CS 180 Discussion 1A/1E

Week 2

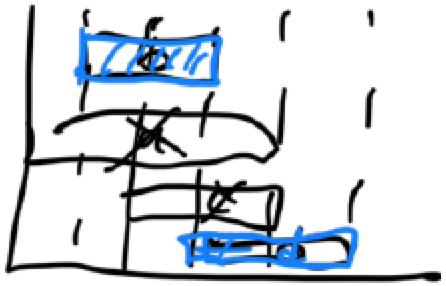
Haoxin Zheng

10/13/2023

Scheduling

- Class 1,2,…n, each class has starting time s(i) and finishing time f(i).
- Goal: Schedule maximum # of classes that are compatible (no overlap)
- Starting time?
- Finishing time?
- Length of the class?

a: [0, 4]
b: [1, 3]
c: [2, 4]
d: [3, 5]



b, a, c, d

Scheduling

- Algorithm:
  - Sort classes by finishing time $t_1, t_2\ \dots t_n$, s.t. $\mathrm{t}_1 \leq t_2 \dots \leq t_n$
  - A = Ø
  - For I = 1,2, … n:
    - If t_i is compatible to the current solution A:
      - A = A + $\{t_i\}$

- Time complexity: O(nlogn)
  - Just remember this for sorting right now, will introduce in later lectures.

Scheduling

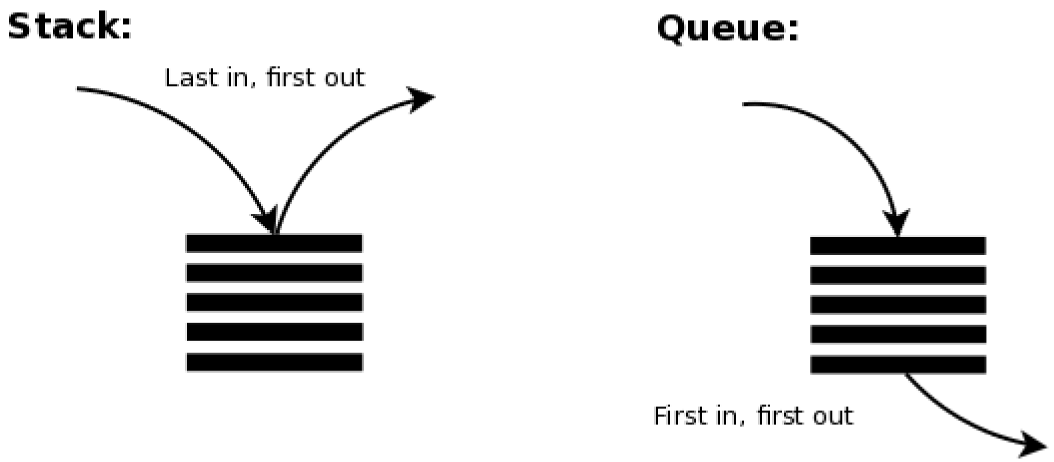
- Logic – Show the solution from greedy algorithm is optimal
  - Let $\{i_1, i_2, \dots i_k\}$ be the solution of the greedy algorithm. There can be 3 situations:
    - $\exists$ another compatible set $\{j_1, j_2, \dots j_b\}$ (b<k) -> Doesn't matter, not optimal since b<k.
    - $\exists$ another compatible set $\{j_1, j_2, \dots j_k\}$. -> Same number, greedy algorithm gives a tie, still OK.
    - $\exists$ another compatible set $\{j_1, j_2, \dots j_c\}$. (c>k) -> this is better. **Disprove this by contradiction**

- We **claim** if $\{j_1, j_2, \dots j_k\}$ is another compatible set, then $f(i_r) \leq f(j_r), \forall r = 1,2,\dots k$
- Then we can know that, if there exists a compatible $\{j_1, j_2, \dots j_k\}$, the solution generated from the greedy algorithm ($\{i_1, i_2, \dots i_k\}$) will also be compatible.
- Since $f(i_r) \leq f(j_r)$, if $\exists$ another compatible set $\{j_1, j_2, \dots j_c\}$ (c>k), then $\{j_{k+1}, \dots j_c\}$ can also be appended to the end of $\{i_1, i_2, \dots i_k\}$ -> $\{i_1, i_2, \dots i_k, j_{k+1}, \dots j_c\}$, length c.
- However, if we use the greedy algorithm, we cannot forget to include $\{j_{k+1}, \dots j_c\}$ behind i_k, since they are compatible with $\{i_1, i_2, \dots i_k\}$. **Contradiction – Disproved.**
- Next step: prove the **claim**

Scheduling

- We **claim** if $\{j_1, j_2, \ldots j_k\}$ is another compatible set, then $f(i_r) \leq f(j_r), \forall r = 1,2,\ldots k$
- Proof: by induction
  - Base case: m=1, $f(i_1) \leq f(j_1)$. This is true since the GD algorithm always picks the class with earliest ending time
  - Induction Hypothesis: $f(i_r) \leq f(j_r), \forall r \leq m$
  - Induction step: we want to show $f(i_{m+1}) \leq f(j_{m+1})$
  - $f(i_m) \leq f(j_m) \Rightarrow f(i_m) \leq s(j_{m+1})$
  - $\Rightarrow$ class j_{m+1} is compatible with $\{i_1, i_2 \ldots i_m\}$
  - $\Rightarrow$ GD will always pick i_{m+1} such that -> $f(i_{m+1}) \leq f(j_{m+1})$
  - Proved.

Stack & Queue



- Stack: Last in First out

- Queue: First in First out

- Priority Queue: Every item pushed is associated with a priority

push(5, A), push(8, B), push(6, C)

