* **ArrayList Efficiency**
* Access and Search

1. Time complexity of accessing data at a known index – O (1)
2. **VVIP** Data in a sorted ArrayList and you do not know its index – O (log n)
3. Data in an unsorted ArrayList and you do not know its index – O (n)

* Add

1. Time complexity to add data at the front of an ArrayList which is not at capacity – O (n)
2. Time complexity to add data at the back of an ArrayList which is not at capacity – O (1)
3. Time complexity to add data at an arbitrary index within the ArrayList which is not at capacity – O (n)
4. Time complexity to resize an ArrayList – O (n)
5. Time complexity to add data at the front of a full ArrayList – O (n)
6. Time complexity to add data at the back of a full ArrayList – O (n) or O (1)\*
7. Time complexity to add data at an arbitrary index within the full ArrayList – O (n)

* Remove

1. Time complexity to remove data at the front of a full ArrayList – O (n)
2. Time complexity to remove data at the back of a full ArrayList – O (1)
3. Time complexity to remove data at an arbitrary index within a full ArrayList – O (n)

* **Singly LinkedList Efficiency**

Singly Linked List don’t have tail reference.

* Access and Search

1. Time complexity of accessing data at a known position in a Linked List that is not the head – O (n)
2. Time complexity of accessing data at an unknown position in a Linked List that is not the head – O (n)
3. Time complexity of accessing data at the head of a Linked List – O (1)
4. Time complexity of accessing data at the tail of the Linked List without a tail reference – O (n)
5. Time complexity of accessing data at the tail of the Linked List with a tail reference – O (1)
6. Time complexity of finding the size of the Linked List without a size variable – O (n)
7. Time complexity of finding the size of a Linked List with the size information stored at the head – O (1)

* Add

1. Time complexity of adding the data at the head of a Linked List – O (1)
2. Time complexity of adding data at the tail in a Linked List – O (n)
3. Time complexity of adding data at the tail in a Linked list that has a tail reference – O (1)
4. Time complexity of adding data at an arbitrary position in a Linked List that is not the head – O (n)

* Remove

1. Time complexity of removing data at the head in a linked list – O (1)
2. Time complexity of removing data at the tail in a linked list – O (n)
3. Time complexity of removing data at the tail in a linked list that has a tail reference – O (n)
4. Time complexity of removing data at an arbitrary position in a linked list that is not head – O (n)
5. **VVIP** Remove from back in a Circular Linked List – O (n)
6. **VVIP** Add to back in CSLL – O(1)

* Advantages of using LinkedList over ArrayList

1. Adding to front and removing from front is O (1).

* Disadvantages of using LinkedList over ArrayList

1. Accessing a given index is O (n) in Linked List.
2. Removal from back is O (n) in all cases in Linked List.

* **Stacks backed by Linked List**

1. Time complexity of peeking at the top of the stack – O (1)
2. Time complexity of pushing data onto the stack – O (1)
3. Time complexity of popping data off the stack – O (1)
4. Time complexity of determining if the stack is empty – O (1)
5. Time complexity of checking the size of the stack – O (1)

* **Stacks backed by an array**

1. Time complexity of peeking at the top of the stack – O (1)
2. Time complexity of pushing data onto a stack that is not full – O (1)
3. Time complexity of pushing data onto a full stack – O (n) or O (1)\*
4. Time complexity of popping data off the stack – O (1)
5. Time complexity of determining if the stack is empty – O (1)
6. Time complexity of checking the size of the stack – O (1)

