Contents

Data Structure

Treap:

Definition:

Also called Cartesian Tree, randomized binary search tree, is a data structure which combines Binary Tree and Binary Heap

(Tree + Heap -> Treap)

More specifically, treap is a data structure that stores pairs (X, Y) in a binary tree in such a way that it is a binary search tree by X and a binary heap by Y. Assuming that all X and all Y are different, we can see that if some node of the tree contains values (X_0, Y_0), all nodes in the left subtree have $X < X_0$, all nodes in the right subtree have $X > X_0$, and all nodes in both sides have $Y < Y_0$.

Advantages:

When **X values are keys** (and at same time the values stored in the treap), and **Y values called priorities**.

Without priorities, the treap would be a regular binary search tree by X, and one set of X values could correspond to a lot of different trees, some of them degenerate (for example, in the form of a linked list), and therefore extremely slow (the main operations would have O(n) complexity).

At the same time, priorities allow to uniquely specify the tree that that will be constructed (of course, it does not depend on the order in which values are added), which can be proven using corresponding theorem. Obviously, if we choose the priorities randomly, we will get non-degenerate trees on average, which will ensure O(log n) complexity for the main operations.

Hence another name of this data structure - randomized binary search tree.

Time complexities:

Insert (X, Y): O(log n)

Adds a new node to the tree. One possible variant is to pass only X and **generate Y randomly** inside the operation (while ensuring that it's different from all other priorities in the tree).

Search (X): O(log n)

Looks for a node with the specified key value X. The implementation is the same as for an ordinary binary search tree.

Delete (X): O(log n)

Just looks for a node with the specified key value X and removes it from the tree. (Same with Searching)

Summary:

When add new node to the tree with X is the key, generate Y randomly.

Adding, Searching, deleting do normally like doing with binary search tree.

After change of node (adding or deleting), heapify the tree using Y values to ensure the Treap.

Sort Algorithms

Pigeonhole Sort:

Definition:

Pigeon Sorting is a sorting algorithm that suitable for sorting lists of elements where the number of elements and the number of possible key values are approximately the same.

It is similar to counting sort but differs in that it moves items twice: once to the buckets array (pigeonhole) and again to the result array.

Algorithm:

1. Find min and max value in the given array, also find number of pigeonholes = max – min + 1
2. Initialize an empty array with number of pigeonholes we found from step 1.
3. Visit each element and put it in the respectively pigeonhole, the element arr[i] is put at index arr[i] – min.
4. Start the loop all over the pigeonhole array in order and put the elements from non-empty holes back to the original array as the result.

Example:

Given an array

5	3	9	5	1	2	6	8

We found **max = 9** and **min = 1**, as well as range = 9 – 1 + 1 = 9.

So, we initialize a new empty array as pigeonholes which **size = 9**

arr[]

#0	#1	#2	#3	#4	#5	#6	#7
5	3	9	5	1	2	6	8

Pigeonholes[]

#0	#1	#2	#3	#4	#5	#6	#7	#8

After that, iterating from arr[0] to arr[7], and put arr[i] to **arr[i] – min** pigeonhole

.e.g., arr[0] = 5, then arr[0] – min = 5 – 1 = 4, so, put 5 in the pigeonhole #4 and so on, we got:

#0	#1	#2	#3	#4	#5	#6	#7	#8
1	2	3		5, 5	6		8	9

Then, move all elements from non-empty holes from #0 to #8 pigeonhole to the result array, as well as sorted:

arr[]

#0	#1	#2	#3	#4	#5	#6	#7
1	2	3	5	5	6	8	9

Odd-Even Sort:

Definition:

Also known as Brick sort, it is basically a variation of bubble-sort. The algorithm is divided into two phases: Odd phase and Even phase

Algorithm:

1. While the array is not sorted
2. Set the flag to marks the array is sorted.
3. For all odd index elements, from arr[1] to arr[n-2]
4. If arr[i] > arr[i+1]
    - Swap those two elements.
    - i increased by 2.
    - Marks the array is not sorted using flag.
5. For all even index elements, from arr[0] to arr[n-2]
6. If arr[i] > arr[i+1]
    - Swap those two elements.
    - i increased by 2.
    - Marks the array is not sorted using flag.
7. The array is sorted when did not jump in 4 and 6.