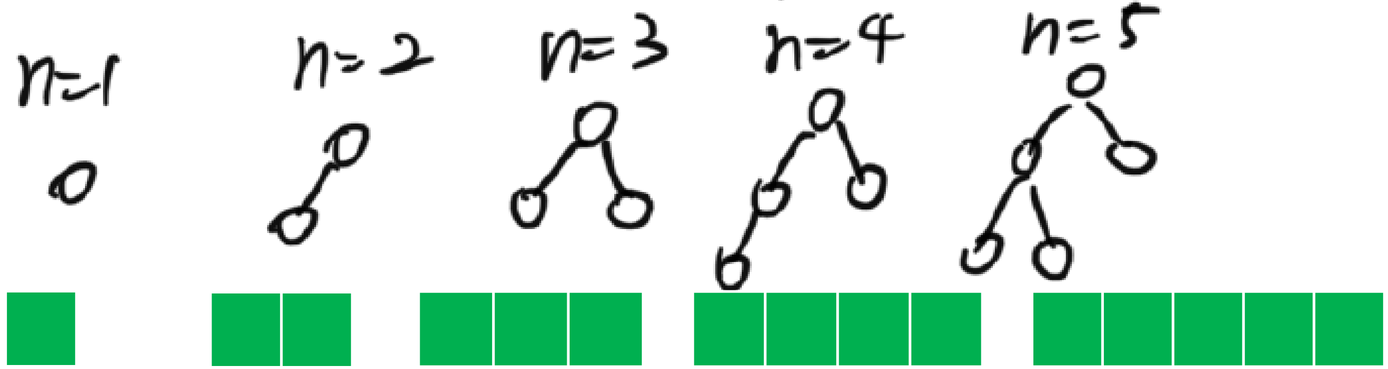
priority item

pop() → (8, B), pop() → (6, C) pop() → (5, A).

Build a priority Queue - Heap

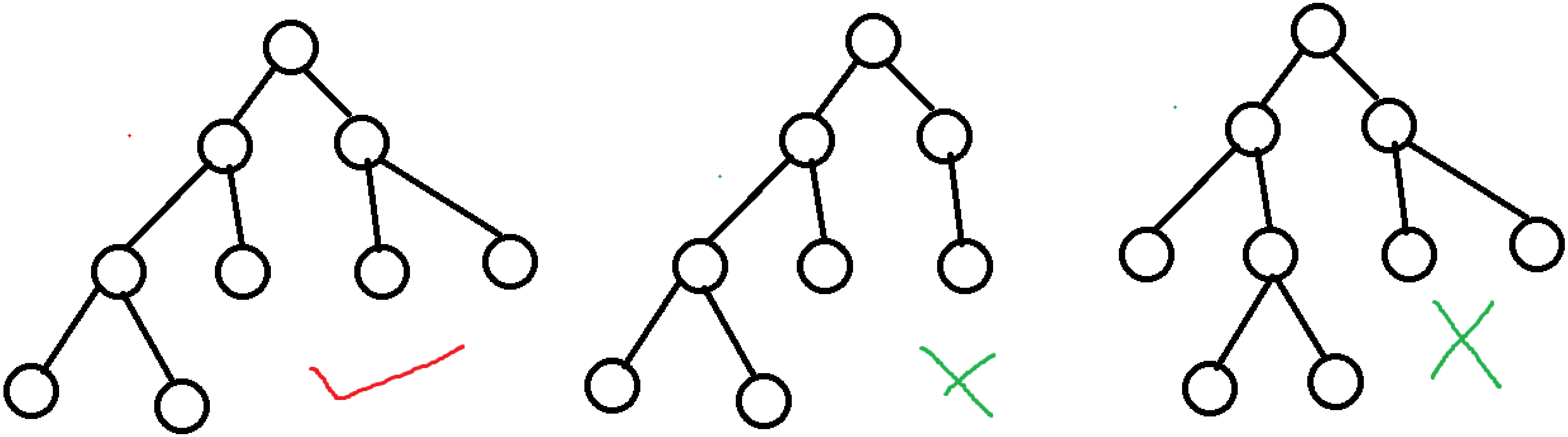
- Push(a): O(logn)
- Pop(): O(logn)
- Heap sort: O(nlogn)
- Heap is a “complete binary tree”. Similar functionalities as a priority queue.
- **Complete binary tree**: Every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

store n elements in a “complete binary tree”



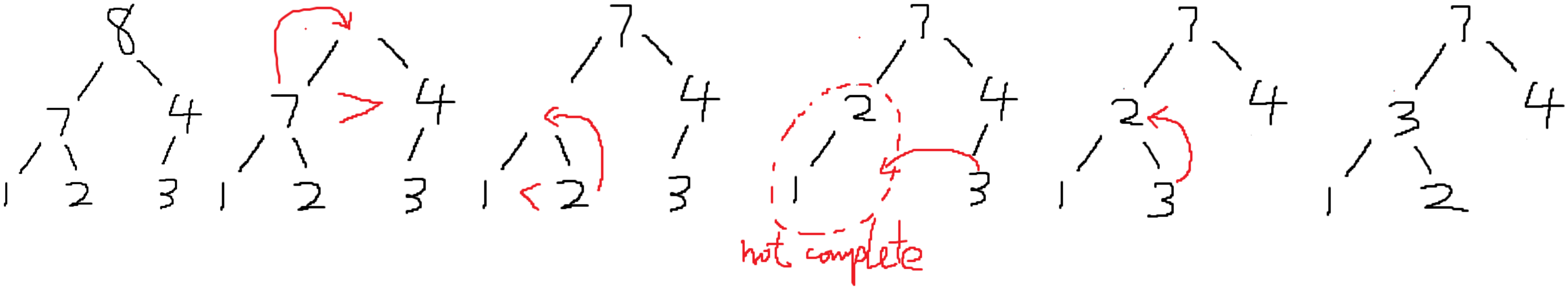
Heap

- Complete binary tree



Heap

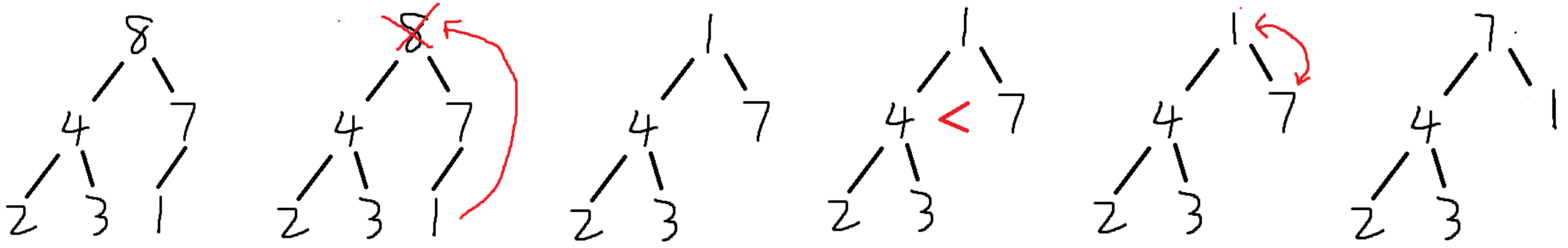
- Pop – Method 1:
  - 1) Fill the empty position with one of its children nodes until the bottom of the tree. (Swap with its smaller child in a min-heap, or its larger child in a max-heap.)
  - 2) If it is not a complete tree, fulfill the empty position with the rightmost node.
  - 3) Compare this new node with its new parent node.
    - Min-heap: Swap the new node with its new parent node if the new node is smaller.
    - Max-heap: Swap the new node with its new parent node if the new node is larger.



