Algorithms and Data Structures

Notes on Algorithms, Data Structures and other concepts

ERIC LI

March 2, 2019

Contents

1 Disclaimer

The author(s) of this document assume(s) no responsibility or liability for any errors or omissions in the content of this document. The information contained in this site is provided on an "as is" basis with no guarantees of completeness, accuracy, usefulness or timeliness or of the results obtained from the use of this information.

Information provided in this document has been taken from various sources and is by no means a comprehensive record of the source's views nor of the concepts represented.

Note:
The concepts considered here will be largely captured in the scope of the language C, but can be generally be adapted to other languages, and are conceptually similar.

2 Complexity

There are three types of complexity:

- Big-Omega (Ω) (Best Case)
- Big-Theta (Θ) (Average Case)
- Big-Oh (O) (Worst Case)

Definition
The worse case time complexity of an algorithm is a function that maps input sizes ($\mathbb{N}$) to the running times ($\mathbb{R}^+$).

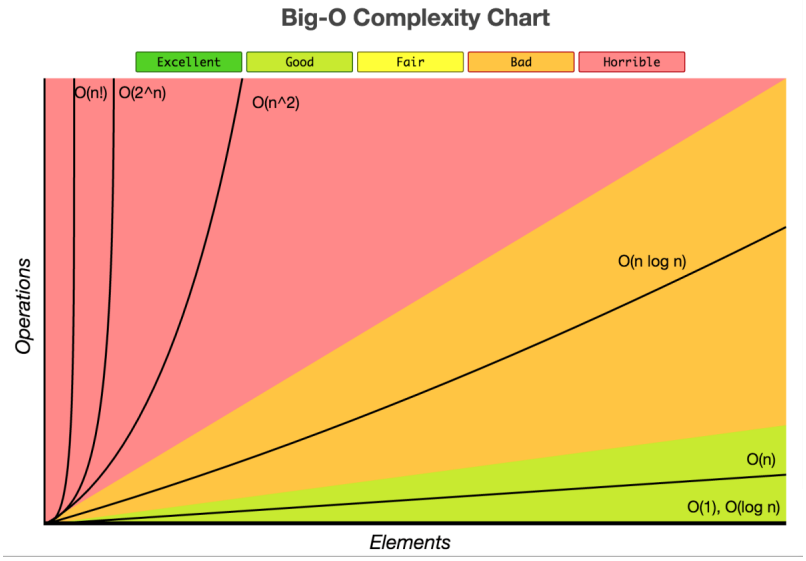

Figure 1: Graph of Time Complexities

Note:

To find the Big-Oh of an algorithm we must determine a function that represents the worst possible running time of the algorithm for various sizes.

2.1 Rules

- We are only interested in the "biggest function"
- We do not care about coefficients

2.2 Examples

- $O(9n^2 + 6n + 5) = O(9n^2) = O(n^2)$
  - We are interested in only the biggest function (largest power) and not coefficients
- $O(n + \log n + 5) = O(n)$
  - We are interested in only the biggest function (largest power)
- $O(n \log) = O(n \log)$
  - Irreducible because it is a product of functions
- $O(7) = O(1)$

2.3 Finding Big-Oh for Code

- Straight-line code is $O(1)$
  - "straight-line code" only runs once regardless of list size
- For if statements, the Big-Oh is the biggest function of any case
- For *loops*, the Big-Oh is the number of times that loop iterates multiplied by the Big-Oh of the inner code

Now the Big-Oh of the algorithm is the **most expensive** (simplified) function generated from the code.

Example:

```
1     int main(){
2         int n = [user input].............\\O(1)
3         int sum = 0;.....................\\O(1)
4         int i,j,k;.......................\\O(1)
5         if (n==7){.......................\\O(1)
6             return 1;....................\\O(1)
7         }
8         else{
9             for(i = 1;i<n;i*=2){.........\\O(log n)
10                for(j=n;j>0;j/=2){.......\\O(log n)
```

```
11                      for(k=j;k<n;k+=2){......\\O(n)
12                          sum +=(i+j+k);......\\O(1)
13                      }
14                  }
15              }
16          }
17      }
```

If we consider the *else* block, we can multiply the worst case complexities and get $O(n(\log n)^2)$ from multiplying the outer for loop, the first nested, and the second nested loop, as well as the straight line code above.

