Event Counter Project Report

COP5536 - Paul Nickerson

General Info

My event counter implementation is written in C++, heavily utilizing modern features from the C11 standard for robustness and code readability. The structure relies on traditional object-oriented approaches; the event counter class inherits from an AVL class, which inherits from a binary search tree (BST) class.

In the BST, the class holds a node array, which is 1-based. Nodes reference children by storing the index to the child within the node array, rather than a pointer to the node itself. “Node-not-found” is indicated when node index 0 is encountered, since that node does not exist in the node array.

Actually there are two trees under the hood – the main tree, which holds the actual data stored by the BST abstract data type (ADT), and a free tree, which acts as a pool of unoccupied nodes. All nodes belonging to either tree are held in the nodes array. The free tree has a similar structure to a linked list, where free tree traversal proceeds down to the left (the right child is never occupied in the free tree). Class initialization populates the free tree with as many nodes as is specified by the initial capacity, which is determined on class instantiation. The main tree is initially un-populated. Node procurement pops the root node from the free tree and adds it to the main tree, then sets the free tree root node to the left child of the old free tree root node.

To initialize the tree in O(N) time, I implemented an algorithm similar to binary search. The main tree root is set to the middle node within a given sorted list of key-value pairs (note that this involves procuring an unoccupied node from the free tree). The left child is set to the middle node of first half of the sorted input array; while the right child is set to the middle node of the second half. The process repeats recursively until all nodes are inserted into the tree.

The AVL class inherits from the BST class, and wraps inherited methods in a rebalance operation where the inherited might change the height of a subtree. In this way the balance factor is kept within a [-1, 1] interval.

Many methods include debugging code which validate tree structure; however, this code is generally quite slow (e.g. validating balance factors is a O(N) operation, which is slow compared to many of the O(lg N) operations). For this reason the debugging code is enabled or disabled by way of a boolean DEBUG preprocessor macro variable in the main.cpp file.

Main.cpp simply creates an instance of the Driver class and passes input to it. The driver creates an instance of the event counter and populates it with the key-value pairs includes in the input file. The driver then receives input lines, parses them for the command functions and parameters, and performs operations on the event counter based on those commands, while printing the output. When the driver receives the *quit* command, it exits the program.

Class Diagrams (public members)

|  |
| --- |
| EventCounter |
| EventCounter(*in* init\_capacity)  EventCounter(*in* kv\_list)  uint64\_t increase(*in* id, *in* m)  uint64\_t reduce(*in* id, *in* m)  kv\_pair next(*in* id)  kv\_pair previous(*in* id)  uint64\_t count(*in* id)  void in\_range(*in* id1, *in* id2, *out* values) |

|  |
| --- |
| AVL Tree |
| AVL(*in* init\_capacity)  AVL(*in* init\_kvs)  int insert(*in* key, *in* value)  int remove(*in* key, *out* value)  is a |

|  |
| --- |
| Binary Search Tree |
| BST(*in* init\_capacity)  BST(*in* init\_kvs)  int insert(*in* key, *in* value)  int remove(*in* key, *out* value)  int search(*in* key, *out* value)  void clear()  bool is\_empty()  size\_t capacity()  size\_t size() |

is a

|  |
| --- |
| Node |
| size\_t validate\_children\_count\_recursive(in nodes)  size\_t get\_height\_recursive(in nodes)  void update\_height(in nodes)  void disable\_and\_adopt\_free\_tree(in free\_index)  void reset\_and\_enable(in new\_key, in new\_value)  int balance\_factor(in nodes) |

|  |
| --- |
| Driver |
| bool load\_file(*in* inp\_f)  void run\_cmd(*in* line) |

Note: The Node class a protected member of the BST class; therefore is only accessible to that class and those classes which inherit from it. BST, AVL, and EventCounter are accessed via keys and values, not nodes