Heap

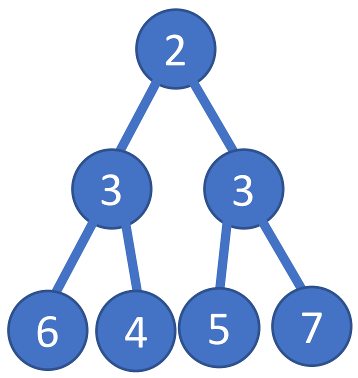
- Pop – Method 2:
  - 1) Replace the root of the heap with the last element on the last level
  - 2) Compare the new root with its children; if they are in the correct order, stop.
  - 3) If not, swap the element with one of its children and return to the previous step. (Swap with its smaller child in a min-heap and its larger child in a max-heap.)

- Example:

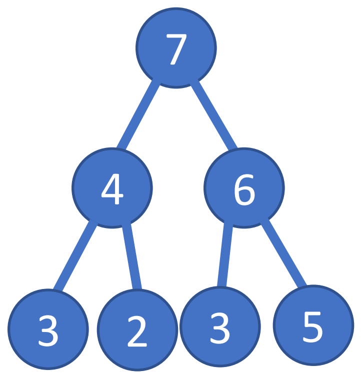


Heap – Exercise (Pop)

- Pop:
  - 1) Replace the root of the heap with the last element on the last level
  - 2) Compare the new root with its children; if they are in the correct order, stop.
  - 3) If not, swap the element with one of its children and return to the previous step. (Swap with its smaller child in a min-heap and its larger child in a max-heap.)
- Q1: Min Heap Pop



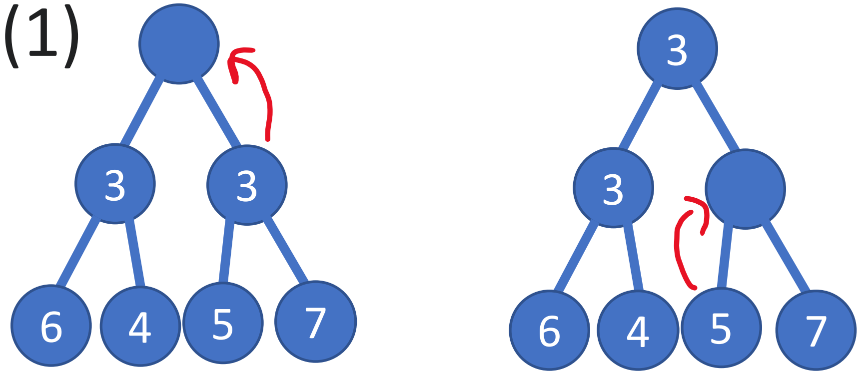
- Q2: Max Heap Pop



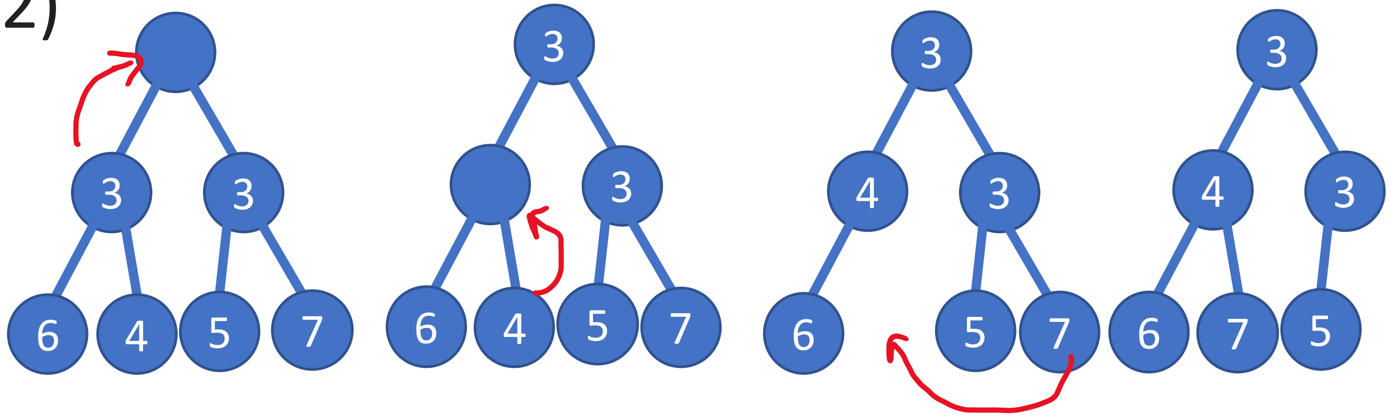
Heap – Exercise (Pop) – Method 1

- Q1: Min Heap Pop

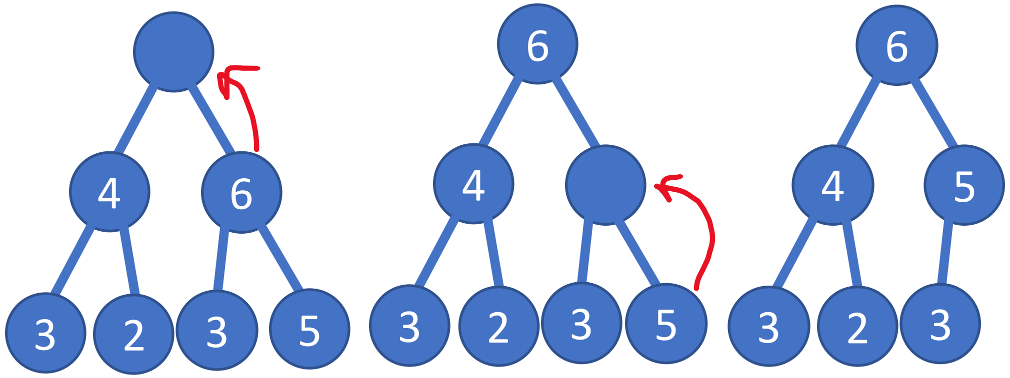
(1)



(2)