* **Queues backed by Linked List**

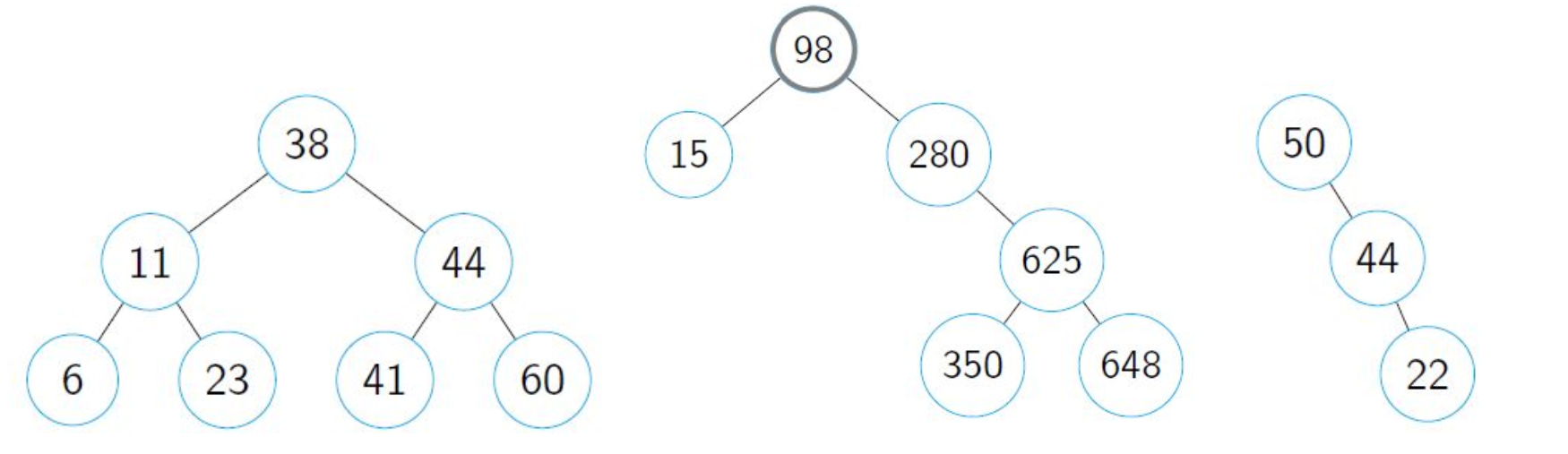
1. Time complexity of looking at front of the queue – O (1)
2. Time complexity to enqueue data into the queue – O (1)
3. Time complexity to dequeue data out of the queue – O (1)
4. Time complexity of determining if the queue is empty – O (1)
5. Time complexity of checking the size of the queue – O (1)

* **Queues backed by an Array**

1. Time complexity of looking at the front of the queue – O (1)
2. Time complexity to enqueue data into the queue that is not full – O (1)
3. Time complexity to enqueue data into a full queue – O (n) or O (1)\*
4. Time complexity to dequeue data out of the full queue – O (1)
5. Time complexity of determining if the queue is empty – O (1)
6. Time complexity of checking the size of the queue – O (1)

AddFirst and AddLast in Array backed Deque is O (1). When resizing is needed, its O (n) or O (1)\*

* **BST Efficiency**

****

1. Number of Nodes in Tree A – 7
2. Number of Levels in Tree A – 3
3. Number of External Nodes in Tree A – 4
4. Time complexity of searching for a data value in tree A – O (log n)
5. Time complexity of adding a data value in tree A – O (log n)
6. Number of Nodes in Tree B – 6
7. Number of Levels in Tree B –4
8. Number of External Nodes in Tree B –3
9. Time complexity of searching for a data value in tree B– O (log n)
10. Time complexity of adding a data value in tree B – O (log n)
11. Number of Nodes in Tree C – 3
12. Number of Levels in Tree C – 3
13. Number of External Nodes in Tree C – 1
14. Time complexity of searching for a data value in tree C – O (n)
15. Time complexity of adding a data value in tree C – O (n)
16. **VVIP** Time complexity to find the height of BST tree- O(n) since height formula is 1 + Math.max(left.height, right.height) so we are basically traversing all nodes.

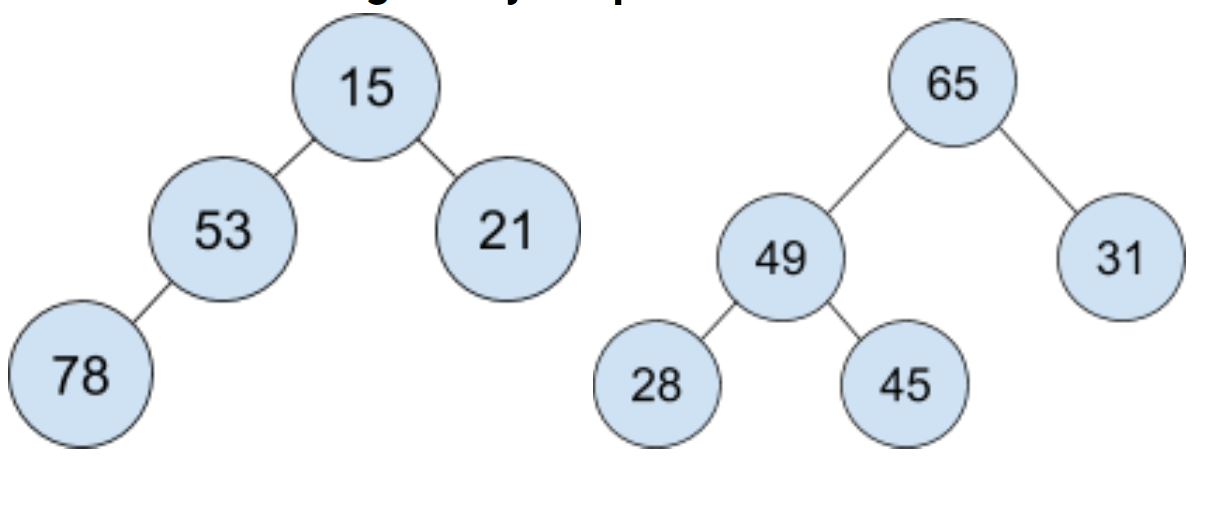
* **Similarities between Binary Trees and Binary Search Trees**

