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)

EventCounter —is a→ AVL Tree

AVL Tree

- AVL(*in* init_capacity)
- AVL(*in* init_kvs)
- int insert(*in* key, *in* value)
- int remove(*in* key, *out* value)

AVL Tree —is a→ Binary Search Tree

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()

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)

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

---

Driver

- bool load_file(*in* inp_f)
- void run_cmd(*in* line)

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)
```