2. Big O Analysis and Linked List

Notes Outline

- Definition of data structure
- What is algorithm?
- Rate of Growth
- Ω, Big O, θ
- Big (O) notation
- Lower bound Ω - Best case
- Tight bound θ - Average case
- Different types of complexity
    - Constant
    - Log n
    - N
    - N long n
    - N^2
    - N^2
    - ...
    - 2^n
- Linear Data Structure
    - Array
    - Linked List - Singly, Doubly, circular

Detailed Notes

-
- 2x^2 + y + 1 -> equation . X and y are place holders (data holder) for data storage.
- In computer science, data types
    - Primitive data types
    - Abstract data type (ADT)

Linear	**Non-Linear (Abstract data type)**
-Stack -Queue -Linked List	-Tree -Graph

- What is algorithm?
    - It is a step by step process to solve a problem.
    - E.g. Go from Sydney CBD to Bondi Junction.
- You need variables. Go operation and give output.

- Identify which algorithm is optimal for solving the particular problem
    - You cannot decide by the followings:
        - Number of statements
        - Execution time (it is based on the system) ram 8 , 16, 32
- You have to check the **rate of growth**.
- As the input is increasing,
    - In what way, the execution time is increasing? **Time Complexity**
    - How about the space? **Space Complexity**

Linear growth	O(n)

- For(int i = 0; i < 10 ; i++){
    - Print(i)
    - }
    - 10 sec, 20 sec, 30 sec, 40 sec
    - It is n linear growth, because it depends on the growth.

Constant Time	Print("Food");

- Increment can be in many ways
    - Linear
    - Non - Linear
- What is the way to measure the complexity?

Best case	Ω	Lower bound
Worst case	Big O	Upper bound
Average case	θ	Tight bound

- 2 < log n < sqrt n < n < n long n < n^2 < n^2 < ..... < 2^n < .....

- Big O notation
    - function(n) <= constant * g(n)
    - E.g.   2n + 3 <= 6n
    - This function is <= constant * g(n) | n should not be 0. n must be >= 1.
    - O(g (n))
    - 2n + 3 <= 2n^2 or 2n^3 or 2n^4 ... All are possible
    - All are upper bound

- For(i = 0; i < n ; i++){
    - Sout(i); // 1, 2, 3, 4, ...
- }
- Therefore it is O(n)

Lower bound Ω - Best case

- 1 < log n < sqrt (n) < n
- 2n + 3 >= 1 * n
- F(n) will depend on g(n)
- Give the closest bound

Tight bound θ - Average case

- 2, 5, 7, 0, 3, 5, 6

- Linear search
    - best case (lower bound) - Order of 1 . Element you are searching is in the index position of the array
    - Worst case (upper bound) - traverse all the elements and get the element you want. Element you are searching is in the later indexes of the array
- O (n)
- For(i = 0 ; i < n ; i++) {
    - Sout(i)
- }

- O(log n)
- For(i = 0 ; i >= 1; i = n/2) {
    - Sout(i);
- }
- Iteration reduced by half, n is reduced by half
- N , n /2, n/4, n/8, ... , n / 2^k = 2
- N = 2^k
- Log(base 2) n = log(base 2) (base 2) = k
- Therefore k = Log(base 2) n

O(sqrt (n))

- For(i = 1; p <= n; i++) {
    - P = p + 1;
- }

I	P	N
1	1	
2	1 + 2 = 3	
3	1 + 2 + 3 = 6	
	If p > n, it will break out from the loop	
	K (k + 1) / 2 K^2 + k > 2n K^2 = n K = sqrt (n)	
1	2	4
2	2 + 3 = 5	4

- For(i = 1; p > n ; i ++) {
    - I = p + i;
- }

I	p	n
1	1	
2	1 + 2	

3	1 + 2 + 3	
4	1 + 2 + 3 + 4	

- 1 + 2 + 3 + ... + k < n
- K (k + 1) / 2 < n
- K^2 + k < 2n
- K^ 2 = n
- K = sqrt (n)

O(n log n)

- For (i = 0; i < n ; i ++) {
    - For (j = n ; j > 1 ; j = j / 2) {
        - Sout(j);
    - }
- }

- **Linear Data Structure**
    - Array - fixed size contagious collection of elements

<table>
<tr><th>Array Limitations</th><th>Linked List</th></tr>
<tr><td>-Fix size
-E.g. 1000 elements size array, you only use 200 elements space.</td><td>-Dynamic
-You can insert the node and insert the data
-Insertion and deletion - same cost O(n) , but they are dynamic
-Structure of Linked List
Data<table><tr><td>Data
A</td><td>Pointer (location for next node location) to B</td></tr></table></td></tr>
<tr><td></td><td>- <table><tr><td>B</td><td>null</td></tr></table></td></tr>
<tr><td></td><td></td></tr>
</table>

- Design own linked List
    - Need a node - should have
        - one data part
        - Node next - for pointer
- Node {
    - Int data;
    - Node next;
- }

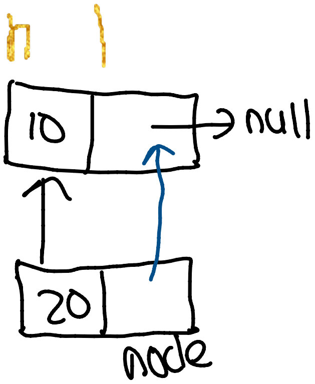
- LinkedList {
    - Node head;
    - LinkedList( ) {
        - Head = null ; // initially it will be null.
    - }

- }

- Create LinkedList methods
    - insertAtStart(int data) {
        - ○ Node node = new Node (data);

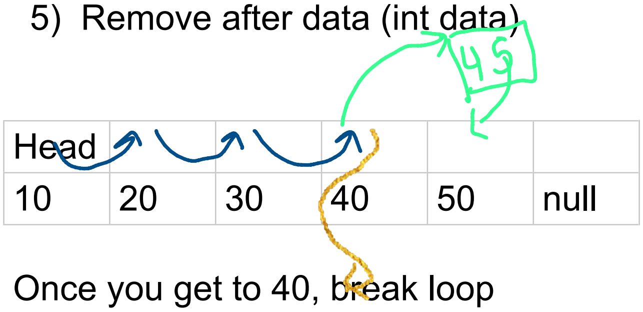
10	null	

        - If (head == null) {
            - ▪ Head = node;
        - }
        - Else {
            - ▪ Node.next = head;
            - ▪ Head = node;
        - }
- The next element become head



Linked List problems (Singly Linked List)

1) Insert at start
2) Insert at last
3) Insert after data (int data1)
4) Insert before data (int data2)
5) Remove after data (int data)



Head					
10	20	30	40	50	null

- Once you get to 40, break loop
- Then add 45
- Temp = Current.next
- Current = added.next

Insert before 40

- If (current.next == 40) {
- You need to reach the second last node before the current.next node
- While (current.next.data != 40) {

Delete first node

10	20	40	50	null

- 20 will become head
- 10 needs to be deleted and will not be referenced as head - JVM will collect

- Head = head.next
- Save memory (java garbage collection)

Delete data from last

10	20	40	50	null

If you reach to second last node and current.next = null, it will be deleted