Node class members

**size\_t validate\_children\_count\_recursive(nodes):**

**#this function is for debugging purposes (ie normally disable for the release version)**

**#does recursive traversal to find the correct number of children**

**child\_count = 1 + left\_child.validate\_children\_count\_recursive if left\_child exists**

**+ 1 + right\_child.validate\_children\_count\_recursive if right\_child exist**

**if child\_count != stored num\_children:**

**throw an error**

**return child\_count**

**size\_t get\_height\_recursive(nodes):**

**#this function is for debugging purposes (ie normally disabled for the release version),**

**#does recursive traversal to find the correct height**

**return 1 + std::max(left\_subtree.get\_height\_recursive height, right subtree height)**

**void update\_height(nodes):**

**this.height = height = 1 + std::max(left\_height, right\_height);**

**if DEBUG:**

**verify this.height == this.get\_height\_recursive. If not, throw an error**

**void disable\_and\_adopt\_free\_tree(size\_t free\_index):**

**mark the current node as unoccupied and set the left child to point to the root of the free tree**

**Adds current node to the pool of unoccupied nodes**

**void reset\_and\_enable(new\_key, new\_value):**

**mark the current node as occupied and having no children**

**set the nodes key and value to that provided from the input**

**int balance\_factor(nodes):**

**return left\_child.height - right\_child.height**

BST (Binary Search Tree) class members

## **Protected**

**size\_t remove\_smallest\_key\_node\_index(out subtree\_root\_index)**

**#recursive function**

**#returns the index of the node with the smallest key, while**

**#setting its parent's left child index to the smallest key node's**

**#right child index. recursion downward through this function updates**

**#the heights of the nodes it traverses**

**subtree\_root = nodes[subtree\_root\_index]**

**if subtree\_root has left child:**

**smallest\_key\_node\_index = remove\_smallest\_key\_node\_index(subtree\_root.left\_index)**

**decrement subtree root children counter, O(1)**

**update subtree root height, O(1)**

**else:**

**#current root is the smallest (left-most child)**

**smallest\_key\_node\_index = subtree\_root\_index**

**subtree\_root\_index = subtree\_root.right\_index**

**return smallest\_key\_node\_index**

**size\_t remove\_largest\_key\_node\_index(out subtree\_root\_index)**

**#recursive function**

**#returns the index of the node with the largest key, while**

**#setting its parent's right child index to the largest key node's**

**#left child index. recursion downward through this function updates**

**#the heights of the nodes it traverses**

**subtree\_root = nodes[subtree\_root\_index]**

**if subtree\_root has right child:**

**largest\_key\_node\_index = remove\_largest\_key\_node\_index(subtree\_root.right\_index)**

**decrement subtree root children counter, O(1)**

**update subtree root height, O(1)**

**else:**

**#current root is the largest (right-most child)**

**largest\_key\_node\_index = subtree\_root\_index**

**subtree\_root\_index = subtree\_root.left\_index**

**return largest\_key\_node\_index**

**void remove\_node(out subtree\_root\_index):**

**subtree\_root = nodes[subtree\_root\_index]**

**if subtree\_root has at least one child:**

**if subtree root has right child:**

**replace the root with the smallest-keyed node in the right subtree**

**else:**

**#otherwise, it has a left child:**

**replace the root with the largest-keyed node in the left subtree**

**Set the node referenced by subtree\_root\_index as the new subtree root, and**

**have it adopt the children of the old root**

**remove subtree\_root from the main tree and add it to the free tree**

**int do\_remove(nodes\_visited, out subtree\_root\_index, key, out value, out found\_key)**

**recursively traverse the tree until we find the node storing "key"**

**set found\_key output argument to whether or not the node was found**

**if found:**

**remove\_node(index of the matching node)**

**update height and number of children when bubbling back up the tree**

**set value output argument = the value of the matching node**

**void add\_node\_to\_free\_tree(node\_index):**

**disable node reference by node\_index and add to the free tree**

**size\_t procure\_node(key, value)**

**get an unoccupied node from the free tree, updating the free tree to point to that nodes left child**

**activate the procured node and set its key/value**

**return the new node index**

**int insert\_at\_leaf(nodes\_visited, out subtree\_root\_index, key, value, out found\_key)**

**recursively traverse down the tree until either fall off or find the node matching "key"**

**if found:**

**set node.value = value**

**else:**

**procure a new node, setting its key/value to that provided by the input**

**append the node