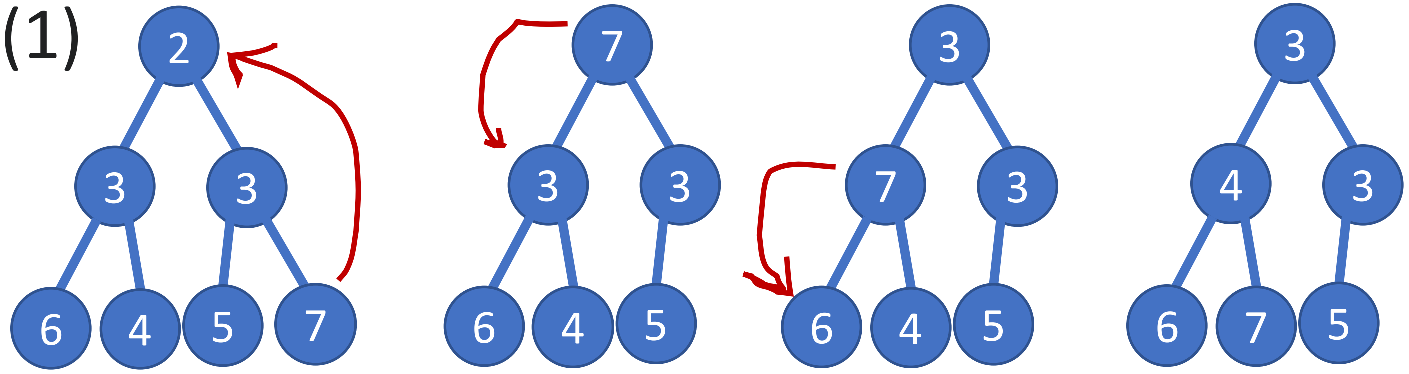
- Q2: Max Heap Pop



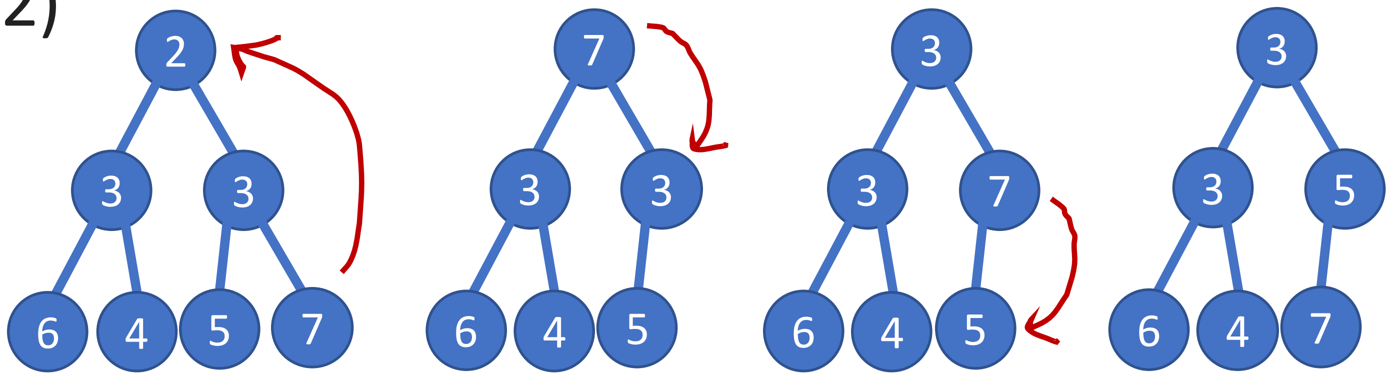
Heap – Exercise (Pop) – Method 2

- Q1: Min Heap Pop

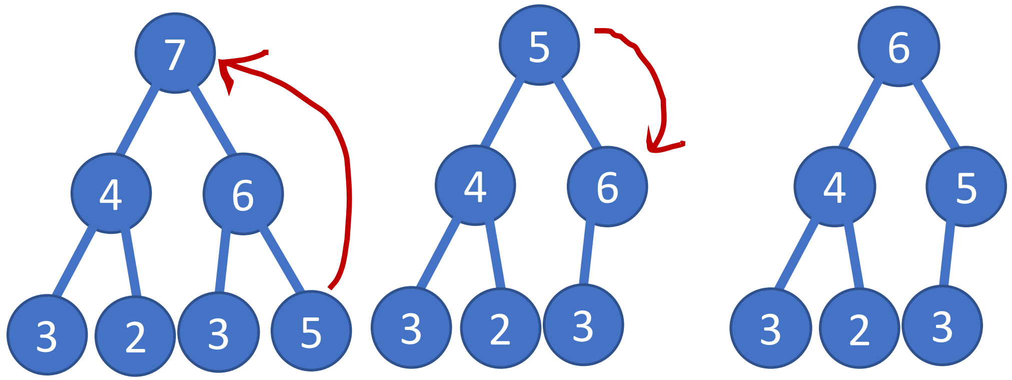
(1)



(2)



- Q2: Max Heap Pop



Heap

- Push:
  - 1) Add the element to the bottom level of the heap at the leftmost open space
  - 2) Compare the added element with its parent; if they are in the correct order, or become the root node, stop.
  - 3) If not, swap the element with its parent and return to the step 2).

- Example:

Heap – Exercise (Push)

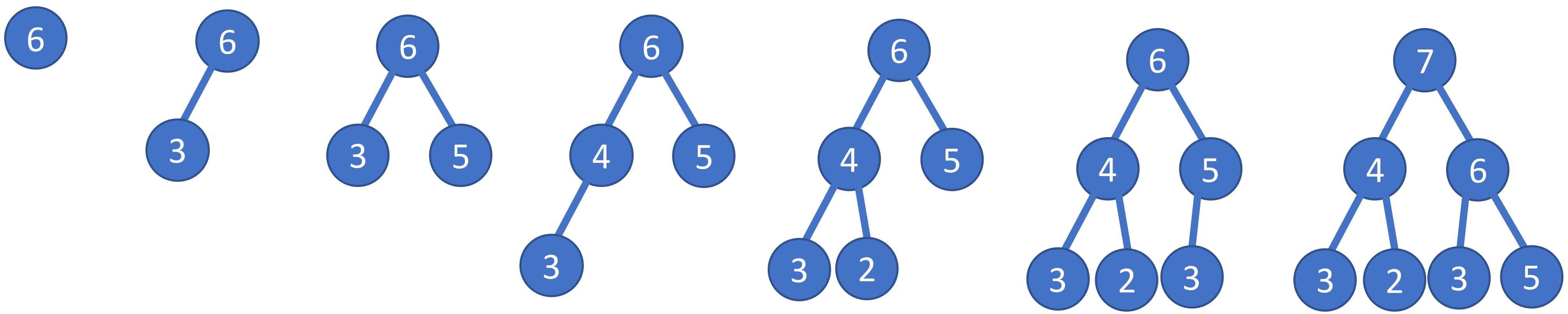
- Push:
  - 1) Add the element to the bottom level of the heap at the leftmost open space
  - 2) Compare the added element with its parent; if they are in the correct order, or become the root node, stop.
  - 3) If not, swap the element with its parent and return to the step 2).

