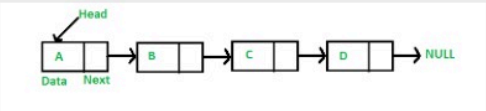
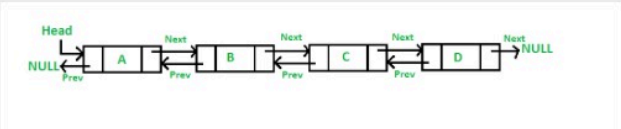
Tabular Difference

CHARACTERISTIC	AWT	SWING	JavaFX
DEPENDENCY	Platform Dependent	Platform Independent	Platform Independent
WEIGHT OF COMPONENT	Heavyweight	Lightweight	Lightweight
CURRENTLY IN USE	Discarded	Use It In A Few Places.	Currently In Use
PLUGGABLE	Does Not Support A Pluggable Look And Feel	Support Pluggable Look And Feel. Components Look The Same On All Systems	Support Pluggable Look And Feel. Components Look The Same On All Systems.
MVC	Does Not Follow MVC	Follow MVC	Follow MVC.
NO. OF COMPONENTS	Less	More Than AWT	More Than AWT But Less Than SWING.
PACKAGE	Java.Awt.Package	Javax.Swing	Javafx.Util
RELEASED	1995	1997	2008

J

ava 中，可以使用访问控制符来保护对类、变量、方法和构造方法的访问。Java 支持 4 种不同的访问权限。

- **default** (即默认，什么也不写）: 在同一包内可见，不使用任何修饰符。使用对象：类、接口、变量、方法。
- **private** : 在同一类内可见。使用对象：变量、方法。 **注意：不能修饰类（外部类）**
- **public** : 对所有类可见。使用对象：类、接口、变量、方法
- **protected** : 对同一包内的类和所有子类可见。使用对象：变量、方法。 **注意：不能修饰类（外部类）**。

SLL nodes contains 2 field -data field and next link field.	DLL nodes contains 3 fields -data field, a previous link field and a next link field.
	
In SLL, the traversal can be done using the next node link only. Thus traversal is possible in one direction only.	In DLL, the traversal can be done using the previous node link or the next node link. Thus traversal is possible in both directions (forward and backward).
The SLL occupies less memory than DLL as it has only 2 fields.	The DLL occupies more memory than SLL as it has 3 fields.
Complexity of insertion and deletion at a given position is O(n).	Complexity of insertion and deletion at a given position is O(n / 2) = O(n) because traversal can be made from start or from the end.
Complexity of deletion with a given node is O(n), because the previous node needs to be known, and traversal takes O(n)	Complexity of deletion with a given node is O(1) because the previous node can be accessed easily
We mostly prefer to use singly linked list for the execution of stacks.	We can use a doubly linked list to execute heaps and stacks, binary trees.
When we do not need to perform any searching operation and we want to save memory, we prefer a singly linked list.	In case of better implementation, while searching, we prefer to use doubly linked list.
A singly linked list consumes less memory as compared to the doubly linked list.	The doubly linked list consumes more memory as compared to the singly l[illegible]d list.

Stable and Unstable Algorithms

This is a rather niche use, and only makes an actual difference in certain types of data. However, it remains an important requirement that is needed for these certain scenarios.

Sorting Algorithm	Stable Sort?
Bubble Sort	Yes
Insertion Sort	Yes
Selection Sort	No
Quick Sort	No
Merge Sort	Yes
Heap Sort	No

Insertion sort takes linear time when input array is sorted or almost sorted (maximum 1 or 2 elements are misplaced).

All other sorting algorithms mentioned above will take more than lienear time in their typical implementation.

Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of O(nLogn). The worst case is possible in randomized version also, but worst case doesn't occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice. **Quick Sort is also a cache friendly sorting algorithm as it has good locality of reference when used for arrays.** Quick Sort is also tail recursive, therefore tail call optimizations is done.

