COMP2521: Data Structure and Algorithms

Haeohreum Kim (z5480978)

2023 T3

1 Sorting algorithms

Sorting algorithms are a fundamental problem in computer science. At the core, having a sorted list of some objects lets us do very useful things to it - specifically within COMP2521, we see binary search being used often to achieve $O(\log n)$ performance.

1.1 Comparison sorts

Quadratic algorithms are often very easy to implement - but also come at a time cost. Quadratic algorithms generally have nested loops, which are easy to understand, as it compares each node with every single other node n times.

O(n^2) algorithms

1. Selection Sort
   Selection sort works by creating a sorted component and an un-sorted component. The sort works by finding the smallest element left in the unsorted component, and then inserting it in the appropiate place in the sorted component.
   Worst case time complexity: $O(n^2)$
   Best case time complexity: $O(n^2)$
   Stable: No

2. Insertion Sort
   Insertion sort is similar to selection sort, but instead, just chooses the next element to insert into the sorted component. It begins (usually) with the left most element being considered 'sorted'.
   Worst case time complexity: $O(n^2)$
   Best case time complexity: $O(n)$
   Stable: Yes

3. Bubble Sort
   Bubble sort works in a cyclical manner, where it iterates through the array, compares adjacent elements, and swaps them if they are out of order.
   Worst case time complexity: $O(n^2)$
   Best case time complexity: $O(n)$
   Stables: Yes

O(nlogn) algorithms

$O(n \log n)$ sorts are more difficult to implement, but offer superior, noticeable performance in large datasets.

1. Quicksort
   Quicksort is a divide and conquer algorithm, which works by choosing a *pivot*, and then partitoning the elements into sets that are smaller than the element, and bigger than the element. Notably, quicksort can have a worst-case time complexity of $O(n^2)$, when an array is sorted. There are multiple different 'pivot' choices that a programmer can make, begin:

   (a) Naive
   Naive quick sort chooses the first element (or last, or some fixed element) as the pivot. This is the easiest to implement, but also, the worst performing.

   (b) Median-of-three
   Median-of-three quick sort chooses the median of the first, middle and last element as the pivot. This minimises the risk of choosing a worst case pivot, but doesn't entirely remove it. The worst case is still $O(n^2)$, but the average time complexity will be better empirically.

   (c) Randomised
   Randomised quick sort chooses a random pivot. This has the best empirical performance. It is highly unlikely that the worst choice may be chosen.

   Worst case time complexity: $O(n^2)$
   Best case time complexity: $O(nlogn)$
   Stable: No

2. Merge Sort
   Merge Sort is also a divide and conquer algorithm, which works by splitting the array into two halves, and then recursively sorting the two halves, and then merging them together. Merge sort is a stable algorithm, and is often used in practice.
   Worst case time complexity: $O(nlogn)$
   Best case time complexity: $O(nlogn)$
   Stable: Yes

1.2 Non-comparison sorts

Non-comparison sorts don't compare elements directly - but rather use some sort of external sorting method.
Radix sort
Radix sort is a non-comparison sort, which works by sorting the elements by their individual digits. It works by sorting the elements into "buckets" by their least significant digit, and then moving onto the next digit. This is repeated until the most significant digit is reached. (From the last digit to the first digit). An example of radix sort would be:

2 Binary search tree

A binary search tree is a tree with nodes with at most two children, and has the property that the left child is less than the parent. With this, a search can be performed in $O(\log n)$ time.

2.1 Traversal

There are three main types of traversal:

1. Pre-order traversal
   Pre-order traversal can be seen as "stay left" algorithm. It works by visting the current node, and then recursively going down the tree, marking the nodes. The pre-order traversal can be represented as:

   ```
   print(node)
   preorder(node.left)
   preorder(node.right)
   ```

2. In-order traversal
   In-order traversal can be seen as the "left parent right" algorithm - starting from the bottom left, and sweeping upwards, making sure each node is visited. In a BST, it would go bottom left, it's parent, it's right sibling, the parent's parent, and then the parent's sibling subtree.

   ```
   inorder(node.left)
   print(node)
   inorder(node.right)
   ```

3. Post-order traversal
   Post-order traversal can be seen as the "left right parent" algorithm. It trys to stay as low as possible within the tree, but then sweeps up similar to an in-order traversal. In a BST, it would go bottom left, it's sibling, THEN it's parent, then the parent's sibling subtree, and then the parent's parent.

