Chris Stasiowski

7795412

05/28/2020

PA02 Report

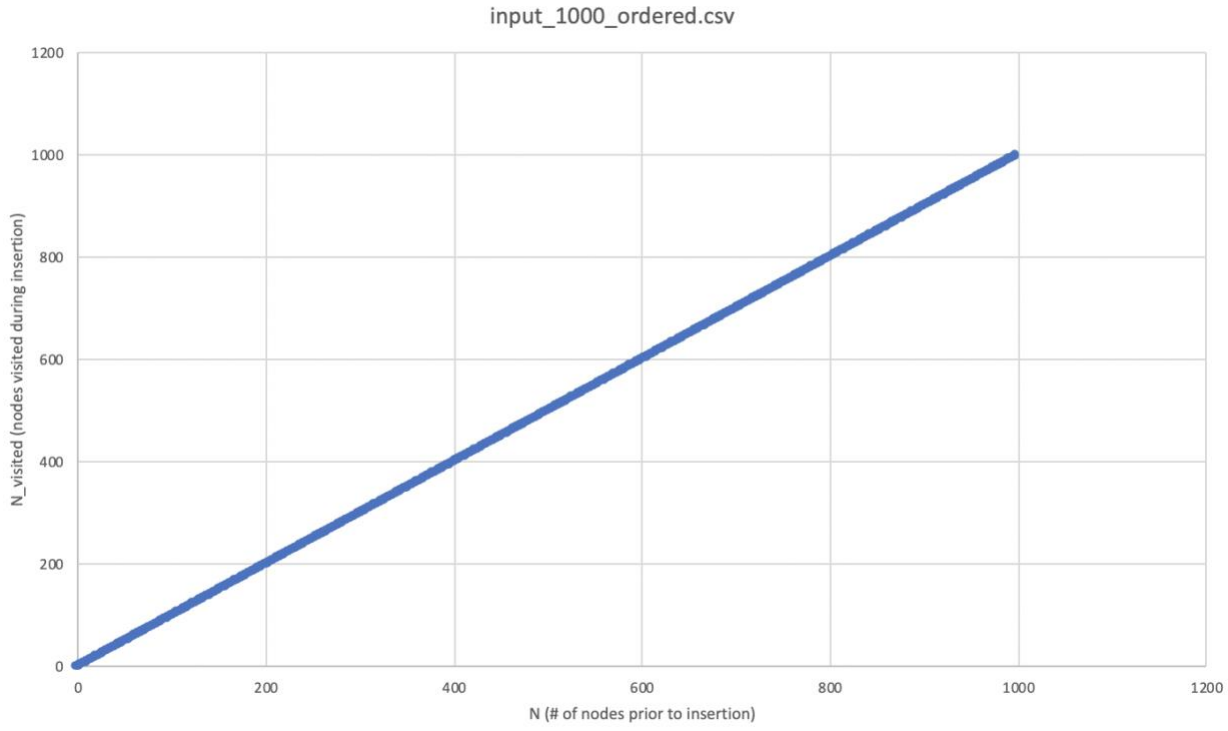
**Number of ordered/random inputs vs. run-time for contains() function:**

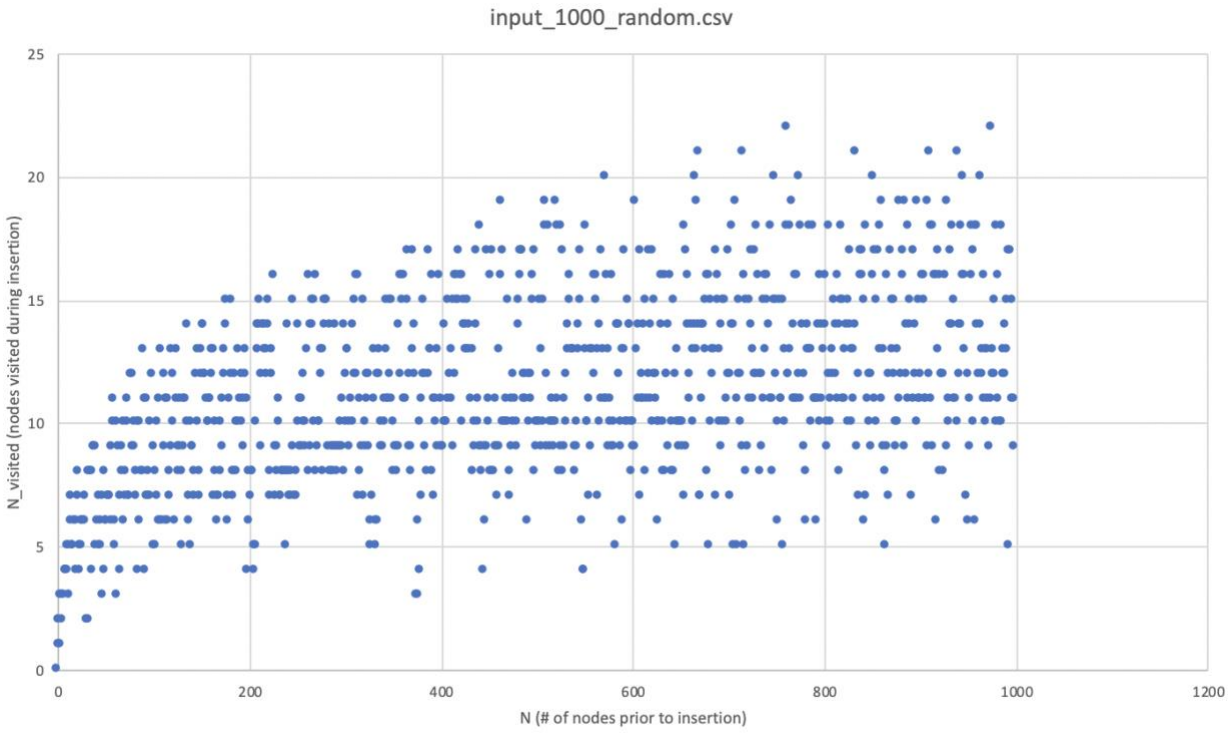
| Dataset | Number of runs | Minimum time (ms) | Maximum time (ms) | Median time (ms) |
|---|---|---|---|---|
| 20 ordered | 100 | 0 | 17 | 2 |
| 20 random | 100 | 0 | 21 | 1 |
| 100 ordered | 100 | 0 | 29 | 6 |
| 100 random | 100 | 0 | 19 | 2 |
| 1000 ordered | 100 | 0 | 276 | 46 |
| 1000 random | 100 | 0 | 34 | 2 |

