COMP20003 – Algorithms & Data Structures

Project 1

Name: Peter Mansour

Partner: Stephen Gillespie-Jones

COMP20003 – Algorithms & Data Structures

Introduction: Dictionaries

A dictionary is an abstract programming concept that allows programmers to map keys (of any commutable type) to values (of any type). They are invaluable to programmers as they allow data to be indexed, and so programmers can do basic things like associate a user id with a name, and even to create more complex data structures such as databases. They are provided as a part of many programming language standards, such as Python (using hash tables) and PHP. Dictionaries can be implemented using a few possible underlying data structures. The ones I've looked at here are:

Unordered Array

All records are stored in a normal (linear) C array of a prefixed size.

To search: go through each element and compare the search key with the element's key.

To insert: add the element to the last free index of the array.

Ordered Array

Similar to the unordered array, but all of the elements are ordered by key.

To search: use binary search recursively since the keys are sorted.

To insert: search for the key to see where it should go, add it there, and shuffle all the higher pointers up (to keep the array in order)

Linked List

Each list item (node) consists of a record and a pointer to the next node.

To search: go through each element and compare the search key with the element's key.

To insert: search for the last node and add a pointer from there to the new node.

Binary Search Tree

Each node in the tree consists of a record and two child nodes. The non-linear tree structure allows finding records without comparing with every node in the tree. If given ordered inputs, this structure will be rendered useless as it will effectively just be a slower linked list.

To search: traverse the tree, going right wherever the search key is bigger than the node key, left wherever it is smaller, and returning the node itself if the search key is equal to the node key.

To insert: perform a failed search that returns the parent of where the new node should be, and set one of the parent's children to point to the new node.

Red-Black Tree

Similar to the binary search tree, except that it guarantees that the tree is always perfectly balanced, even for ordered inputs. Links between nodes are color-coded to logically allow nodes to have 2 – and sometimes 3 – keys.

To search: exactly the same as for a perfectly balanced binary search tree.

To insert: same as a normal BST, but also rotate unbalanced nodes and promote the middle nodes of any 3-keyed nodes (to maintain balance).

Theory: Timing & Testing

Expected Time: The expected worst case time for an input of size n for each structure is in the order of:

Structure	Search	Insert
Unordered Array	n	1
Ordered Array	log n	n
Linked List	n	n
Binary Search Tree	n	n
Red-Black Tree	log n	log n

NB: In our implementation, all inserts perform a search first to see whether the element already exists, or where it should go if it doesn't exist. Hence inserts always take at least the same time as searches (if not longer) regardless of the underlying structure.

Testing: To test these structures, I wrote a few C programs to automate some tasks on a large-scale:

- **generate:** Generates inputs and saves them to a file. Command-line arguments specify the number of inputs, output filename, and order (ordered vs random, ascending vs descending). This allows the quick creation of, for example, million-word ordered strings.
- **gen_all:** Generates all three types of inputs (ascending, descending, and random) and saves them to three files. Command-line arguments specify the number of inputs, and the suffix to add to the filenames. This is useful for generating many different sized input files at once, and giving each input size its own suffix, and then using them interchangeably with the dictionaries.
- **test_all:** Given some input sizes (e.g. 10, 100, 1000, etc...), c programs (e.g. testa, testoa, etc...), and the number of times n to try each size in each program (e.g. 5): It generates the inputs and tests each program with ordered and unordered inputs of various sizes n times, averages them out, and then saves the results to an output file.
  For example, say we want to test “testb” and “testrbt” for input sizes 10 and 100 – and we want to do it fifty times just to ensure it's accurate. The program would:
  1. generate inputs of all orders of sizes 10 and 100
  2. measure how long it takes to execute “testb” with an ascending input of size 10 fifty times, then average the time and write it down
  3. do the same for descending and random inputs of size 10
  4. repeat steps 2 and 3 for inputs of size 100
  5. repeat steps 2 – 4 for “testrbt”

Test: Results

The only structure that could efficiently manage significantly large amounts of input (more than 20,000) was the balanced binary search tree: most of the others took more than 10 minutes on a Core i5 laptop.

You may notice that the unordered array and linked list have almost the same performance with all kinds of keys (ascending, descending, random), because they only compare the keys for equality when searching, and don't care about order.

Another thing to note is that, at first, inserts get faster as we add more inputs. This is most likely because of the overhead on the function that we used for measuring time. After a certain threshold, however, they start to get slower again because the dictionaries get filled up, and searches and inserts become more expensive.