(a) What does it mean to sort "in place", and why would we want this?

In general, we consider a sorting operation to be done in place if it does not require significant extra space. "Significant extra space" is often defined as linear or greater, with respect to the number of elements we are sorting. For example, if our sorting algorithm requires making a whole new array and copying elements over to it, then it is NOT in place because we had to allocate significant space for this new array. Some algorithms that can be implemented in place are quick `sort`, selection sort, insertion sort, and heap sort. Mergesort technically can be implemented in place, but it's rather complex. Doing operations in place is beneficial because we typically want to use as little memory/computer resources as possible. Though in this class, we focus on time efficiency, space efficiency is important too in the real world!

In Place: Bubble sort, Selection Sort, Insertion Sort, Heapsort.
Not In-Place: Merge Sort. Note that merge sort requires O(n) extra space.
QuickSort uses extra space for recursive function calls. It is called in-place according to broad definition as extra space required is not used to manipulate input, but only for recursive calls.

(b) What does it mean for a sort to be "stable"?

Which sorting algorithms that we have seen are stable? If given a list with equivalent elements (according to whatever metric we're using to compare items), a stable sorting algorithm will leave those elements in the same order as they were originally. Some stable sorts we've seen are insertion sort and merge sort.

(c) Which algorithm would run the fastest on an already sorted list?

Insertion sort would run the fastest on an already sorted list, because there would be no inversions! Then at every step, insertion sort would see that there is no inversion and simply move on to the next element. After going through the whole list like this, insertion sort would terminate, resulting in linear time!

(d) Given any list, what is the ideal pivot for quicksort?

The ideal pivot will be the median, or the element that WOULD be at the exact halfway point if the list were to be sorted. This is because it will split the list exactly in half after partitioning around it. When we are able to break our subproblem perfectly in half at every

time step, it gives is the recursive tree height of log(n). With n work done at each level, we achieve θ(n log n) runtime in this best case where we pick the median at each step. Note that the worst case pivot choice is choosing the minimum or maximum element (apply the same logic we've used above to see why!), in which we'd have θ(n 2)! Then technically, the upper bound or Big O limit on quicksort is O(n 2). On average though, with random pivot selection, we often see behavior that closely resembles our best case. (e) So far, in class, we've mostly applied our sorts to lists of numbers. In practice, how would we typically make sure our sorts can be applied to other types? In practice, if we want our items to be sortable by a comparison sort, we make sure they implement the Comparable interface! This involves defining a compareTo() method, such that a sorting algorithm has a way to compare our elements, just as naturally as it can compare numbers. This is only relevant for comparison sorts (all of the sorts in this discussion), but not counting sorts (e.g. radix sorts).

quicksort has a worst case runtime of Θ(N2), if the array is partitioned very unevenly at each iteration.

Give a 5 integer array that elicits the worst case runtime for insertion sort. A simple example is: 5 4 3 2 1. Any 5 integer array in descending order would work.
mergesort has Θ(NlogN)worst case runtime versus quicksort's Θ(N2). Mergesort is stable, whereas quicksort typically isn't. Mergesort can be highly parallelized because as we saw in the first problem the left and right sides don't interact until the end. Mergesort is also preferred for sorting a linked list.

