Graph

Tree:

-> Tree is a **connected graph** with V vertices and $E = V - 1$ edges

-> Acyclic: 没有环

-> 每一对节点仅有唯一的一条路径 path

DAG: 有向无环图

Entries of *edge list*: number of edges

Number of *filled cells* of *adjacency matrix*:

$$N_{\text{vertices}}^2 - N_{\text{edges}} \times 2$$

Entries of *adjacency list*: 每个节点的（哈希）表都存储着和其相邻的节点

Suitable DS for different situation:

-> Adjacency Matrix, AM:

-> -> 适合稠密图，即边的数量约等于节点数量的平方

-> ->适合频繁查询两个节点间是否存在边的情况

-> -> Space complexity $O(n^2)$

-> Adjacency List, AL:

-> ->适合稀松图，即边的数量远小于节点数量的平方

-> ->适合频繁查询节点的相邻节点的情况

-> ->适合有限内存的情况

-> -> Space complexity $\approx 2e$

-> Edge Lists, EL:

-> -> 适合较小数量的边

-> -> 适合有限内存的情况

-> -> 适合简单的图

-> -> Space complexity e

-> ->适合频繁检索（排序）所有的边的情况

BFS & DFS

DFS:

1. 从一个选定的源节点开始，将其标记为"已访问"，并将其放入栈中。
2. 取栈顶元素为当前节点，探索当前节点的一个未访问的邻居节点。
3. 将新发现的节点标记为"已访问"并放入栈中。
4. 如果当前节点没有未访问的邻居节点，则将它从栈中弹出（回溯）。
   - 这意味着如果重复步骤，则在步骤 2 选定的节点为该节点的上一个节点

BFS:

1. 初始化：首先将根节点放入队列中。

2. 循环遍历：只要队列不为空，就重复以下步骤：

2.1 从队列的前端取出一个节点。

2.2 检查它是否为目标。如果找到目标，则搜索结束。

2.3 如果它不是目标，则将该节点的所有未访问的邻接点加入队列，并标记这些邻接点为已访问。

3 **访问节点**：对于队列中的每个节点，访问该节点，并检查它是否是目标节点。如果是，则结束搜索并返回结果。如果不是，则将其所有未被访问过的邻居节点加入队列。

4 **标记已访问**：在加入队列的同时，应该将节点标记为已访问，以防止将节点重复加入队列。

Print the traversal path

-> For DFS, print each node when a node is marked as visited

->-> i.e., print node once explore to it

-> For BFS, print each node when dequeue a node

->-> i.e., print node once it is get out form the queue to explore

Bipartite Graph 二分图: 将图分为两个集合，只有集合之间存在边，集合内部没有边

Simple Path 简单路径: 即路径上没有重复的节点

Edges that make up the spanning tree 构成生成树的边:

-> 从源点开始 DFS/BFS，遍历节点经过的边可以作为构成生成树的边

-> 除了能够构成环的边（即除了到达已经访问过的节点的边）

Edges that must belongs to every spanning tree: 只有单个度的节点的边

Number of spanning tree of a complete graph with N vertices

$$T = N^{N-2}$$

Running time for DFS and BFS in different graph structure

-> Connected Graph (not complete): $O(V + E)$

-> Complete Graph: $O(V^2)$

-> Bipartite Graph: $O(V^2)$, worst $O(V+E)

-> DAG: $O(V^2)$, worst $O(V+E)

-> Tree: $O(V)$

-> Acyclic Graph: $O(V + E)$

Topological Sort 拓扑排序

-> When dequeue a node, add this node to the list

-> After add all node to the list, reverse the list

Strongly Connected Component

-> 每个顶点都可以通过有向路径到达分量中任何其他顶点

-> 每个节点只属于一个强连通分量

-> 强连通分量是该节点区域内最大的一个子图

-> 单节点也是强连通分量

Binary Max Heap

Binary Heap Height: If we have a Binary Heap of N elements, its height will not be taller than $O(log\ N)$.

-> 判断一个最大堆是否合法的条件为，判断一个内部节点的是否大于其所有（一个或者两个）子节点

-> 向最大堆中插入元素时，将元素插入到堆的末尾（即索引的最后一位），随后向上修复最大堆属性

-> Maximum number of **comparisons** between heap elements required to construct a max heap of N elements using the **O(n)** Build Heap:

$$N_{need\_compare} = \lfloor N/2 \rfloor$$

接着对于每个节点的子节点进行递归比较，比较的次数取决于其子层数的数量，得到以下：

N=9, C=14; N=11, C=16; N=12, C=18

-> Minimum number of **comparisons** between heap elements required to construct a max heap of N elements using the **O(n)** Build Heap:

$$N_{need\_compare} = \lfloor N/2 \rfloor$$

接着，每个需要对比的节点与其拥有的一个或两个子节点比较，不需要递归地比较。得到以下：

N=9, C=8; N=10, C=9; N=11, C=10; N=12, C=11

-> **Maximum** number of **swaps** to construct max heap of N elements using $O(N)$ Build Heap:

$$N_{need\_compare} = \lfloor N/2 \rfloor$$

接着对于每个节点和其每层的子节点进行递归交换，交换的次数取决于其子层数的数量，得到以下：

N=9, S=7; N=10, S=8; N=11, S=8; N=12, S=10

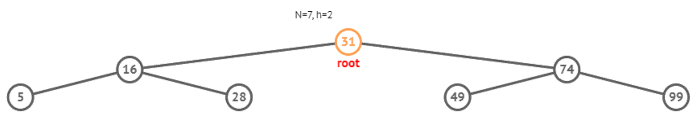
- An array A of n distinct integers that are sorted in descending order forms a valid Binary Max Heap. Assume that A[0] is not used and the array values occupy index [1:n].
- Given a Binary Max Heap, calling ShiftDown(i) ∀i > heapsize/2 will **never** change anything in the Binary Max Heap.