1. Both of them don’t have cycles.
2. Both of them are defined by shape and order.
3. The first node is called root.
4. The nodes linked to parent node are called children
5. Children are labelled left and right.

* **Difference between Binary Trees and Binary Search Trees**

1. The left child data is less than parent data which is less than right child data.

* **Heaps Efficiency**

****

1. Number of levels in Tree A – 3
2. Number of external nodes in Tree A – 2
3. **VVIP** Time complexity of searching a data in Tree A – O (n) since we need to traverse all data. Heaps are not built for searching data.
4. Number of levels in Tree B – 3
5. Number of external nodes in Tree B – 3
6. When adding to a binary heap, the heap has you place the new element at the back of the heap and then do swaps until the element satisfies the order property. What is the time complexity of adding for a data value in Tree A? Tree B? – O (log n)
7. When removing from a binary heap, the heap has you replace the root with the last element from the back of the heap, and then do swaps until the element satisfies the order property. What is the time complexity of removing a data value from Tree A? Tree B? – O(log n)
8. Time complexity for accessing the heap – O (n)
9. Time complexity of Build Heap – O(n)
10. **VVIP** Converting MinHeap to Max Heap – O(n) (Run a build Heap)
11. **VVIP** Runtime of getting 5 smallest elements of Min Heap – O (log n) (need to call remove operation on Heap 5 times)
12. Running BuildHeap on array sorted in ascending order to turn into MinHeap – O (n/2) = O (n)
13. **VVIP** Getting the minimum data from the MaxHeap if the access to array is restricted – O(n log n) because we need to call remove operation n times.

* **HASH MAPS**
* HashMap using External Chaining

1. time complexity of adding to the HashMap when no collision occurs – O (1)
2. **time** complexity of adding to the HashMap when multiple collisions occur – O (n)
3. **time** complexity of adding to the HashMap when every array index has ​n ​items in its linked list – O (capacity \* n) because we need to resize.
4. cost to resize the backing array – O (n)
5. Review questions 1 to 3 for search and remove methods – 3rd one is O(n)
6. **VVIP** Average case of adding to a Hash Map with external chaining backed by a BST instead of SLL– O (1). Worst will be O(n) since BST can be degenerate.

* HashMap using Linear Probing

1. time complexity of adding to the hashmap when no collision occurs and the array is open – O (1)
2. **time** complexity of adding to the hashmap when multiple collisions occur – O (n^2)
3. **Cost** to resize the backing Array – O(n^2)
4. Review questions 1 to 3 for search and remove methods - Same
5. Is the time complexity different if you use quadratic probing – No
6. **VVIP** Average case of adding to a HashMap using Linear Probing where the hash function always return 0 – O(n) since we don’t consider resizing due to average case

* **AVLs**

1. AVL node needs to store Height and Balance Factor as extra properties.
2. Time complexity of storing Height and Balance Factor constant – Yes
3. Will the extra node information impact the overall time complexity – No
4. Cost to perform a rotation and it is always this cost – O (1) and the cost is always same
5. Average & worst time complexity of searching for a data value in an AVL tree – O (log n)
6. Average & worst time complexity of adding a data value to an AVL tree – O (log n)
7. Average & worst time complexity of removing a data value to an AVL tree – O (log n)
8. **Best** time complexity to add, remove or search for a data value in an AVL tree – O (log n)
9. Cost to find the height of an AVL tree – O (1)
10. **VVIP** Worst-case runtime of adding to an AVL tree with 𝑛 data where the balance factor is allowed to range from −𝑛 to 𝑛 – O (n)