to the bottom of the tree**

**update num\_children and height on the way back up**

**set found\_key output argument to whether or not matching node was found**

**int do\_search(nodes\_visited, subtree\_root\_index, key, out value, out found\_key)**

**recursively traverse down the tree until either fall off or find the node matching "key"**

**if found:**

**set value output argument to the matching nodes value**

**set found\_key output argument to whether or not matching node was found**

**void increase\_capacity():**

**double the size of the node array**

**add newly-created nodes to the free tree**

**size\_t init\_from\_kv\_list(kv\_list, start\_idx, end\_idx):**

**procure a node, n, for the key/value pair stored at kv\_list[(end\_idx + start\_idx) / 2]**

**#left child gets set based on first half**

**n.left\_child = init\_from\_kv\_list(kv\_list, start\_idx, (end\_idx + start\_idx) / 2)**

**#left child gets set based on second half**

**n.right\_child = init\_from\_kv\_list(kv\_list, (end\_idx + start\_idx) / 2, end\_idx)**

**return n's index in the node array**

## **Public**

**BST(init\_capacity):**

instantiate init\_capacity number of nodes into the node array

**BST(init\_kvs):**

initialize the main tree from a sorted list of key-value pairs

**int insert(key, value):**

if tree is at capacity, increase the capacity

insert key-value pair as a leaf node

**int remove(key, out value)**

remove the node associated with specified key from the tree.

if found, set value output argument to what that was

**int search(key, out value)**

look for a node associated with "key"

if found, set value output argument to what that is

**void clear():**

disable all occupied nodes and put them in the free tree

**bool is\_empty():**

return true if main tree is empty, false otherwise

**size\_t capacity():**

return current main tree capacity

**size\_t size():**

return the number of children in the main tree

AVL class members

## **Protected**

**int insert\_at\_leaf(nodes\_visited, out subtree\_root\_index, key, value, out found\_key):**

call BST::insert\_at\_leaf, then rebalance the tree if necessary

**size\_t remove\_smallest\_key\_node\_index(out subtree\_root\_index):**

call BST::remove\_smallest\_key\_node\_index, then rebalance the tree if necessary

**size\_t largest\_key\_node\_index(out subtree\_root\_index):**

call BST::largest\_key\_node\_index, then rebalance the tree if necessary

**int do\_remove(size\_t nodes\_visited, out subtree\_root\_index, key, out value, out found\_key):**

call BST::do\_remove, then rebalance the tree if necessary

**void rotate\_left(out subtree\_root\_index):**

subtree\_root.right\_child = right\_child.left\_child

right\_child.left\_child = subtree\_root

#update number of children and height variables as necessary - O(1)

#make the right child the new subtree root

subtree\_root\_index = right\_child\_index

**void rotate\_right(out subtree\_root\_index):**

subtree\_root.left\_child = left\_child.right\_child

left\_child.right\_child = subtree\_root;

#update number of children and height variables as necessary - O(1)

#make the left child the new subtree root

subtree\_root\_index = left\_child\_index

**void balance(out subtree\_root\_index):**

if (subtree\_root.balance\_factor == -2):

#right subtree is too heavy

switch(subtree\_root.right\_child.balance\_factor):

case 1:

#right left

rotate\_right(right\_index)

rotate\_left(subtree\_root\_index)

case -1 or 0:

#right right

rotate\_left(subtree\_root\_index)

if (subtree\_root.balance\_factor == 2):

#left subtree is too heavy

switch(subtree\_root.left\_child.balance\_factor):

case -1:

#left right

rotate\_left(left\_index)

rotate\_right(subtree\_root\_index)

case 1 or 0:

#left left

rotate\_right(subtree\_root\_index)

**void validate\_avl\_balance(subtree\_root\_index):**

verify subtree\_root's balance factor is within [-1, 1]

validate\_avl\_balance(left child)

validate\_avl\_balance(right child)

**size\_t init\_from\_kv\_list(init\_kvs, start\_idx, end\_idx):**

call BST::init\_from\_kv\_list, then rebalance the tree if necessary

#note: experiments seem to indicate that rebalancing is not necessary here

## **Public**

**int insert(key, value):**

#similar in functionality to BST::insert, but with rebalancing functionality

#and optional validation functionality when debugging mode is on

**int remove(key, value):**

#similar in functionality to BST::remove, but with rebalancing functionality

#and optional validation functionality when debugging mode is on

EventCounter Class Members

## **Protected**

**bool do\_find\_next(subtree\_root\_index, search\_k, out found\_k,**

**out found\_v, out nodes\_visited):**

#do in-order traversal to find the first key which is

#greater than search\_k, while skipping subtrees that cant

#possibly contain a match (makes this algorithm O(lg N)).

