TRINITY COLLEGE DUBLIN, SCHOOL OF MATHEMATICS

Assignment #3, Module: MA5611

Gustavo Ramirez

February 25, 2017

1 PART 1: SERIAL

In this case, the 'main' function (the only thread within the process executed by this program) is in charge of the three tasks: insert, extract and re-balance. After each insertion or extraction, a re-balancing is performed, but only if the depth of the tree changes, as generally these BSTs are used for searching, in which case keeping the depth as small as possible is the main goal.
Description of the implemented functions:

**** core functions:**

- **insert(int key, struct node** tree):** without a returning value, this function simply inserts the passed int 'key' into the tree to which the pointer *tree points. If that key already exists within the tree, then no insertion is performed.
- **extract_elem(int, struct node**):** similar to 'insert': if the int doesn't exist, no extraction is performed, and if it exists, then that node is deleted and the tree is reorganized to fill that hole.
- **balance(struct node**):** for the specific implementation here, all the values (all the 'key's) in the tree are extracted in a inorder fashion, then stored in a 1D array, and then a balanced tree is created from that array.
- **build_balanced_tree(struct node**, int*, int, int, int):** this function is called as the last step in the previous function: from a 1D array of sorted keys, a balanced tree is created by recursively splitting that array and calling the 'insert' function.
- **destroy_tree(struct node*):** recursively destroys the tree, leaving in the end the tree's root pointer pointing to NULL.

**** auxiliary functions:**

- **print_tree(struct node*):** prints the tree in a nice formatted way; the only downside up to this point, is that no padding has been added, so that this function prints nicely for numbers from 0 to 9, but the prints shift horizontally if number are >= 10.

- **smallest(struct node**):** returns a pointer to the node in which the smallest key_value of this tree lives.
- **max_depth(struct node*):** returns an integer: the current total depth of the tree.
- **nr_nodes(struct node*, int*):** gives the total number of nodes in the tree.
- **list_inorder(struct node* tree, int* array, int* counter):** given a pointer to the tree, this function stores in 'array' all the keys in the tree, in a sorted manner. Also, stores the length of that array at the variable pointed by 'counter'.

2 PART 2: PARALLEL

This implementation stands on top of the serial implementation described before.
The way the 'main()' thread works is as follows: creates a 'balance' thread which is constantly running on the background, checking (every Δt =5 ms) if the depth of the tree changes, and if that's the case, re-balancing the tree; then creates an 'insert' thread which is assigned the job of inserting a certain amount of random ints into the tree, also doing these insertions a certain amount of time separated from each other, but this time with a separation determined by a temporal Poisson distribution (with mean 0.5 ms); finally, it creates an 'extract' thread, which does the same as the 'insert' thread, time wise at least, but the opposite job (eliminating random ints from nodes).
The values for the Poisson distribution (0.5 ms of expectation value) and for the interval of calls withing the balancing thread (5 ms), where picked like this: for the Poisson distribution, starting from 500 ms for expectation value, successive reductions of that value were tried, up to the point in which the threads were seen to interwind, i.e. not necessarily executing one after the other; for the balancing thread, a frequency of 10 times the expectation value of the Poisson distribution was taken, as less than that might be a waste of resources, and also because of the implementation for detection of 'inactivities' within that thread #2.
After creating all those threads, main() simply calls a join on those threads, waiting for all of them to finish.

**** core functions:**

- 'insert' handler (**inserts**): this function is executed from thread 0, and calls multiple times to the original 'insert' serial function; it implements the appropriate mutex, so that "tree-changing functions" won't access the tree.
- 'extract' handler (**extract_elems**): same as 'insert', but with extractions of keys.
- 'balance' handler (**balance_parallel**): checks the tree every 5 ms, and if the depth changes, then it re-balances the tree (locking the mutex for it).

3 PART 3: ON EXECUTION TIMES

For the parallel implementation, if 30 000 nodes are 'inserted' and 'extracted' (which is never the case, as random integers are taken, and some clashes are present all the time), the execution time is of around 3.895 s total (and is important to take into consideration that all of the operations - insert, extract, balance - are being delayed a small amount, either 5 ms or 0.5 ms on average for Poisson distribution).
On the other hand, if the serial program is run as:

./bbtree-serial -n 30000

i.e. with 1/10 of the nodes for the parallel case, it takes roughly 2.8 s to execute.