For the sake of graphing these results, I will ignore the first one (with 10 inputs) because it's not an accurate measurement, and to prevent a small quadratic shape from appearing at the beginning of the sketch.

COMP20003 – Algorithms & Data Structures

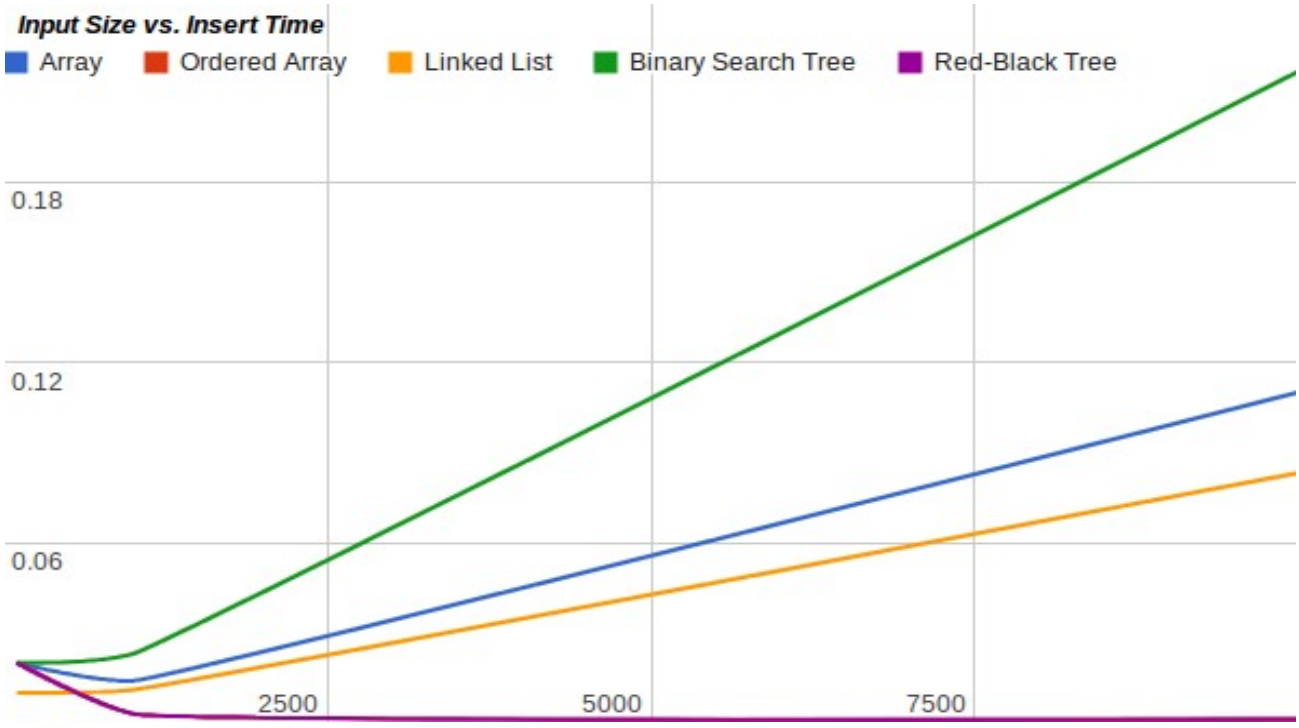
Inputs in ascending order:

Input Size	Unordered Array	Ordered Array	Linked Lists	Binary Search Trees	Red-Black Trees
10	0.2	0.1	0.9	0.1	0.1
100	0.02	0.02	0.01	0.02	0.02
1,000	0.014	0.003	0.011	0.023	0.003
10,000	0.1099	0.001	0.0831	0.2165	0.0014
100,000	-	-	-	-	0.00151
1,000,000	-	-	-	-	0.001891
10,000,000	-	-	-	-	0.0022029

The ordered array performed best with ascending inputs: search time was proportional to log n and insert time was about the same because there was no need to re-order the array.

The red-black tree performed second best (also proportional to log n) but it may have been slightly slower than the ordered array because of its rotations, color flips, and large size.

The binary search tree was the slowest structure for this kind of input because it ended up with all of its nodes on only one side of the tree.