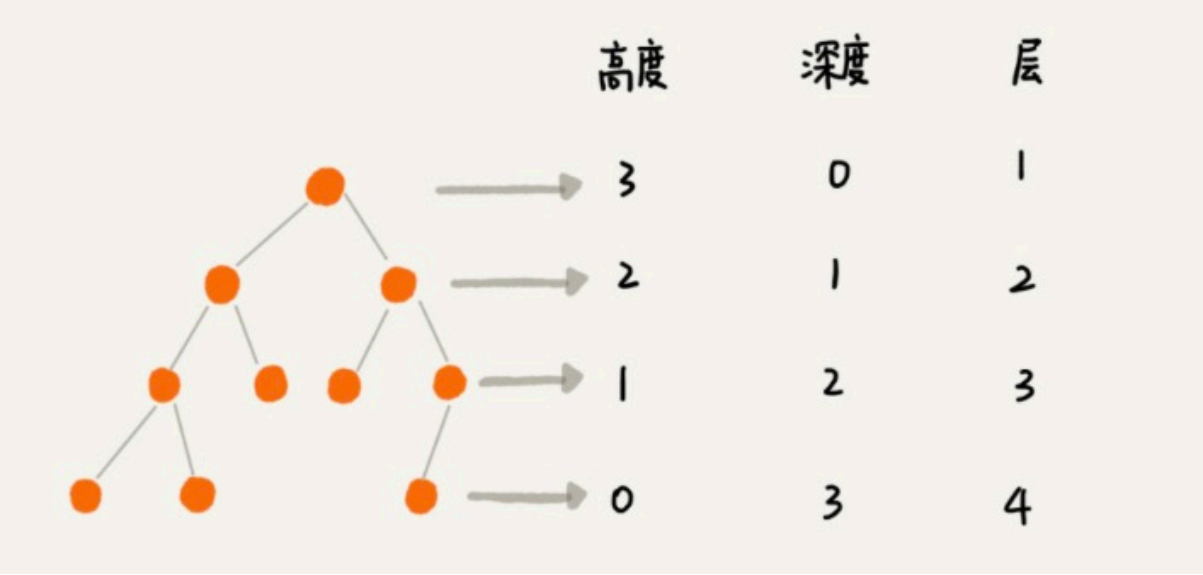
1. Quick Sort –
   This sorting algorithm is also based on Divide and Conquer algorithm. It picks an element as pivot and partitions the given list around the picked pivot. After partitioning the list on the basis of the pivot element, the Quick is again applied recursively to two sublists i.e., sublist to the left of the pivot element and sublist to the right of the pivot element.
   We can use Quick Sort as per below constraints :
   - Quick sort is fastest, but it is not always **O(N*log N)**, as there are worst cases where it becomes **$O(N^2)$**.
   - Quicksort is probably more effective for datasets that fit in memory. For larger data sets it proves to be inefficient so algorithms like merge sort are preferred in that case.
   - Quick Sort in is an in-place sort (i.e. it doesn't require any extra storage) so it is appropriate to use it for arrays.

1. Merge Sort –
   This sorting algorithm is based on Divide and Conquer algorithm. It divides input array into two halves, calls itself for the two halves, and then merges the two sorted halves.
   The **merge()** function is used for merging two halves. The **merge(arr, l, m, r)** is a key process that assumes that **arr[l..m]**and **arr[m+1..r]** are sorted and merges the two sorted sub-arrays into one.
   We can use Merge Sort as per below constraints :

   - Merge sort is used when the data structure doesn't support random access since it works with pure sequential access that is forward iterators, rather than random access iterators.
   - It is widely used for external sorting, where random access can be very, very expensive compared to sequential access.
   - It is used where it is known that the data is similar data.
   - Merge sort is fast in the case of a linked list.
   - It is used in the case of a linked list as in linked list for accessing any data at some index we need to traverse from the head to that index and merge sort accesses data sequentially and the need of random access is low.
   - The main advantage of the merge sort is its stability, the elements compared equally retain their original order.

1. Selection Sort –
   This sorting algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array, the subarray which is already sorted and remaining subarray which is unsorted. In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.
   We can use Selection Sort as per below constraints :

   - When the list is small. As the time complexity of selection sort is **$O(N^2)$** which makes it inefficient for a large list.
   - When memory space is limited because it makes the minimum possible number of swaps during sorting.