- Q1: [6, 3, 5, 4, 2, 3, 7] – Max Heap?

- Q2: [6, 3, 5, 4, 2, 3, 7] – Min Heap?

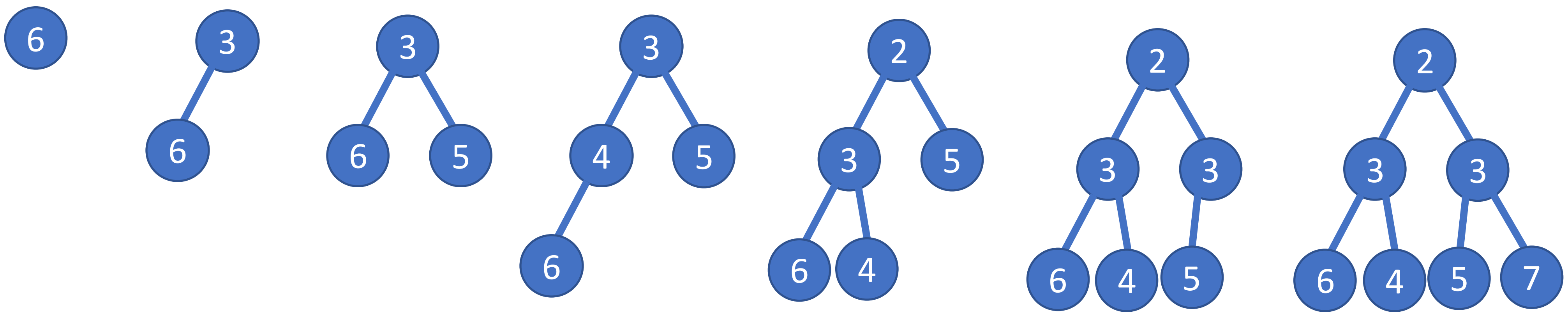
Heap – Exercise (Push)

- Q1: [6, 3, 5, 4, 2, 3, 7] – Max Heap?



Heap – Exercise (Push)

- Q1: [6, 3, 5, 4, 2, 3, 7] – Min Heap?

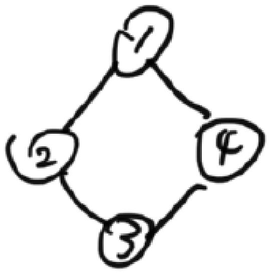


Graph

- Definitions:
  - G = (V, E). V:nodes(vertices), E:edges
  - n = |V|, number of nodes
  - m = |E|, number of edges

Graph

- Definitions:
  - *Undirected Graph*:
    - degree(u): number of edges associated with node u



degree(3)=2

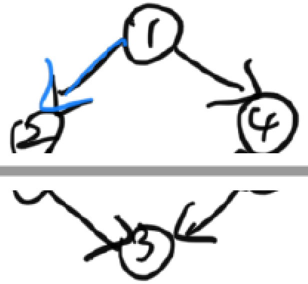
$V=\{1,2,3,4\}$

$E=\{\boxed{(1,2)}, (1,4), (2,3), (3,4)\}.$

$$\begin{array}{c c}
1 & \\
2 & \\
3 & \\
4 &
\end{array}
\begin{bmatrix}
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0
\end{bmatrix}$$

Graph

- Definitions:
  - *Directed Graph*:
    - indegree(u): number of edges pointing to node u
    - Outdegree(u): number of edges node u pointing to



$V = \{1, 2, 3, 4\}$

$E = \{(1,2), (1,4), (2,3), (4,3)\}$

indegree (u) = # edges pointing to u.

indegree(1) = 0

indegree(3) = 2

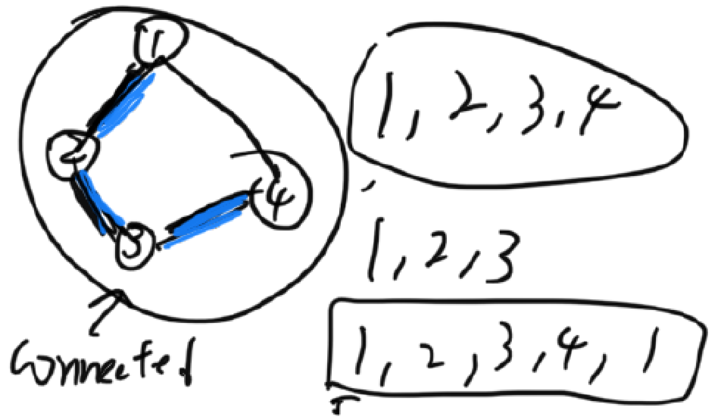
outdegree (1) = 2

outdegree (3) = 0.

outdegree (u) = # edges pointing from u.

Graph

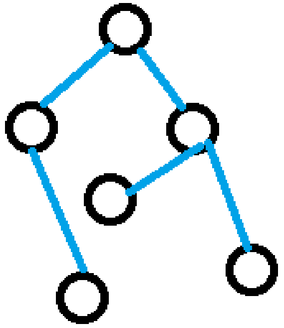
- Definitions:
  - *Path*: A path is a sequence of nodes connected by edges

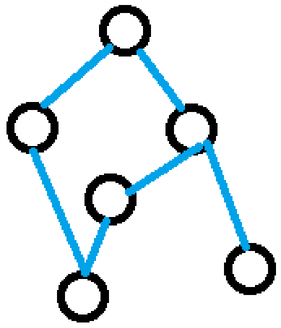


  - *Connect*: Node u and v are connected <=> ∃ a path between u and v
  - *Connected Graph*: All nodes are connected with each other
  - *Cycle:* A sequence of nodes $v_1, v_2 \ \ldots v_k \ (k > 2), v_1 = v_k$. No repeat edges, not repeat nodes except $v_1 \& v_k$

Graph

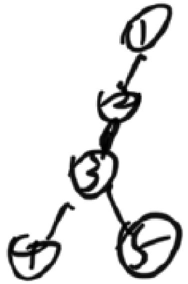
- Definitions:
  - *Tree*: an undirected graph that
    - 1) Connected
    - 2) Don't have any cycles





  - *Root a Tree*: Define a node as root.