* **2- 4 Trees**

1. **Extra** info 2-4 Trees hold – A node needs to store the count of the data in it and its parent.
2. **Time** complexity of storing extra information – O (1)
3. **Will** extra node information impact the overall time complexity – No
4. Cost to perform a promotion and is it always this cost – O (1) and yes
5. Cost to perform a transfer and is it always this cost – O (1) and yes
6. Cost to perform a fusion and is it always this cost – O (1)
7. Cost to perform a fusion resolving underflow and is it always this cost – O (log n)
8. Average & worst time complexity of searching for a data value in an 2-4 tree – O (log n)
9. Average & worst time complexity of adding a data value to an 2-4 tree – O (log n)
10. Average & worst time complexity of removing a data value to an 2-4 tree – O (log n)
11. **Best** time complexity to add, remove or search for a data value in an 2-4 tree – O (log n)
12. **Cost** to find the height of an 2-4 tree – O (log n)
13. **VVIP** Worst case runtime of counting the number of nodes in a 2-4 Tree with 2 data elements – O(n)

* **Skip Lists**

1. What information does a Skiplist node need to store beyond the basic data information? – Has top and bottom references too, beside the previous and next references.
2. Time complexity of storing this extra information constant – Yes
3. Will the extra node information impact the overall time complexity – No
4. **Cost** to perform an insertion on every level of the skiplist and is it always this cost – O (log n) for best and O(n) for worst.
5. Average & worst time complexity of searching for a data value in a SkipList – O (log n)/O (n)
6. Average & worst time complexity of adding a data value to a SkipList –O (log n)/O (n)
7. Average & worst time complexity of removing a data value to a SkipList – O (log n)/O (n)
8. Best case time complexity to add, remove or search for a data value in a SkipList – O (log n)
9. **Cost** to find the height of a Skip List – O (log n)
10. Average case Space complexity of Skip List – O (n) (Data is divided as n, n/2,n/4…)
11. Worst case Space complexity of a Skip List – O (n log n) (Every data is on all levels)
12. **VVIP** Average case of adding to a SkipList with a coin with tails on both sides – O (n) since no data is promoted to upper levels

* **Sorting**

1. Worst case of Cocktail Shaker Sort when the array is in Ascending order until last element – O (n)
2. Runtime of Merge Sort’s Merge procedure if both subarrays have n elements – O (n)
3. Runtime of Merge Sort’s Divide procedure if array have n elements – O(log n)
4. In place sorts require no more than O(1) extra memory.
5. **VVIP** Worst case scenario for Quick Select if 1st element is pivot – Sorted or Reverse sorted array

* **Pattern Matching**

1. Best case time complexity to find the single occurrence in Brute Force – O(m)
2. Best case time complexity to find all occurrences in Brute Force when no match occurs– O(n)
3. Best case time complexity to find all occurrences in Brute Force when multiple matches occur – O (mn)
4. Average case time complexity of Brute Force – O (mn)
5. Worst case time complexity of Brute Force – O (mn)
6. Best case time complexity to find the single occurrence in Boyer Moore– O(m)
7. Best case time complexity to find all occurrences in Boyer Moore when no matches occur– O((n/m) + m)
8. Worst case time complexity of Boyer Moore – O(mn)
9. Boyer Moore works best for Large Alphabet size.
10. Best case time complexity to find the single occurrence in KMP – O (m)
11. Best case time complexity to find all occurrences in KMP when no matches occur– O (m + n)
12. Worst case time complexity of KMP – O (m + n)
13. Best case time complexity to find the single occurrence in Rabin Karp – O (m)
14. Best case time complexity to find all occurrences in Rabin Karp – O (m + n)
15. Worst case time complexity of Rabin Karp – O (mn)
16. **VVIP** If I know the alphabet only has 5 characters, should I use Boyer Moore or KMP – KMP
17. **VVIP** If I know the alphabet has alphanumeric characters, should I use KMP or Boyer Moore – BM
18. **VVIP** How many time would we shift the pattern if no character in the text exists in the pattern for BM, as a function of n and m – (n/m)

* **Graphs**

1. Time complexity of DFS – O (|E| + |V|)
2. Time complexity of BFS – O (|E| + |V|)
3. Time complexity of Dijkstra’s – O ((|E| + |V|) log|V|)
4. Time complexity of Prim’s - O ((|E| + |V|) log|V|)
5. Time complexity of Kruskal’s – O (|E| log|V|)
6. Space complexity of Adjacency List – O (|V| + |E|)
7. Space complexity of Adjacency Matrix – O (|V|^2)
8. Space complexity of Edge List – O (|V| + |E|)
9. Time complexity to find incident edges of vertex u from an Adjacency List– O (Deg(|u|))
10. Time complexity to find incident edges of vertex u from an Adjacency Matrix– O (|v|)
11. Time complexity to find incident edges of vertex u from an Edge List– O (|E|)
12. Time complexity to find whether vertices u and w are adjacent from an Adjacency List – O (min ((deg(|u|), (deg(|W|)))
13. Time complexity to find whether vertices u and w are adjacent from an Adjacency Matrix – O (1)
14. Time complexity to find whether vertices u and w are adjacent from an Edge List – O (|E|)
15. Time complexity to insert a vertex in an Adjacency List – O (1)
16. Time complexity to insert a vertex in an Adjacency Matrix – O (|V|^2) or O (|V|.|E|)
17. Time complexity to insert a vertex in an Edge List – O (1)
18. Time complexity to insert an edge in an Adjacency List – O (1)
19. Time complexity to insert an edge in an Adjacency Matrix – O (1)
20. Time complexity to insert an edge in an Edge List – O (1)
21. Time complexity to remove a vertex from an Adjacency List – O (deg(|V|)
22. Time complexity to remove a vertex from an Adjacency Matrix – O (|V|^2) or O(|V|.|E|)
23. Time complexity to remove a vertex from an Edge List – O(|E|)
24. Time complexity to remove an edge from an Adjacency List – O (1)
25. Time complexity to remove an edge from an Adjacency Matrix – O (1)
26. Time complexity to remove an edge from an Edge List - O (1)