经过前人的总结，二叉树具有以下几个性质：

1. 二叉树中，第 i 层最多有 2^{i-1} 个结点。
2. 如果二叉树的深度为 K，那么此二叉树最多有 2^K-1 个结点。
3. 二叉树中，终端结点数（叶子结点数）为 n_0，度为 2 的结点数为 n_2，则 $n_0=n_2+1$。

满二叉树

如果二叉树中除了叶子结点，每个结点的度都为 2，则此二叉树称为满二叉树。

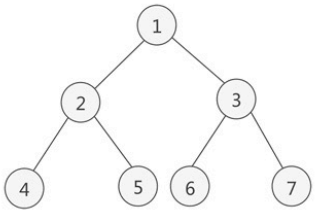


图 2 满二叉树示意图

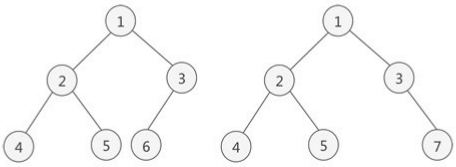
如图 2 所示就是一棵满二叉树。

满二叉树除了满足普通二叉树的性质，还具有以下性质：

1. 满二叉树中第 i 层的节点数为 2^{n-1} 个。
2. 深度为 k 的满二叉树必有 2^{K}-1 个节点，叶子数为 2^{k-1}。
3. 满二叉树中不存在度为 1 的节点，每一个分支点中都两棵深度相同的子树，且叶子节点都在最底层。
4. 具有 n 个节点的满二叉树的深度为 $\log_2(n+1)$。

完全二叉树

如果二叉树中除去最后一层节点为满二叉树，且最后一层的结点依次从左到右分布，则此二叉树被称为完全二叉树。



a) 完全二叉树　　b) 非完全二叉树

图 3 完全二叉树示意图

如图 3a) 所示是一棵完全二叉树，图 3b) 由于最后一层的节点没有按照从左向右分布，因此只能算作是普通的二叉树。

完全二叉树除了具有普通二叉树的性质，它自身也具有一些独特的性质，比如说，n 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$。

> $\lfloor \log_2 n \rfloor$ 表示取小于 $\log_2 n$ 的最大整数。例如，$\lfloor \log_2 4 \rfloor = 2$，而 $\lfloor \log_2 5 \rfloor$ 结果也是 2。

对于任意一个完全二叉树来说，如果将含有的结点按照层次从左到右依次标号（如图 3a)），对于任意一个结点 i，完全二叉树还有以下几个结论成立：

1. 当 i>1 时，父亲结点为结点 [i/2] 。（i=1 时，表示的是根结点，无父亲结点）
2. 如果 2*i>n（总结点的个数），则结点 i 肯定没有左孩子（为叶子结点）；否则其左孩子是结点 2*i 。
3. 如果 2*i+1>n，则结点 i 肯定没有右孩子；否则右孩子是结点 2*i+1 。

Quicksort complexity when all the elements are same?

This depends on the implementation of Quicksort. The traditional implementation which partitions into 2 (< and >=) sections will have $O(n*n)$ on identical input. While no *swaps* will necessarily occur, it will cause n recursive calls to be made - each of which need to make a comparison with the pivot and n-recursionDepth elements. i.e. $O(n*n)$ comparisons need to be made

Assuming all the keys are distinct, how many children can the smallest or maximum element in a binary search tree have?
At most 1

The longest simple path from node *x* in a red-black tree to a descendant leaf has length L. The shortest simple path from node *x* to a descendant leaf is length S.

Which of the answers below are always correct?

L<=2S

A full binary tree: all internal nodes have two children. As such, each node will have either two children (internal nodes), or no children at all (external or leaf nodes). In another word, each node is either a leaf or has degree exactly 2. There are no degree 1 nodes.

A complete binary tree: a binary tree in which all leaves in the tree have the same depth and all internal nodes have degree k.

In this case a full binary tree can be a complete binary tree, but not all full binary trees are a complete binary tree.

A complete binary tree however is also a full binary tree, as all internal nodes have a degree of exactly 2.