We do not consider the if block since the if block is not the worst case complexity.

3 Linked Lists

Linked lists are helpful in addressing some of the limitations that arrays face, such as inherent dynamic memory allocation and ease of modification. Instead of memory being assigned for use (as in the programming language C) linked lists are built on the premise of allocating memory as needed, and inserting or deleting as required by the user or data set.

Linked Lists can be built with an average complexity of $O(N)$, and searched with an average and worst case complexity of $O(N)$, and bst case complexity of $O(\frac{N}{2})$.

3.1 Structure

A linked list is a linear data structure constructed from building blocks called nodes. Like other data structures, a node contains two items: data, which can be simple, compound, pointer, etc. and a memory address reference (or link to the next node, called a pointer in Figure 1). The address reference will access the next node's memory address in the list, and creates a 'chain-like' relationship which will naturally produce a linear structure. There are two important nodes that are called the 'head node' and the 'tail node', which will influence operations and be explained in depth in the following sections.

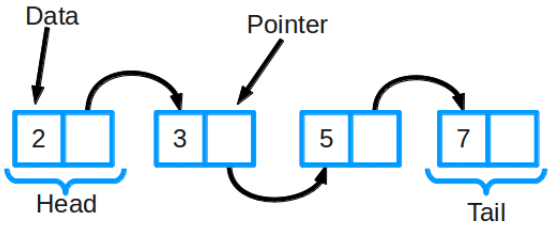

Figure 2: Linked List

3.2 Operations

Linked lists support basic ADT operations of inserting, deleting, and modifying. For operations of inserting and deleting involving the head or tail, implementation requires several changes, but operates in a similar manner nonetheless.

In order to understand operations on linked lists, we must identify what head pointers or head memory address are. Intuitively, the head address's are used to keep track of the linked list and allow for access on all functions. Without the head pointer, we lose the key to the linked list and are unable to both operate, and modify the list.

Therefore, it **imperative** that we return new head address's when modifications to the linked list occur that will affect it.

3.2.1 Insertion

Conceptually, inserting requires a list traversal to the desired point of insertion and memory address manipulation so that the address reference of the nodes link together appropriately.

Inserting in Linked Lists
Inserting inside the list requires a few steps;

- Traversal of the list to the desired location
- Accessing the previous node's reference to the succeeding node
- Setting the memory address of the succeeding node to the address link of the inserted node (linking inserted node to next node)
- Set the memory address of the inserted node to the address link of the previous node (linking inserted node with previous node)
- Return the new head address

Similarly, this can be adapted for **inserting at tail**.

Insertion at Head
Inserting at the head of the list is by far the quickest insertion due to the lack of list traversal and ease of access to the head. Because of this, inserting at the head requires much fewer steps;

- Access the head memory address
- Set the address to the memory link of the inserted node
- Set the head pointer to the inserted node's memory address(thereby setting the inserted node as the head)
- Return the new head address

As we can see, the steps are much more straight-forward and as such, if the items do not require a methodical order or specific location of insertion, inserting at the head is recommended.

3.2.2 Deletion

Conversely, we have deletion which follows a similar format as insertion of nodes;

Deleting in Linked Lists
Deleting in linked lists requires several steps;

- Traversal of the list to the desired node
- Accessing the previous node's reference to the node to delete
- Setting the previous node's reference to the node to delete's reference
- Free the node to delete from memory
- Return the new head address

While this may be conceptually hard to grasp at first, when the implementation is shown, deleting will be much more clear and evident. Intuitively, we are aiming to delete the node, while allowing the previous node to preserve the connection with the succeeding nodes.