In rooted tree:



2 is the parent of 3.
3 is the child of 2.
2 is an ancestor of 5.
5 is a descendant of 2.

  - A tree has only one path (distinct edges and distinct nodes) between any two nodes

Data Structure used to store graph

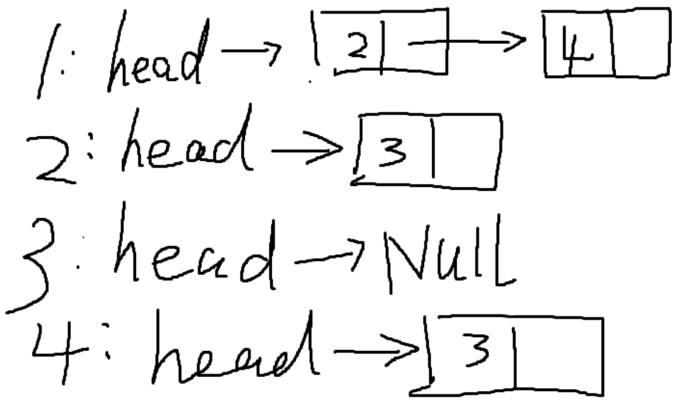
- 1) Adjacency matrix
  - A[u, v]=1 if and only if (u, v)∈E

$$A = \begin{array}{cc} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

  - **Pros:** checking whether (u, v)∈E in O(1) time
  - **Cons:** O(n^2) Space complexity
  - **Cons:** Slow to list all neighbors of a given node u since you need to traverse all v to check if (u, v)∈E

Data Structure used to store graph

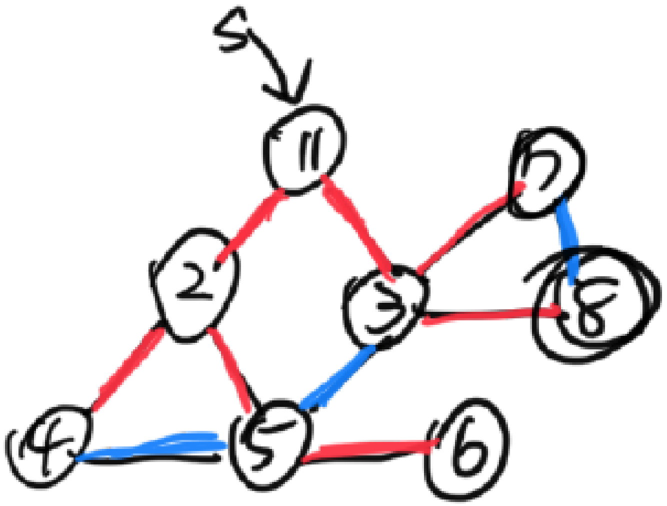
- 2) Adjacency list

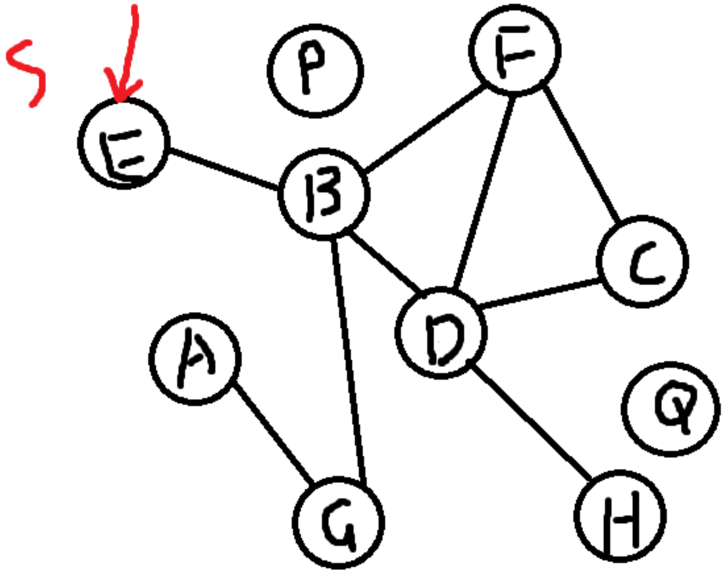


- **Cons:** checking whether (u, v)∈E in O(degree(u)) time
- **Pros:** O(|E|+|V|) Space complexity.
- **Pros:** O(degree(u)) to list all neighbors of a given node u

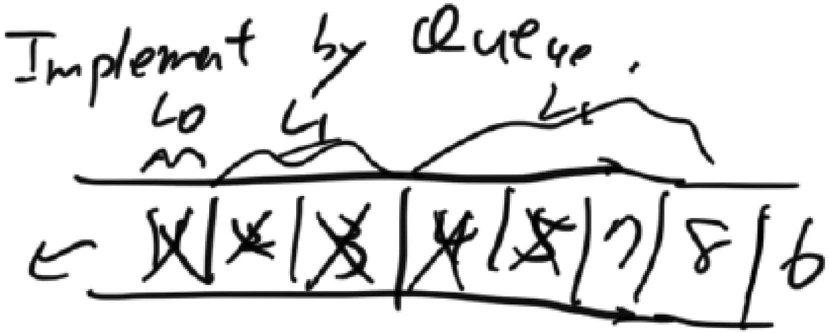
Breadth First Search

- s-t connectivity: Is node s and node t connected?