COMP20003 – Algorithms & Data Structures

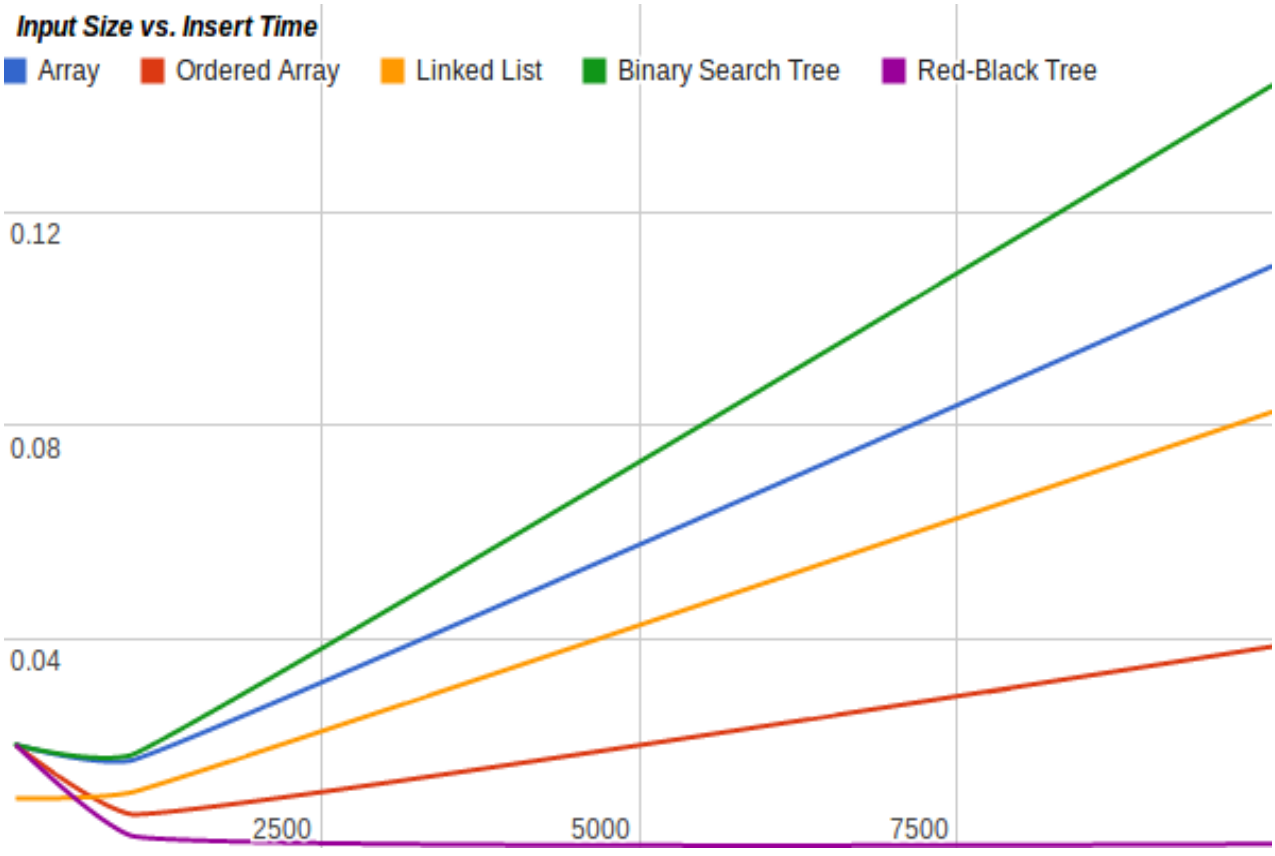
Inputs in descending order:

Input Size	Unordered Array	Ordered Array	Linked Lists	Binary Search Trees	Red-Black Trees
10	0.2	0.1	0.1	0.1	0.2
100	0.02	0.02	0.01	0.02	0.02
1,000	0.017	0.007	0.011	0.018	0.003
10,000	0.11	0.0385	0.0826	0.144	0.0014
100,000	-	-	-	-	0.00167
1,000,000	-	-	-	-	0.002053
10,000,000	-	-	-	-	0.0024943

The red-black tree performed best with descending inputs, and the ordered array clearly lost it's advantage, because now it has to constantly re-arrange its elements.

The array and linked list perform exactly as they did before, because they don't use the order of the keys for any search or insert functionality.

The binary search tree performed significantly better than before because even though it still has most of its nodes on only one side (the left), there is at least one node on the right side of the tree. This tiny saving of at least one comparison per insert may have actually saved 10,000 comparisons!