Example:

Given an array

#0	#1	#2	#3	#4	#5	#6	#7	#8	#9
5	1	12	55	12	0	6	19	77	3

The elements with the same color are swapped.

Phase 1: Odd

#0	#1	#2	#3	#4	#5	#6	#7	#8	#9
5	1	12	12	55	0	6	19	77	3

Phase 2: Even

#0	#1	#2	#3	#4	#5	#6	#7	#8	#9
1	5	12	12	0	55	6	19	3	77

Phase 3: Odd

#0	#1	#2	#3	#4	#5	#6	#7	#8	#9
1	5	12	0	12	6	55	3	19	77

Phase 4: Even

#0	#1	#2	#3	#4	#5	#6	#7	#8	#9
1	5	0	12	6	12	3	55	19	77

Phase 5: Odd

#0	#1	#2	#3	#4	#5	#6	#7	#8	#9
1	0	5	6	12	3	12	19	55	77

Phase 6: Even

#0	#1	#2	#3	#4	#5	#6	#7	#8	#9
0	1	5	6	3	12	12	19	55	77

Phase 7: Odd

#0	#1	#2	#3	#4	#5	#6	#7	#8	#9
0	1	5	3	6	12	12	19	55	77

Phase 8: Even

#0	#1	#2	#3	#4	#5	#6	#7	#8	#9
0	1	3	5	6	12	12	19	55	77

So, the array is sorted.

Search Algorithms

We focus on Graph Search algorithms because it is helpful to us on the next course: Introduction to Artificial Intelligence.

Given a graph with vertices and edges

Breath First Search:

Algorithm:

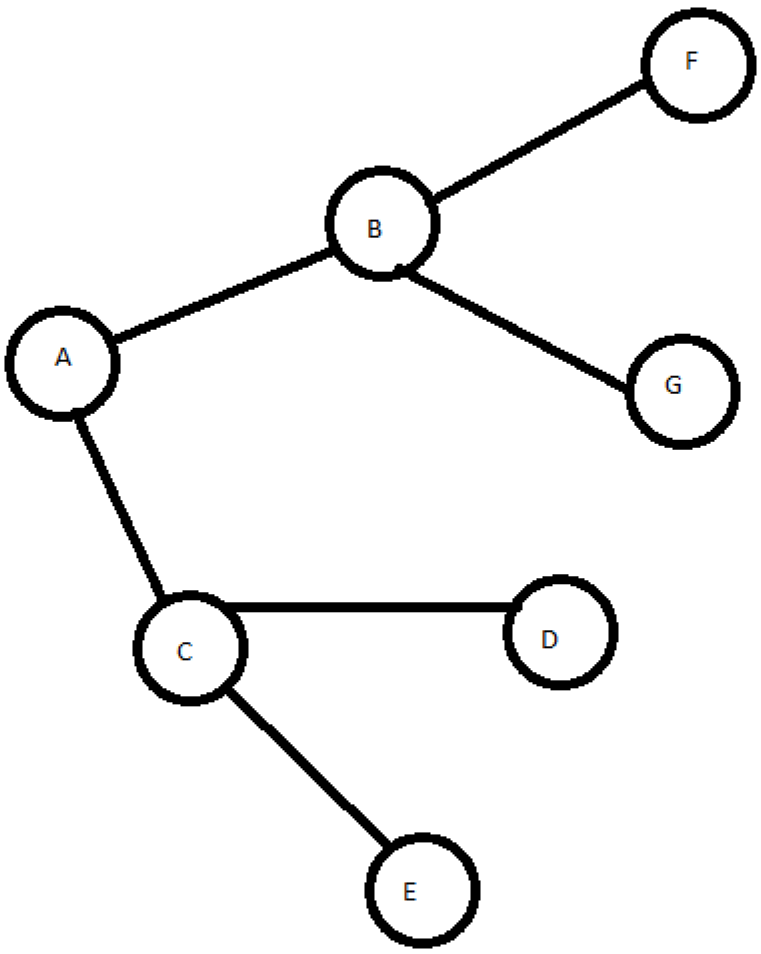
1. Traverse from root node.
2. Traverse to **all successor** of the current node.
3. When all child nodes are visited, select one successor to traverse to it own successors.
4. Return to another nodes from step 2 and do the same with step 3.
5. Until done step 3 for all nodes from step 2, move to all node from step 3 and do the same in step 2.