HashMap

Open Addressing (Linear Probing):
-> 当发生哈希冲突时，向后寻找下一个空/已删除的槽位
-> 如果到达槽位末尾，则下一个寻找的目标为第一个槽位
-> 删除操作时，将该槽位设置为 DEL，以防止在搜索时失去哈希冲突之连续性
Closed Addressing (Separate Chaining): 当发生碰撞时，即两个键散列到相同的索引时，冲突会通过将值存储到表外来解决(对于 Separate Chaining 为 DLL)。

Binary Search Tree

-> BST(二分搜索树)中，某个节点的左侧子树中的每个节点必须小于该节点值，而右侧子树中的每个节点则必须大于该定点值
Leaf Vertex：没有子节点的节点。
Internal Vertex：有子节点的节点（root 节点除外）
Insert, Search, Remove 操作都从 root 节点开始遍历，判断要插入/搜索/删除的值大于或小于当前节点
Successor: 下一个比目标节点大的值。从左往右看 BST，为目标节点的右边一个节点（无论上下）
Predecessor: 上一个比目标节点小的值。从左往右看 BST，为目标节点的左边一个节点（无论上下）



Traversal：分为三种方式, Inorder, Preorder, Postorder
Inorder:
1. 优先遍历 root 节点的左侧子树，在节点没有左侧和右侧字数的情况下访问该节点(5)
2. 返回**并访问**至上一个节点(16)
3. 遍历右侧的子节点，访问最底部节点(28)
4. 循环执行步骤 2，3
5. 返回至 root 节点并访问(31)
6. 遍历 root 节点的右侧子树，同时优先遍历右侧中的左侧子树
7. 循环执行步骤 2，3

- 访问顺序: 5 -> 16 -> 28 -> 31 -> 49 -> 74 -> 99

Preorder:
- 遍历顺序与 inorder 一致
- 但是在遍历时就访问该节点，例如从 root 节点开始，故 root 节点被第一个访问
- 访问顺序: 31 -> 16 -> 5 -> 28 -> 74 -> 49 -> 99

Postorder:
- 遍历顺序与 inorder 一致
- 但是在遍历时，只有在**当前节点没有子节点，或所有子节点都被访问过之后，才访问该节点**
- 访问顺序: 5 -> 28 -> 16 -> 49 -> 99 -> 74 -> 31
- How many structurally different BSTs can you form with **n distinct** elements?

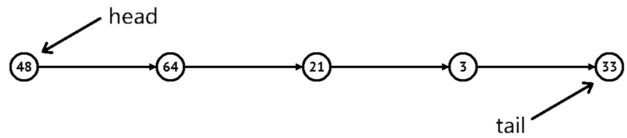
$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)! \times n!}$$

- What is the value of the element with **rank n** in this BST?
  - 从左往右看 BST，为从左往右数的第 N 个节点（无论上下）
- What is the **minimum** possible height of a **BST** with **N** elements?

$$\lfloor log_2 N \rfloor$$

Sort

Quick Sort: $O(n\ log\ n)$, 最坏情况为$O(n^2)$
分区操作： 所有比基准值小的元素摆放在基准前面，所有比基准值大的元素摆在基准的后面（相同数可在任一边）。



Stack

LIFO, `push()`, `pop()` are both $O(1)$
`push()`: 添加新元素至栈顶(head)
`pop()`: 从栈顶移除元素(head)

Queue

FIFO, `push()`, `pop()` are both $O(1)$
`push()`: 添加新元素至末尾(tail)
`pop()`: 从顶部移除元素(head)

Postfix expression

- push operand to stack, pop first 2 operand if an operator pushed in
- 4 1 2 9 3 / * + 5 * +
  - (9/3 * 2 + 1) * 5 + 4 = 39

Prefix expression

- 运算符在操作数之前，即 9/3 = /93, 9/3*2 = */932
- 先算排列在操作数前的最后一个运算符
- (9/3 * 2 + 1) * 5 + 4 -> + * + * / 9 3 2 1 5 4

Binary Max Heap Operations

ExtractMax()
-> 取出最大堆中最大的数值，对于一个合法的最大堆，为 root 节点。
-> 将索引最后一位元素提升至 root 节点，并向下修复最大堆属性
-> 时间复杂度为$O(log\ N)$
UpdateKey(i, newv)
-> 如果值的索引 `i` 已知，则可以直接更新 `A[i]=newv`
-> 然后向上和向下修复最大堆属性
-> 在知道索引的情况下，时间复杂度为$O(log/N)$
Delete(i)
1. 将该索引的值提升至 root 节点的值+1，使其成为最大堆中最大的数
2. 向上修复最大堆属性 `ShiftUp`
3. 进行 `ExtractMax()`操作

-> 时间复杂度为$O(log\ N)$