   ```
   postorder(node.left)
   print(node)
   postorder(node.right)
   ```

2.2 Balanced BST's and AVL Trees

2.2.1 Calculating balance

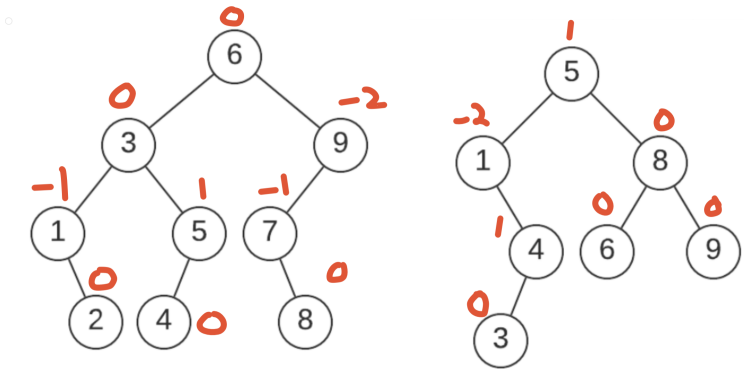
The size balance of a tree, is calculated by the formula:

$$size(left) - size(right)$$

The height balance of a tree is calculated by the height of the left subtree, minus the height of the right subtree. It is calculated by:

$$height(left) - height(right)$$

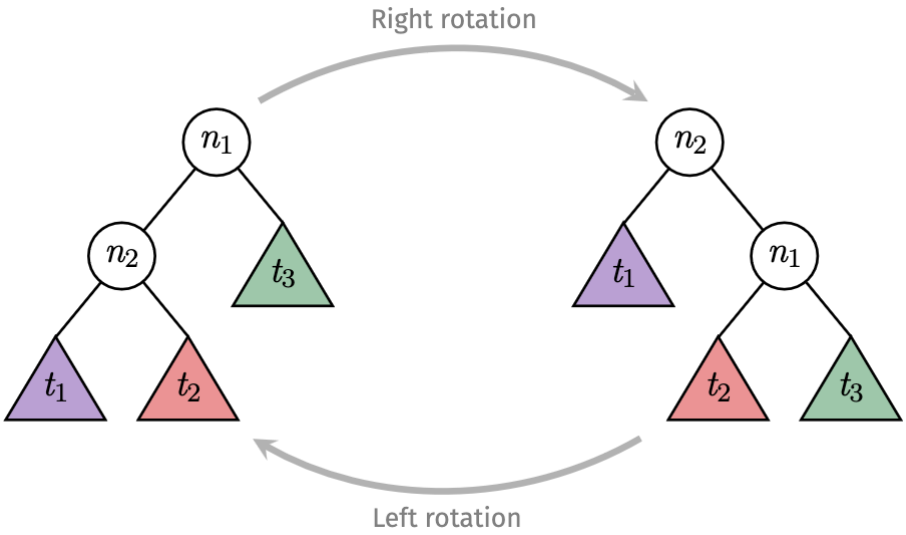
This *balance factor* should be calculated at each node, so that the tree gets balanced from the bottom up.



2.2.2 Rotations

Rotations keep things balanced. If a tree is unbalanced < -1, then it requires a right rotation. If a tree is unbalanced > 1, then it requires a left rotation.

- In different scenarios, you will need a left-right, right-right, right-left, and left-left rotation.
- Right-right and left-left rotations deal with degenerate cases, where the tree is unbalanced in one direction.
- Right-left and left-right rotations deal with the tree being unbalanced in two directions, in a zig-zag pattern.



2.2.3 AVL Trees

AVL trees are constructed by inserting nodes into a BST, and then rotating the tree if it becomes unbalanced. This improves our worst time complexity to $O(\log n)$, as the tree is always balanced.
The steps are:

1. Insert the node into the tree, as you would normally.
2. Calculate the balance factor of the node.
3. If the balance factor is < -1 or > 1, then rotate the tree.
   (a) If the balance factor is < -1 and the value of the inserting node is less than the rotating node's left child, then perform a right-right rotation. Else, perform a right-left rotation.
   (b) If the balance factor is > 1 and the value of the inserting node is greater than the rotating node's right child, then perform a left-left rotation. Else, perform a left-right rotation.
4. Repeat until the tree is balanced.

2.2.4 Global Rebalancing

Global rebalancing works by partitioning the tree into subtrees, and the rebuilding and rebalancing the entire tree as a whole. This is opposed to what an AVL tree does, which is *local rebalancing*, which only rebalances the tree at the node it is inserted at. Global rebalancing ensures a perfect balance.

```
rebalance(tree)
    t = partition(t, size(t) / 2)
    t.left = rebalance(t.left)
    t.right = rebalance(t.right)
```

2.2.5 Root Insertion

Root insertion is a technique used to insert a new node into the BST, and then make it the root node.
Root insertion works as such:

```
insert(tree, value)
    if tree empty
        value <- tree
    else if value < tree->item
        insert(tree.left, value)
        tree = rotateRight(tree)
    else if value > tree->item
        insert(tree.right, value)
        tree = rotateLeft(tree)

    return tree
```

2.2.6 Randomised Insertion

Due to the way AVL trees work, a sorted or partially sorted array of data will result in a worst case performance. This is due to the amount of rotating that is required. Random insertion randomly chooses an element from the to-insert set, and then inserts this into the AVL tree, minimising the risk of having to rebalance a degenerate tree.