesis:

Assume:

William Montgomery
Spring 2013

$H(k) = c * log\ n\ for\ some\ k \geq 1$

Prove:
$H(k + 1) \leq c * log(n + 1)$

Proof:
There are four cases when a new node is added. If it is added to the larger subtree, it can either add one to the height of the subtree or the height of the subtree can remain the same. If it is added to the smaller subtree, it can either add one to the height of the subtree or the height of the subtree can remain the same. Because the addition must hold the properties of a balanced bst, the height of the tree will remain the same, which is O(log k) by the inductive hypothesis.

If the height of the left subtree remains the same, the height of the the tree is O(log k). If the height increases by one, the height is O(log (k + 1)) which reduces to O(log k).

Therefore, in all cases, the height is O(k) and the proof holds.