Operations	Adjacency Matrix	Adjacency List
Storage Space	This representation makes use of VxV matrix, so space required in worst case is **$O(|V|^2)$**.	In this representation, for every vertex we store its neighbours. In the worst case, if a graph is connected O(V) is required for a vertex and O(E) is required for storing neighbours corresponding to every vertex .Thus, overall space complexity is O(|V|+|E|).
Adding a vertex	In order to add a new vertex to VxV matrix the storage must be increases to $(|V|+1)^2$. To achieve this we need to copy the whole matrix. Therefore the complexity is **$O(|V|^2)$**.	There are two pointers in adjacency list first points to the front node and the other one points to the rear node.Thus insertion of a vertex can be done directly in **O(1) time.**
Adding an edge	To add an edge say from i to j, matrix[i][j] = 1 which requires **O(1)** time.	Similar to insertion of vertex here also two pointers are used pointing to the rear and front of the list. Thus, an edge can be inserted in **O(1)**time.
Removing a vertex	In order to remove a vertex from V*V matrix the storage must be decreased to $|V|^2$ from $(|V|+1)^2$. To achieve this we need to copy the whole matrix. Therefore the complexity is **$O(|V|^2)$**.	In order to remove a vertex, we need to search for the vertex which will require O(|V|) time in worst case, after this we need to traverse the edges and in worst case it will require O(|E|) time.Hence, total time complexity is **O(|V|+|E|)**.
Removing an edge	To remove an edge say from i to j, matrix[i][j] = 0 which requires **O(1)**time.	To remove an edge traversing through the edges is required and in worst case we need to traverse through all the edges.Thus, the time complexity is **O(|E|)**.
Querying	In order to find for an existing edge the content of matrix	In an adjacency list every vertex is associated with a list of

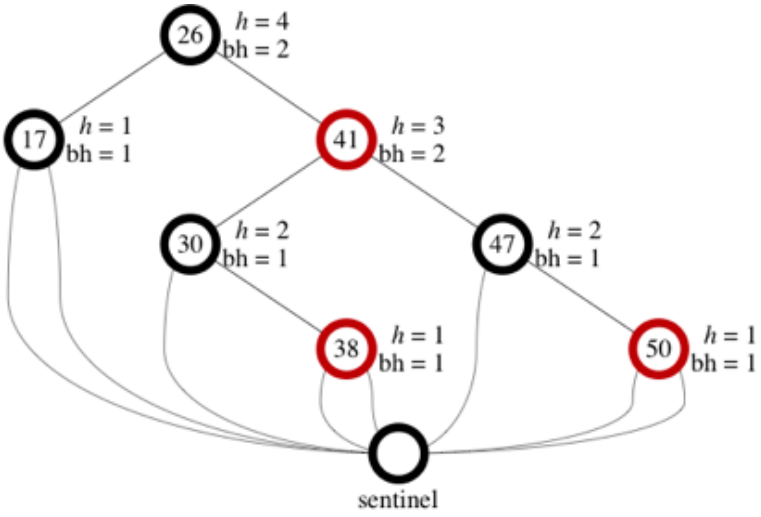
Operations	Adjacency Matrix	Adjacency List
	needs to be checked. Given two vertices say i and j matrix[i][j] can be checked in **O(1)** time.	adjacent vertices. For a given graph, in order to check for an edge we need to check for vertices adjacent to given vertex. A vertex can have at most O(|V|) neighbours and in worst can we would have to check for every adjacent vertex. Therefore, time complexity is **O(|V|)** .

红黑树

https://www.cs.dartmouth.edu/~thc/cs10/lectures/0519/0519.html

A red-black tree obeys the five **red-black properties**:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (the sentinel) is black.
4. If a node is red, then both its children are black. (Hence there can be no two red nodes in a row on a simple path from the root to a leaf.)
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.



by property 2, any node with height h has black-height at least $h/2$. (At most half the nodes on a path to a leaf are red, and so at least half are black.)

A red-black tree with n internal nodes has height at most $2 \lg (n + 1)$.

First observe that any full binary tree has exactly $2n - 12n-1$ nodes.

Push LIFO

QUEUE FIFO