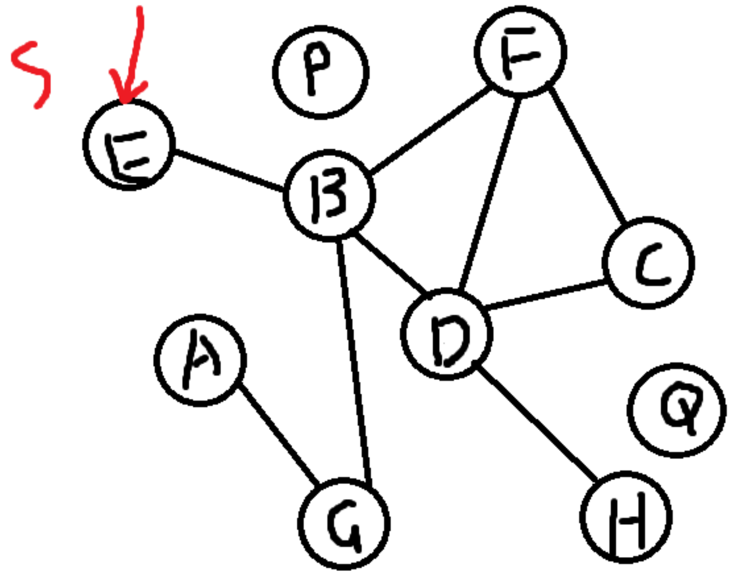


- Implement using Queue



White board

Breadth First Search



- s-t connectivity: whether s and t are connected?
- Breadth First Search (BFS)
  - visited[v]=False, $\forall$ v $\in V$
  - queue = [s]
  - while queue is not empty:
    - u = queue.pop()
    - for all v such that (u, v)∈E
      - if visited[v]==False:
        - visited[v]=True
        - queue.push(v)
  - return visited

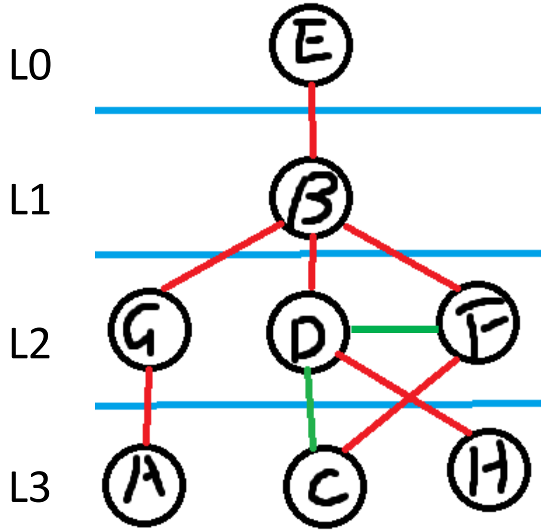
- Time complexity: O(|E|+|V|)

White board

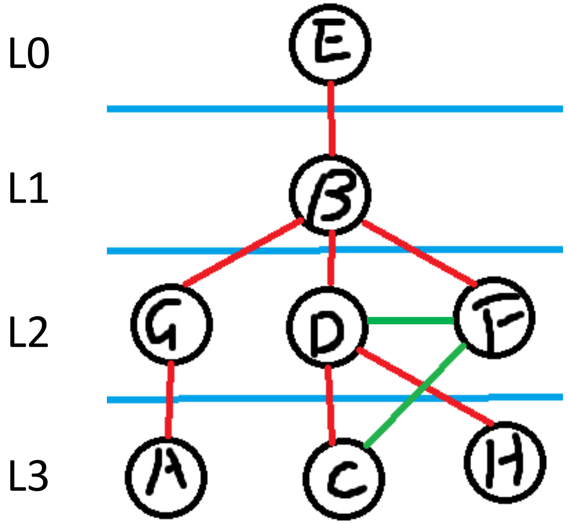
Breadth First Search

- You may get different BFS Tree depends on the order of traversing

F first:

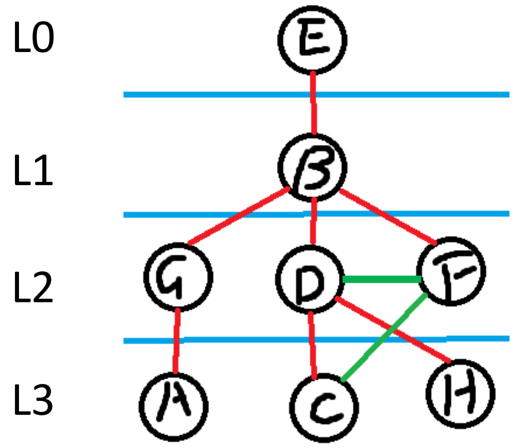


D first:



Breadth First Search

- The shortest path from starting node (E here) to node X, is the level index i of node X after BFS.



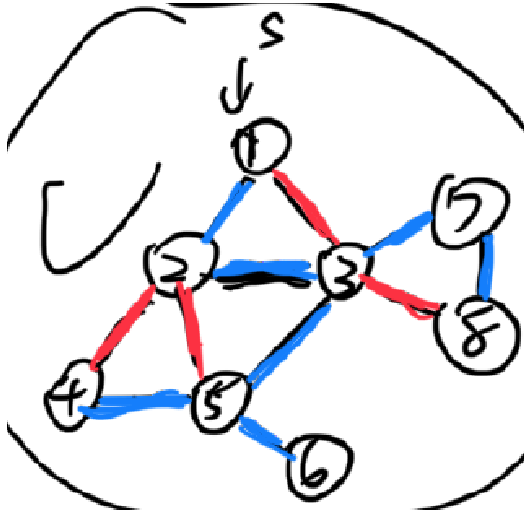
- Proof:
  - 1. Prove the shortest path cannot be longer than index i
    - We want to show there is a path with length of i by connecting different nodes between L_k & L_{k+1} .
    - By BFS, we know there has to be an edge between current node and the node in the previous layer.
  - 2. Prove the shortest path cannot be shorter than index i

    - By proof by contradiction. If there is such a path, there has to be a link between L_k & L_{k+a} $(a \geq 2)$.

    - BFS algorithm doesn't allow such scenario happens. Assume there is an edge (A, B), and node A in L_k and node B in $L_{k+a} (a \geq 2)$, then by BFS, B should be at L_{k+1}. Contradiction.

Depth First Search

- DFS(u):
  - visited[u] = True
  - for v s.t. (u, v)∈E:
    - If visited[v] = False:
      - DFS(v)



1	2	3	7	8	5	4	6

White board

Exercises

2. Give an algorithm to detect whether a given undirected graph contains a cycle. If the graph contains a cycle, then your algorithm should output one. (It should not output all cycles in the graph, just one of them.) The running time of your algorithm should be $O(m + n)$ for a graph with n nodes and m edges.

Exercises

2. Give an algorithm to detect whether a given undirected graph contains a cycle. If the graph contains a cycle, then your algorithm should output one. (It should not output all cycles in the graph, just one of them.) The running time of your algorithm should be $O(m + n)$ for a graph with n nodes and m edges.

- Using BFS. For every visited vertex v, if there is an adjacent u such that u is already visited and u is not a parent of v, then there exists a edge either within the same level of BFS tree, or between the level of BFS tree.
- From the root node, we have at least two paths that can approach v.
- This means there is a cycle in the graph.
- If we don’t find such an adjacent for any vertex, we say that there is no cycle.