* **Lowest Common Subsequence**

1. Runtime of LCS in all cases – O (mn) where m and n are length of Strings

* **Efficiencies asked in previous exams (All are Worst case)**

1. Iterating over a Linked List using an Iterator - O(n)
2. Removing from the back of a Singly Linked List with a tail pointer - O(n)
3. Adding to the back of an ArrayList without a size variable - O(n)
4. Adding to the end of a linked-list-backed deque - O(1)
5. Accessing the data at index 2 of a Singly-Linked List of size at least 4 - O(1)
6. Average case of adding to a skip list with a coin with tails on both sides (recall: tails does not promotes a node). – O(n)
7. Running BuildHeap on an array sorted in ascending order to turn it into a Min Heap. – O(n)
8. Average case of adding to a HashMap using linear probing where the hash function always returns 0. – O(n)
9. Average case of remove() in a Max Heap. – O(log n)
10. Runtime of finding the height of the root node in an AVL tree. – O(1)
11. Difference between max and min data of BST – O(n)
12. Average case of adding to a skip list with a coin with heads on both sides (recall: heads promotes a node). – O(n)
13. Make sorted SLL from unordered stack – O(n^2) because removal of a data from stack and add to SLL at appropriate order is O(n) and we are doing this for n data elements.
14. Transferring queue to SLL, add to back and no tail – O(n^2) because for each data we are removing from queue and adding to SLL with no tail which is O(n) and we are doing it for n data elements.
15. Remove 5th node of SLL – O(1)
16. Finding maximum in balanced BST – O(log n)
17. Adding 4 to MaxHeap whose smallest data is 10 – O(1)
18. Remove most recently added element from queue – O(n) because we are removing from back of SLL
19. Access 50th element of DLL – O(1)
20. Build BST from sorted array, add in order – O(n^2) because adding to a degenerate tree is O(n) and we are doing it for n data elements
21. Add to back of CSLL – O(1)
22. Finding the maximum element in MinHeap – O(n) since we need to traverse all the data in the array
23. Remove from back of SLL – O(n)
24. Add to back of ArrayList – O(n) since its worst case and so we need to consider resizing
25. MergeSort from Preorder BST traversal array – O(n log n)
26. Bubble sort on a sorted array – O(n)
27. Insertion sort on a sorted array – O(n)
28. Selection sort on a sorted array – O(n^2)
29. QuickSort when max is always pivot – O(n^2) because max element falls under worst case
30. LSD Radix Sort on 2 sets of n data, with k and j max digits – O(kn+jn)
31. Creating a heap with the add method – O(n logn)
32. Repeated downheaps on first half of data – O(n) since this is technically Build Heap
33. **VVIP** Adding to a HashMap with BST chain, no resize – O(n) since BST can be degenerate
34. Add to HashMap with Linear Probing, no collisions – O(1)
35. Remove all leaves from 3rd level of BST – O(1)
36. Create an AVL by adding each data in a queue – O(n logn) since a single remove of AVL is O(log n) and we need to remove n data. Add to queue is O(1) so don’t need to consider that.
37. Add to a SkipList with all data on all levels – Searching for place is O(n) and then add is O(log n). So time complexity is O(n + logn) which is O(n)
38. Calling add 3 times in a 2-4 tree – O(log n)
39. What is the worst case Big O of finding the minimum of an AVL tree having n data values – O(log n)
40. What is the worst case Big O of searching a hash table with n key value pairs for a value if you do not know the index – O(n)
41. What is the Big O for finding the value given a key in a hash table that uses external chaining with n key value pairs? The keys have a good hash function and the backing array is reasonably sized – O(1)