Psuedocode:

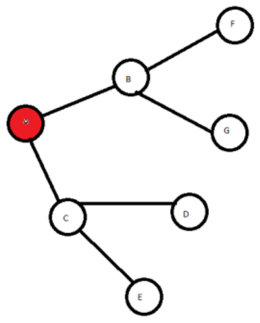
```
procedure BFS(G, root) is
    let Q be a queue
    label root as explored
    Q.enqueue(root)
    while Q is not empty do
        v := Q.dequeue()
        if v is the goal, then
            return v
        for all edges from v to w in G.adjacentEdges(v) do
            if w is not labeled as explored then
                label w as explored
                Q.enqueue(w)
```

Example:

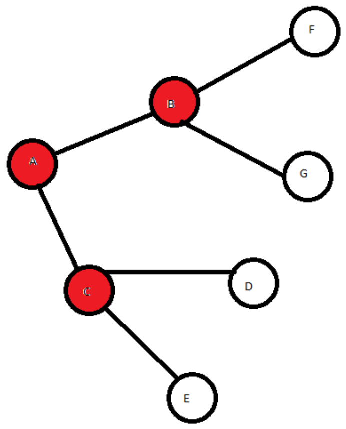
Given a graph



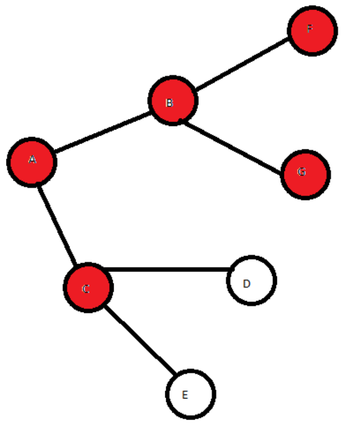
First, set **A** as a root node, then it is visited.



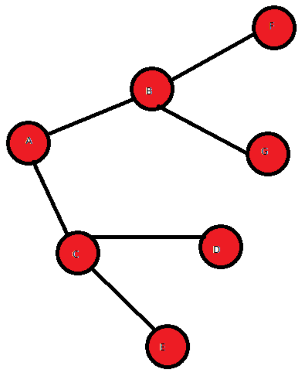
After that, traverse to A child nodes. Following **B** and **C**.



Then, we are traversing with alphabetical order priority, so, traverse to **B** child nodes first: **F** and **G**



When **F** and **G** are visited (All **B** child nodes are visited), we backtrack to **C** child nodes: **D** and **E**



After traversing, we got the path: ABCFGDE

Depth First Search:

Algorithm:

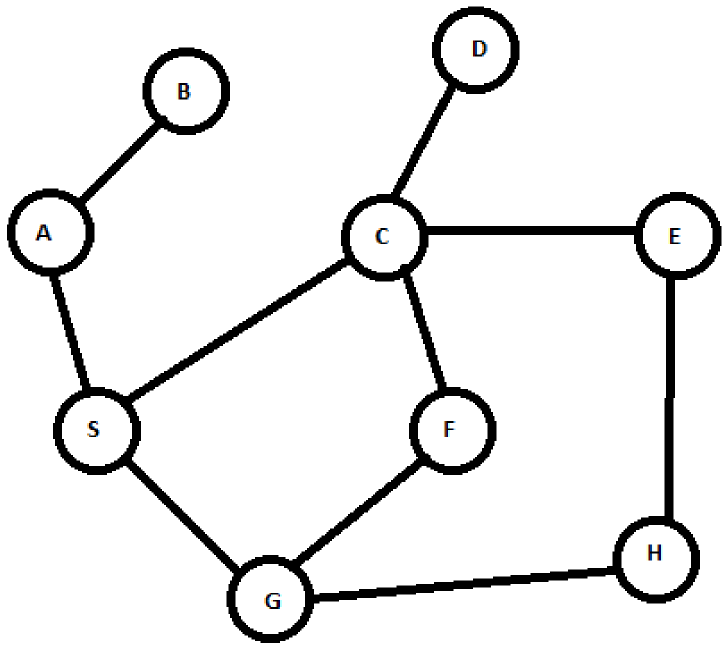
1. Traverse from root node, set as visited.
2. Traverse to root child, called current node, set as visited.
3. When current node is not a leaf node or unvisited node, continue traverse to this next node and so on.
4. When reach the leaf node or a visited node, backtrack to the node that we spit a branch to traverse and go to another branch.