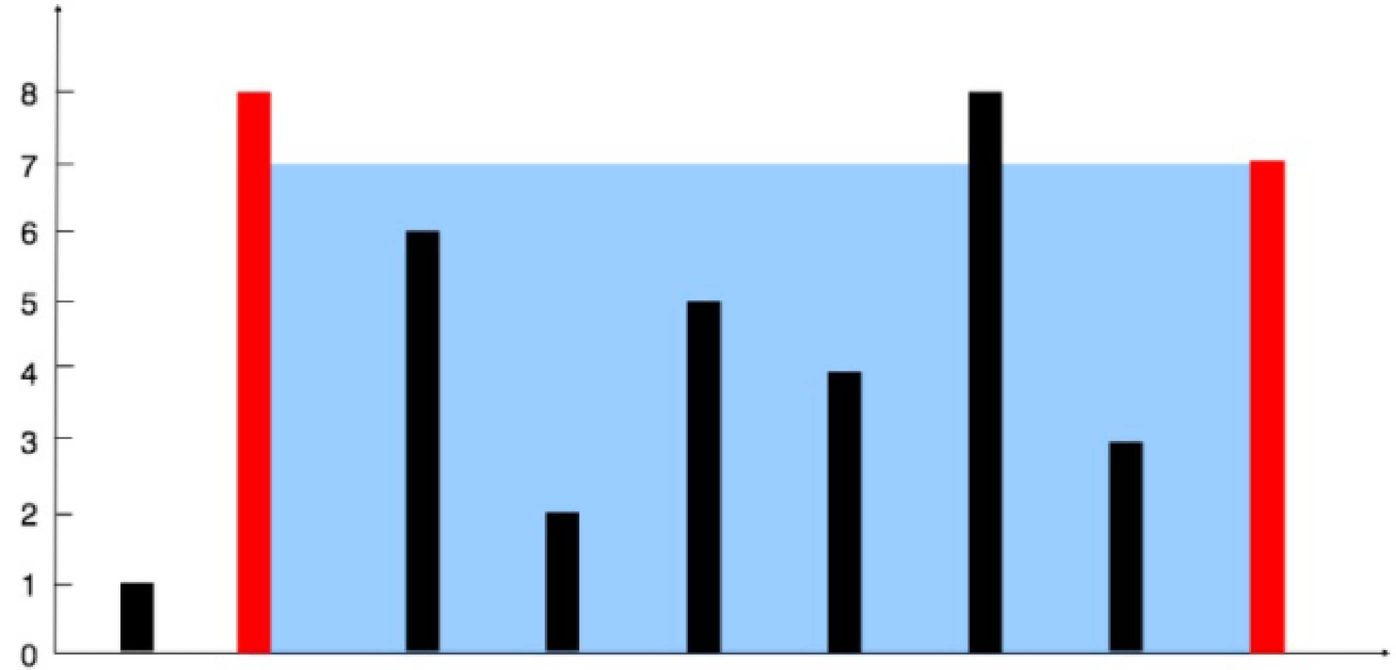
11. Container With Most Water

Medium 12470 802 Add to List Share

Given `n` non-negative integers $a_1, a_2, ..., a_n$, where each represents a point at coordinate (i, a_i) . `n` vertical lines are drawn such that the two endpoints of the line `i` is at (i, a_i) and `(i, 0)`. Find two lines, which, together with the x-axis forms a container, such that the container contains the most water.

Notice that you may not slant the container.

Example 1:




```
Input: height = [1,8,6,2,5,4,8,3,7]
Output: 49
Explanation: The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In
this case, the max area of water (blue section) the container can contain is 49.
```

Example 2:

```
Input: height = [1,1]
Output: 1
```

Example 3:

```
Input: height = [4,3,2,1,4]
Output: 16
```

Proof – by contradiction

- Suppose the returned result is not the optimal solution. Then there must exist an optimal solution, say a container with a_l and a_r such that it has a greater volume V than the one we got (S). $V > S$. The two pointers didn't point them at the same stage using our algorithm, or we will have the V in our record.

- Since our algorithm stops only if the two pointers meet. So, we must have visited only one of them. Let's say we have visited a_l but not a_r.

- When a pointer stops at a_l, two situations:
  - **Didn't move: The other pointer also points to a_l.**
    In this case, iteration ends. But the other pointer must have visited a_r on its way from right end to a_l. Contradiction to the initial discussion.
  - **Moved: The other pointer arrives at $a_r{}'$, that is greater than a_l before it reaches a_r.**
    In this case, we move a_l. Two situations about current volume $V_{current}$ between a_l & $a_r{}'$:
    - a_r is higher than a_l. $V_{current} = h(a_l) \times w(a_l, a_r') > h(a_l) \times w(a_l, a_r) = V$
    - a_r is lower than a_l. $V_{current} = h(a_l) \times w(a_l, a_r') > h(a_r) \times w(a_l, a_r) = V$

    which means that a_l and a_r is not the optimal solution – Contradiction to the original assumption

Exercises

200. Number of Islands

Medium 10562 276 Add to List Share

Given an `m x n` 2D binary grid `grid` which represents a map of `'1'`s (land) and `'0'`s (water), return *the number of islands*.

An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

```
Input: grid = [
  ["1","1","1","1","0"],
  ["1","1","0","1","0"],
  ["1","1","0","0","0"],
  ["0","0","0","0","0"]
]
Output: 1
```

Example 2:

```
Input: grid = [
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]
Output: 3
```

Constraints:

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 300`
- `grid[i][j]` is `'0'` or `'1'`.

Exercises

```
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:

        def bfs(i, j, grid):

            q = collections.deque()
            q.append((i, j))
            grid[i][j] = "#"

            while q:
                (curr_i, curr_j) = q.popleft()
                directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
                for direct in directions:
                    temp_i = curr_i + direct[0]
                    temp_j = curr_j + direct[1]

                    if temp_i>=0 and temp_i <len(grid) and temp_j>=0 and temp_j<len(grid[0]) and grid[temp_i][temp_j] == "1":
                        q.append((temp_i, temp_j))
                        grid[temp_i][temp_j] = "#"

        if not grid: return 0
        count = 0

        for i in range(len(grid)):
            for j in range(len(grid[0])):
                if grid[i][j] == "1":
                    bfs(i, j, grid)
                    count += 1

        return count
```

Example 2:

```
Input: grid = [
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]
Output: 3
```