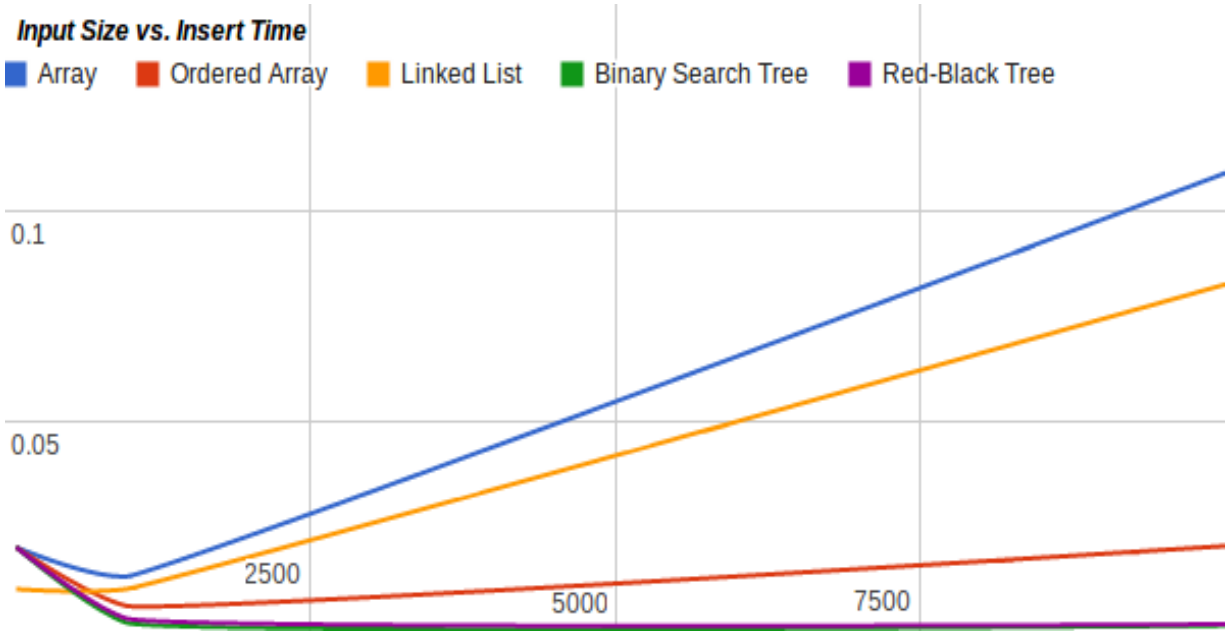


Random Inputs:

Input Size	Unordered Array	Ordered Array	Linked Lists	Binary Search Trees	Red-Black Trees
10	0.2	0.1	0.1	0.2	0.2
100	0.02	0.02	0.01	0.02	0.02
1,000	0.013	0.006	0.01	0.002	0.003
10,000	0.1094	0.0203	0.0828	0.0013	0.0016
100,000	-	-	-	-	0.01133
1,000,000	-	-	-	-	0.003632
10,000,000	-	-	-	-	0.0055208

The binary search tree was the fastest structure for random inputs, since it would have had much more breadth using random keys than ordered ones. It even beat the red-black tree because it doesn't have the overhead of rotating and flipping colors every few nodes.

The ordered array, while not one of the best performers with random input, still performed better than it did with descending input. This could be because, using un-ordered input, it would only shift nodes when it had to, while it used to have to shift most of its nodes in every insert with the descending input.



Conclusion

Through these tests, I found the balanced binary search tree to be the most consistent and efficient data structure for implementing dictionaries. Not only was it one of the top two structures for every kind of input (with only a negligible difference from the first), but it was also the only one that accepted enormous inputs of up to 10 million keys.

The ordered array was also efficient for keys that were already sorted in ascending order, and the normal binary tree was efficient for random keys that were evenly spread out.

The ordered and unordered plain C arrays required memory to be reserved in advance, so the maximum size of the array had to be known from the start, which is not always practical.

Sources

- Red-Black tree implementation by *Robert Sedgewick* and *Kevin Wayne*, fetched from http://algs4.cs.princeton.edu/33balanced on Saturday 18th August 2012
- elapsed_time() function by Joseph Quinsey, written on 5th December 2010, fetched from http://stackoverflow.com/questions/4360073/ on Sunday 19th August 2012