Assignments P1.6 - Part 3

General remarks

All assignments are to be programmed in *portable* Fortran 2003. **In addition**, all programs have to be "valgrind clean", i.e. valgrind's memcheck tool should report: *All heap blocks were freed -- no leaks are possible* and also: *ERROR SUMMARY: 0 errors from 0 contexts* unless the reported errors are caused by the compiler (e.g. Intel fortran). The assignments in this part depend on completion of *all* sections from part 1.

16 Linked list

Integrate the provided file *16_array_lookup.f90* file into your build system from parts 1 and 2. *16_array_lookup.f90* provides a framework for testing and benchmarking data structures and looking up the previously stored data in random order. The file expects to read d3_#.dat files with data pairs. You need to implement operations 1) to initialize the data structure, 2) to add items to the list, 3) to look up items by value (i.e. the "key" element of the pair type); this should be a function returning the pair, and 4) to deallocate the data structure completely.

17 Hash table

Now implement a hash table using linked lists as buckets. Use at least 150 "buckets" and use the provided hashing function *hashfunc()* in *hashfunc.f90* which is based on a C language hashing function for integers in *inthash.c*. This function needs to have a number of buckets that is a power of two, so use the *next_size()* function from the hashfunc module to compute that number. Add this to the testing and benchmarking framework as in section 16 and implement the same 4 types of operations. Discuss the performance of looking up elements by value in the three data structures.

18 Stack class with array

Implement a stack class for integers using a dynamically allocated array as backing storage. Implement a default constructor and a copy constructor, a free() function, and push(), pop() and length() methods. Write a test program that instantiates one stack object, fills it with data, then instantiates a second via the copy constructor from the first, and adds more data to both. Empty both stacks in a while loop each. Finally free all allocated storage and end.

19 Stack class with linked list

Program another stack class with the same API as in section 18 and use a linked list as backing storage this time. Discuss benefits and disadvantages of these two kinds of stack classes.

20 Binary search tree

Implement a binary search tree (BST) and test and benchmark it in a similar fashion to the linked list and hash table sections (16 and 17). Since tree traversal is most easily done with recursions, use recursive subroutines and functions for that. Implement functionality to add items to the tree, locate items by value (the “key” entry of the pair data type), and to free all allocated storage. Compare lookup performance to array, linked list and hash table.

21 Tree utilities

Now add the following features to your BST implementation: a function to count the number of nodes (i.e. stored items), a subroutine to extract the content of the tree, ordered by value, into an array of suitable size provided by the calling code, a subroutine to print out the “depth” of the tree and number of nodes with only one leaf.

22 Tree rebalancing

Implement the following strategy to rebalance the tree: 1) extract the content of the tree into a sorted array; 2) allocate a new root node; 3) determine the middle of the array and assign the data at this location to the new root node; 4) rebuild the tree by calling a function that takes an array of data elements with the part of the array before the middle and the part of the array after the middle element; this function determines the middle of the passed in array and adds it to the tree and then recursively calls itself on the two remaining parts. 5) free the old tree and assign the root node pointer with the location of the new tree.
Benchmark the tree lookup performance after the rebalancing and compare depth and number of open leafs before and after the rebalancing.