Algorithmic Analysis of "add()"

Thursday, February 17, 2022 11:16 PM

Breaking down add() and analyzing its operations and those of the method calls it makes

The add() method begins by initializing a variable to hold the largest node found.

```
public void add(Node<T> newNode) {
        Node<T> larger = this.findLarger(newNode);
```

findLarger is a recursive method, having as many layers as the length of the list - 1. So if the list is empty, there are no recursive calls, just the one call to findLarger which returns null. In the best case, the list is empty, and findLarger does one constant time operation, the comparison. If the list is populated, there are length minus one recursive layers, and in each one two comparisons are made, for $2(N-1)$.

```
public Node<T> findLarger(Node<T> newNode) {
        if (this.head != null) {
            return this.findLargerRecursive(this.head, newNode);
        }
        return null;
}

private Node<T> findLargerRecursive(Node<T> listNode, Node<T>
newNode) {
        if (listNode.data.compareTo(newNode.data) > 0) {
            return listNode;
        } else if (listNode.next != null) {
            return this.findLargerRecursive(listNode.next,
newNode);
        } else {
            return null;
        }
}
```

Back to the add() method. If larger is null, ie, there is no node larger than the node being added, we append the node (and append() makes one comparison and three assignments, two if list is empty).

```
public void add(Node<T> newNode) {
        Node<T> larger = this.findLarger(newNode);
        if (larger == null) {
            this.append(newNode);
```

One operation is performed to make the comparison. If it is not null, then we insert the new node before "larger", the first node larger than it, to maintain sorted order.

```
public void add(Node<T> newNode) {
        Node<T> larger = this.findLarger(newNode);
        if (larger == null) {
            this.append(newNode); // O(1)
            } else {
                this.insertBefore(larger, newNode);
            }
```

We next look at the insertBefore method.

```
public void insertBefore(Node<T> currentNode, Node<T> newNode) {
      // If the list is empty, make the node the head and tail
      if (head == null) {
         head = newNode;
         tail = newNode;
      } // If the element is being inserted before the head, make
it the new head
      else if (currentNode.data.compareTo(head.data) == 0) {
         head.previous = newNode;
         newNode.next = head;
         head = newNode;
      } // If the list is going between existing elements:
      else {
         Node<T> predecessor = currentNode.previous;
         newNode.next = currentNode;
         newNode.previous = predecessor;
         currentNode.previous = newNode;
         predecessor.next = newNode;
      }
   }
```

If the list is empty, this method makes 2 assignments. If the new node is being inserted before the head, it makes 3 assignments, and if it is inserted between existing elements, it makes 5 assignments. That is, this method operates in constant time.

Results

Overall, if the list is empty (Best case):

- findLarger performs one CTO (total count 1)
- The result of findLarger is assigned to "larger" (total count 2)
- add() calls append() which performs 2 operations, for 3 operations this step (total count 5)

In the best case, an empty list, 5 constant time operations are performed. In Big O Notation, the algorithmic efficiency is $O(1)$.

If the list has one element, and it's bigger than the new node:

- findLarger performs one CTO (total count 1)
- The result of findLarger is assigned to "larger" (total count 2)
- add() calls append() which performs 3 operations, for 4 operations this step (total count 6)

In this case, 6 constant time operations are performed, for algorithmic efficiency $O(1)$.

If the list is populated (Average and worst case):

- findLarger performs $2(N-1)$ CTOs
- The result of findLarger is assigned to "larger". (total count $2(N-1)+1$
- add() calls insertBefore, which performs two comparisons and then as many as 5 assignments (total count $2(N-1)+1+5)$

In the average and worst case, $2(N-1)+6$ operations are performed for an algorithmic efficiency that can be represented as $O(N)$ in Big O.