Deletion at Head
With a basic understanding of deleting and inserting in a linked list, deleting at the head, is quite rudimentary;

- Free the head node to delete it from memory
- Return the new head address

4 Binary Search Trees

Binary search trees have the average case search complexity of $O(\log N)$, Worst case search complexity of $O(N)$, and can be built with an average complexity of $O(N \log N)$

4.1 Structure and Properties

Binary search trees are an invaluable data structure to learn as they have many applications in computer science. Binary search trees are defined by several things:

- Only having at most two children nodes, appropriately named left child, and right child
- Putting all data with values less than or equal to the root, or parent node, in the left child, and all data with values greater than the root or parent node in the right child
- Levels are defined by the equivalent depths on the tree
  - Children on the same level will have the same distance to the root
- Nodes with no children are appropriately called leaf nodes
- Each level has two times the number of nodes in the last level and in general, the tree will have 2^l $nodes$, $l = levels$ on a complete binary search tree

- Similarly, on level l with k nodes, level 0 to $l-1$, has $k-1$ nodes in total

- The depth of a full binary search tree with N keys is $\lceil \log_2 N \rceil$

4.2 Operations

4.2.1 Tree Traversals

There are three main traversal types which are differentiated by the order of which nodes are visited:

- In-order/Infix Traversal
- Pre-order/Prefix Traversal
- Post-order/Postfix Traversal

In-order/Infix Traversal
For in-order traversal, we visit nodes in three steps:

1. Access and traverse left sub-tree
2. Access root node
3. Access and traverse right sub-tree

The value of the in-order/infix traversal is that we are able to visit the nodes in a numerically/chronologically ordered fashion.

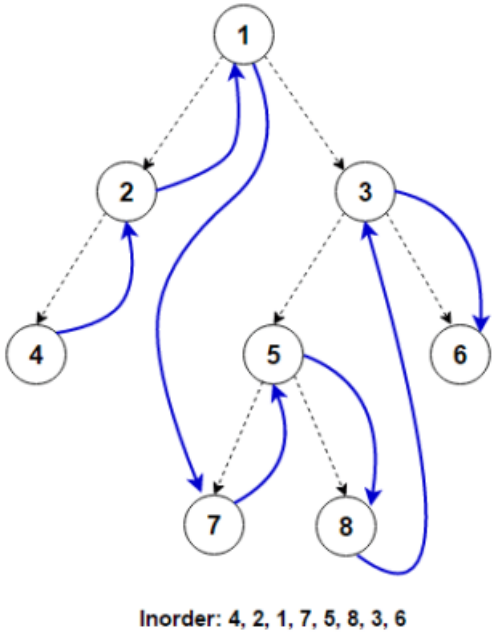

Figure 3: In-order/Infix Traversal

Tip:

Think of in-order traversing as "flagging" the nodes or drawing a line on the **bottom** of each node, and traversing the tree by connecting each "flag" or line from the left side to the right side of the tree.

Taking a look at Figure 1, we can see the visit order at the bottom given the labelled tree, which would produce a sorted set of numbers since we are accessing the smallest values on the left side of the tree and moving towards the larger values.

Pre-order/Prefix Traversal
For pre-order traversal, we visit nodes in three steps:

1. Access root node
2. Access and traverse left sub-tree in pre-order
3. Access and traverse right sub-tree in pre-order

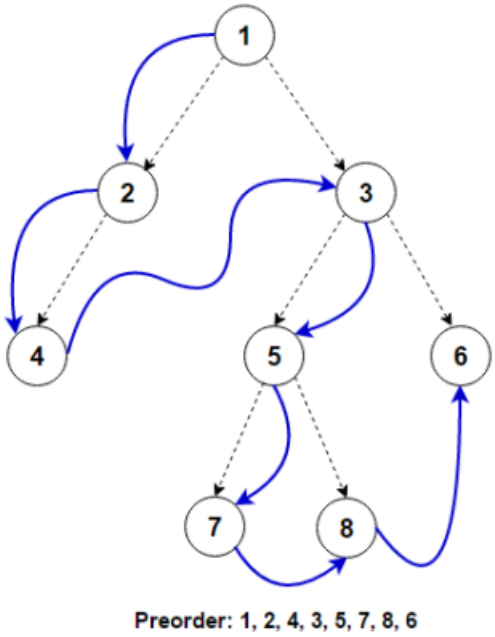

Figure 4: Pre-order/Prefix Traversal

Tip:
Pre-order traversing "flagging" is similar to in-order traversing except we "flag" the **left** side of each node and again, move from the left side of the tree to the right.

Pre-order traversal is the traversal that we use if we would like to copy the binary search tree.

Post-order/Postfix Traversal
For post-order traversal, we visit nodes in three steps:

1. Access and traverse left sub-tree in post-order

2. Access and traverse right sub-tree in post-order
3. Access root node

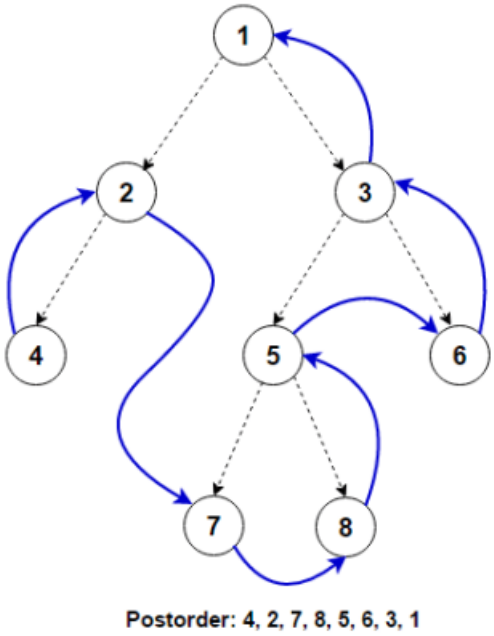

Figure 5: Post-order/Postfix Traversal

Tip:
Post-order traversing "flagging" is again similar to in-order traversing except we "flag" the **right** side of each node and yet again, move from the left side of the tree to the right.

Pre-order traversal is the traversal that we use if we would like to delete the binary search tree.

4.2.2 Insertion

4.2.3 Deletion

There are three cases for deletion

- Deleting a node with no children
- Deleting a node with only one child
- Deleting a node with two children

5 Queues

Queues are built heavily on concepts from linked lists and naturally, follows the same linear data structure. Queues provide a similar environment for "tail insertion" and "head deletion", which are appropriately named "enqueue" and "dequeue" respectively.

6 Additional Notes

The following sections are supplementary and cover topics that may not be considered algorithms and/or data structures but are interesting nonetheless.

6.1 Variadic Functions

Conventional functions, while effective and familiar, can be basic at times, and as such, C allows for variadic functions to handle the various parameter restrictions that conventional functions have.

Variadic functions, as the name suggests, allows you to implement functions that take a variable number of arguments, and does this through a library in C.

To add this library, we must call;

```
1     #include<stdarg.h>
```

6.2 Macros

To use variadic functions, we must use variadic functions macros to access and modify certain areas.

- va_list
- va_start
- va_arg
- va_end
- va_copy

7 Sources

https://www.cs.cmu.edu/ adamchik/15-121/lectures/Linked

"Figure 1"
https://medium.com/@abdurrafeymasood/understanding-time-complexity-and-its-importance-in-technology-8279f72d1c6a

"Figure 2":
https://medium.com/@kenny.lin/singly-linked-lists-5cfdec60bea0

"Figure 3":
https://www.techiedelight.com/inorder-tree-traversal-iterative-recursive/

"Figure 4":
https://www.techiedelight.com/preorder-tree-traversal-iterative-recursive/

"Figure 5":
https://www.techiedelight.com/postorder-tree-traversal-iterative-recursive/