Psuedocode:

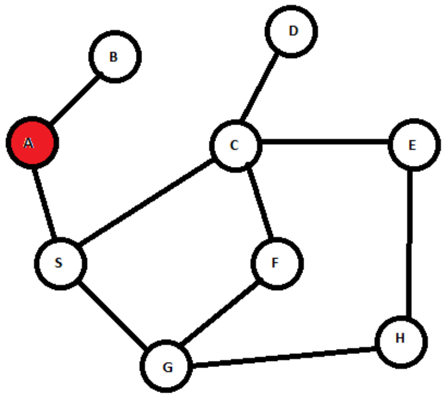
```
procedure DFS(G, v) is
    label v as discovered
    for all directed edges from v to w that are in G.adjacentEdges(v) do
        if vertex w is not labeled as discovered then
            recursively call DFS(G, w)
```

Example:

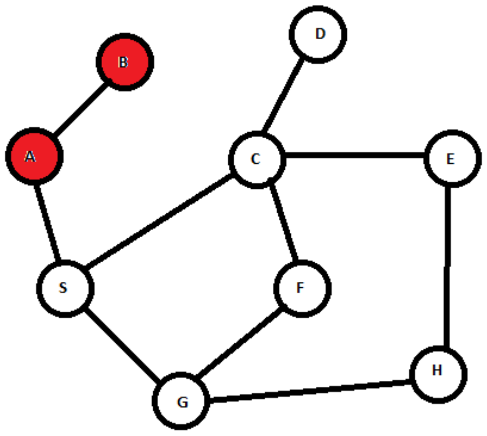
Given a graph



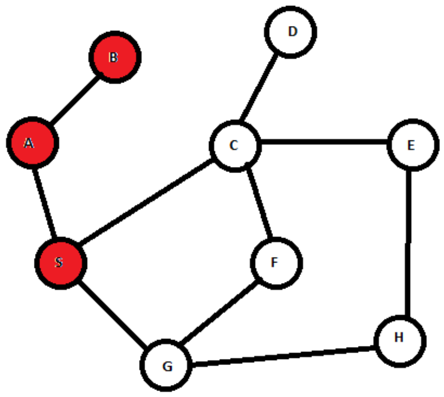
First, set **A** is a root node, as well as visited.



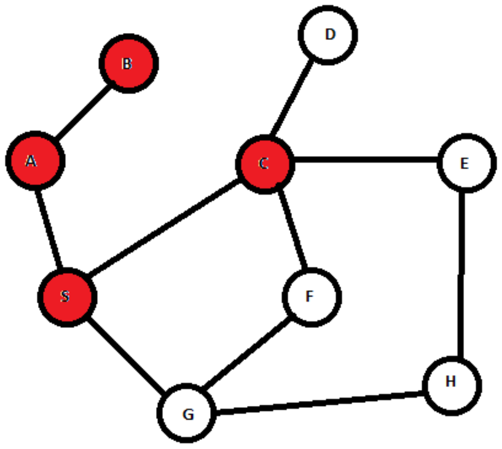
Then, following alphabetical order, we traverse to **B**.