Examining the table above, there a few noticeable trends that clearly stick out. For example, no matter how many movies are inserted into the binary search tree, the contains() function that searches for a specific movie runs significantly faster when the insertions are in random order compared to alphabetical. This is because when the movies are inserted in alphabetical order, the binary search tree is just a single-path chain whose height is equal to the number of nodes. Therefore, the contains() function visits every node in alphabetical order on its way to the desired node. Therefore, worst case scenario, when the function is searching for a movie whose node contains the movie which is last in alphabetical order, it would have to loop through every single node of the binary search tree, which, as seen by the maximum time for the 1000 ordered input,

can take a much longer time than when the nodes are inserted randomly (276 ms vs. 34 ms). Also, the code that calculates the run-time of the contains() function always outputs 0 ms as its minimum value. This is because when inserting the first node into a BST, it runs so quickly that the time is basically negligible, rounding down to 0. Also, as we can see the median time for ordered inputs increases as the number of inputs increases, but for random inputs the median stays generally the same (1-2 ms). This is connected to the previous explanation above, as the middle entry of the input file is much further down the binary search tree compared to the random input. With the random input, the height of the binary search tree is significantly reduced, as there is a 50-50 chance that the inserted node will go on the left vs. right side of the parent node. This results in the contains() function having to travel much less down the binary search tree thanks to the binary search algorithm implemented.

**N vs. N_visited:**






As seen from the scatter plots above, the amount of nodes visited when inserting a new node varies depending on whether the inputs are ordered or random. For inputs in alphabetical order, the relationship between N and N_visited is perfectly linear. This is because every time a node is inserted, the insert() function must traverse down all the nodes of the binary search tree until it gets to the leaf node, making N = N_visited for every instance of insert() being called. However, when the inputs are randomly ordered, the relationship between N and N_visited seems to be logarithmic from the graph above. As we can see, in general, as N increases, N_visited increases. However, while the growth rate of remained constant with an ordered input, with a random input the growth rate continually decreases as N increases. Attached below is the code for the insert() function:

```
bool Movies::insert(string mName, double mRating){
  if(!root){
    root = new Node(mName, mRating);
    return true;
```

```
  }else{
    return insertHelper(mName, mRating, root);
  }
}

bool Movies::insertHelper(string mName, double mRating, Node* n){
  if(mName == n->name){
    return false;
  }
  if(mName < n->name){
    if(n->left){
      return insertHelper(mName, mRating, n->left);
    }else{
      n->left = new Node(mName, mRating);
      n->left->parent = n;
      return true;
    }
  }else{
    if(n->right){
      return insertHelper(mName, mRating, n->right);
    }else{
      n->right = new Node(mName, mRating);
      n->right->parent = n;
      return true;
    }
  }
}
```

Taking a look at the above code and taking into consideration the worst-case scenario of inserting a node into a tree with N nodes, the insert() function will call the insertHelper() function. For big-O analysis, we must take into consideration the worst case scenario, which, in this case, is if each node inserted is alphabetically further than the previous node. Giving us a binary search tree with each node having only one right child and a height of N. Coincidentally,

this scenario occurs when creating a tree with any of the ordered input files. In this case, the insertHelper() function must return recursively with the next node as a parameter, cycling through the whole binary search tree until it reaches the single leaf node. In this case, insertHelper() would be called N times, giving us a big-O approximation of O(N). These trends are expected when analyzing this code. From intuition, you can assume that when the inputs are ordered, the binary search tree would form in the shape of a single chain, and the function would visit every node in the chain before inserting. As the input becomes randomized, however, the efficiency of the binary search algorithm becomes more useful, halving the number of nodes to search for each iteration, giving us logarithmic growth as N increases.