#for a given subtree\_root, there are 3 possiblities:

#1 - subtree\_root.key > search\_k and something in left

# subtree was greater than search key, in which case we

# stop traversal because thats the match

#2 - subtree\_root.key > search\_k and everything in left subtree

# was less than search key, in which case we stop traversal

# because current node is the match

#3 - subtree\_root.key <= search\_k, in which case the

# match may be in the right subtree

return number of nodes visited, which is guaranteed to

equal the height of the tree

**bool do\_find\_previous(subtree\_root\_index, search\_k, out found\_k,**

**out found\_v, out nodes\_visited):**

#do in-order traversal backwards to find the first key which is

#less than search\_k, while skipping subtrees that cant

#possibly contain a match (makes this algorithm O(lg N)).

#for a given subtree\_root, there are 3 possiblities:

#1 - subtree\_root.key < search\_k and something in right

# subtree was less than search key, in which case we

# stop traversal because thats the match

#2 - subtree\_root.key < search\_k and everything in right subtree

# was greater than search key, in which case we stop traversal

# because current node is the match

#3 - subtree\_root.key >= search\_k, in which case the

# match may be in the left subtree

return number of nodes visited, which is guaranteed to

equal the height of the tree

**void do\_in\_range(subtree\_root\_index, k\_l, k\_r, out values, out nodes\_visited):**

#do in-order traversal to find keys which are between k\_l and

#k\_r (inclusive), while skipping subtrees that cant possibly contain a match

if (subtree\_root.key >= k\_l):

recurse to left

if (subtree\_root.key <= k\_r):

current node.key is within the range, so add value to the list

if (subtree\_root.key <= k\_r):

recurse to right

## **Public**

**uint64\_t increase(id, m):**

**Increase the count of the event ID by m. If ID is not present, insert it.**

**Return the count of ID after the addition.**

**uint64\_t reduce(id, m):**

**Decrease the count of ID by m. If ID’s count becomes less than or equal to 0,**

**remove ID from the counter.**

**Return the count of ID after the deletion, or 0 if ID is removed or not present.**

**kv\_pair next(id):**

**Return ID and count of the event with lowest ID that is greater than ID.**

**Use do\_find\_next protected recursive function**

**Return “0 0” if there is no next ID.**

**kv\_pair previous(id)**

**Return ID and count of the event with greatest ID that is less than ID.**

**Use do\_find\_previous protected recursive function**

**Return “0 0” if there is no previous ID.**

**uint64\_t count(id):**

**Return the count of ID. If not present return 0.**

**void in\_range(key\_type id1, key\_type id2, value\_list& values):**

**Return the total count for IDs between ID1 and ID2 inclusively. Note ID1 ≤ ID2.**

**Use do\_in\_range protected recursive function**

Driver class members

## **Protected**

**void split(str, delim, out parts):**

use strtok c function to split a copy of input string by given delimiter

push each resulting component to parts output vector

**std::string str\_to\_lcase(str):**

#make a copy of an input string, with all letters converted to lowercase

**void read\_kvs(if\_handle, out kvs\_out):**

while if\_handle has another line:

grab the line

split it using a space token

if the line had two components:

create a key-value pair and push it to the kvs\_out vector

**bool read\_inp\_f(if\_name, out kvs):**

retrieve a handle to file if\_name

read\_kvs(if\_handle, kvs)

close the handle

**bool increase(parts):**

print the results of ec.increase(parts[1], parts[2])

**bool reduce(parts)**

print the results of ec.reduce(parts[1], parts[2])

**bool inrange(parts):**

print the results of ec.inrange(parts[1], parts[2])

**bool next(parts):**

print the results of ec.next(parts[1])

**bool previous(parts)**

print the results of ec.previous(parts[1])

**bool count(parts):**

print the results of ec.count(parts[1])

## **Public**

**bool load\_file(inp\_f):**

**#set the current copy of the event counter to one instantiated with the given input file name**

**void run\_cmd(line):**

**split line by a space delimiter**

**cmd = parts[0]**

**if (cmd == "increase"):**

**increase(parts)**

**else if (cmd == "reduce"):**

**reduce(parts)**

**else if (cmd == "inrange"):**

**inrange(parts)**

**else if (cmd == "next"):**

**next(parts)**

**else if (cmd == "previous"):**

**previous(parts)**

**else if (cmd == "count"):**

**count(parts)**

**else if (cmd == "quit"):**

**exit(0)**