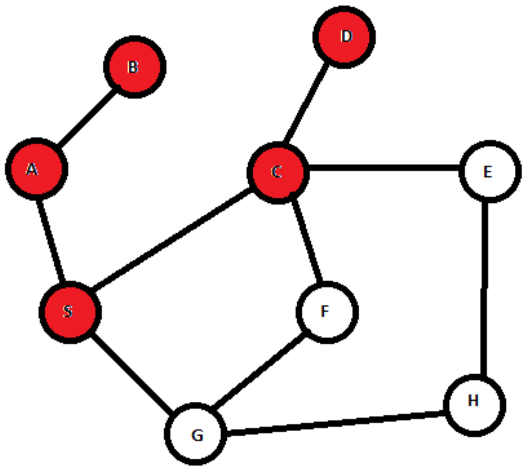
After that, we can see **B** is a leaf node, so, backtrack to **A** and go to another branch, **S**



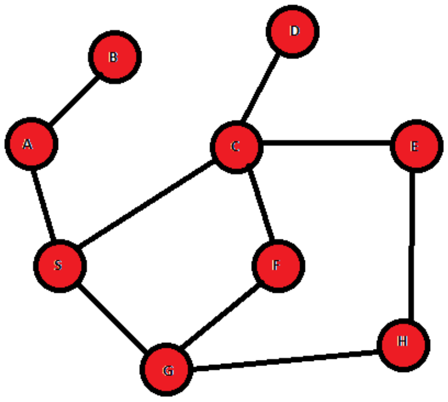
So on, traverse to **C**



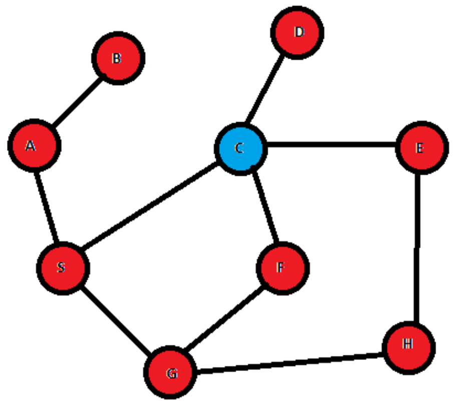
Here we got 3 ways to go, follow alphabetical order, that we go to the branch of **D**



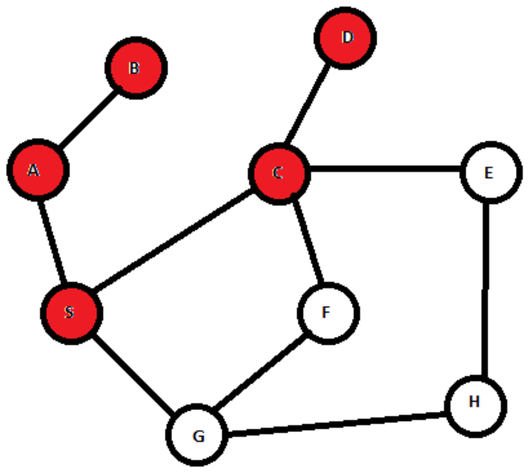
D is a leaf node that we backtrack to **C** and go to **E->H->G->F**



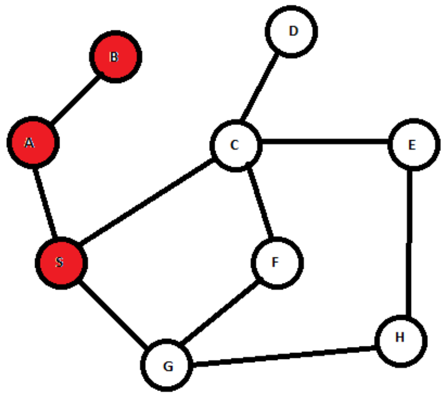
Next, continue traverse to **C**, but **C** is already visited



So, we backtrack to **C**



But there is no child node that unvisited, so, backtrack again to **S**



Again, all **S** child nodes are visited, we backtrack to **A** which is root node, and all child nodes are visited, the algorithm stop.

After traversing, we got the path: ABSCDEHGF