# ADT and Data Structures

Abstract data types are used to represent more a concept or an idea, with no regard for implementation

Say, using a stack. a stack is an abstract type because it has the idea of popping and pushing. but it doesnt say how to implement it. because it can be implemented in many ways. like using arrays that change size, or it can use a linked list that tracks where the head is currently.

**A data structure** is a concrete representation of data, and this is from the point of view of an implementer, not a user

most of the time, using a linked list is doubly linked

## Proving correctness

When proving correctness for a data structure it best to use an invariant that the data structure will always hold and keep true no matter what happens.

Normally i go through all of the functions the data structure will have and choose arbitrary values as the parameters. and show how the invariant still holds even when performing these functions. Going case by case for each function and thinking of every possible way the data structure will respond to it, BUT the numbers must be arbitrary. Otherwise you would be trying to prove correctness by giving examples which will not work.

# Trees

a tree is a graph that is connected and has no cycles

## Terminology

- to make things easier, all the leaves will have 2 children that will be null

| Term | Description |
| --- | --- |
| root | the first node at the beginning of the tree |
| internal node | nodes with at least one child |
| leaf/external nodes | node without children (or represented with both children as NULL) |
| ancestors | the parents |

| Term | Description |
|---|---|
| descendants | a child |
| depth of a node | how far down the node is the number of ancestors it has not including itself |
| level | the set of nodes at a given depth |
| height of a tree | the max depth |
| Subtree | tree made up of node and some decendents |
| edge | pair of nodeswhere one is a parent of another |
| path | sequence of nodes that would lead from one to another |
| forest | a bunch of trees together, unconnected from other trees |

# Heaps

A heap is a binary tree storing (key, value) items at its nodes, satisfying the following properties:

1. Heap-Order: for every node m ≠ root, key(m) >= key(parent(m))
2. Complete Binary Tree: let h be the height
    - every level i < h is full (i.e., there are 2i nodes)
    - remaining nodes take leftmost positions of level h

mainly used for priority queues.

they contain the max or min value is the root of the tree, they are NOT like binary trees, because the left value can be greater than the parent

just **as long as the parent is smaller than children** (or larger in the case of a max heap)

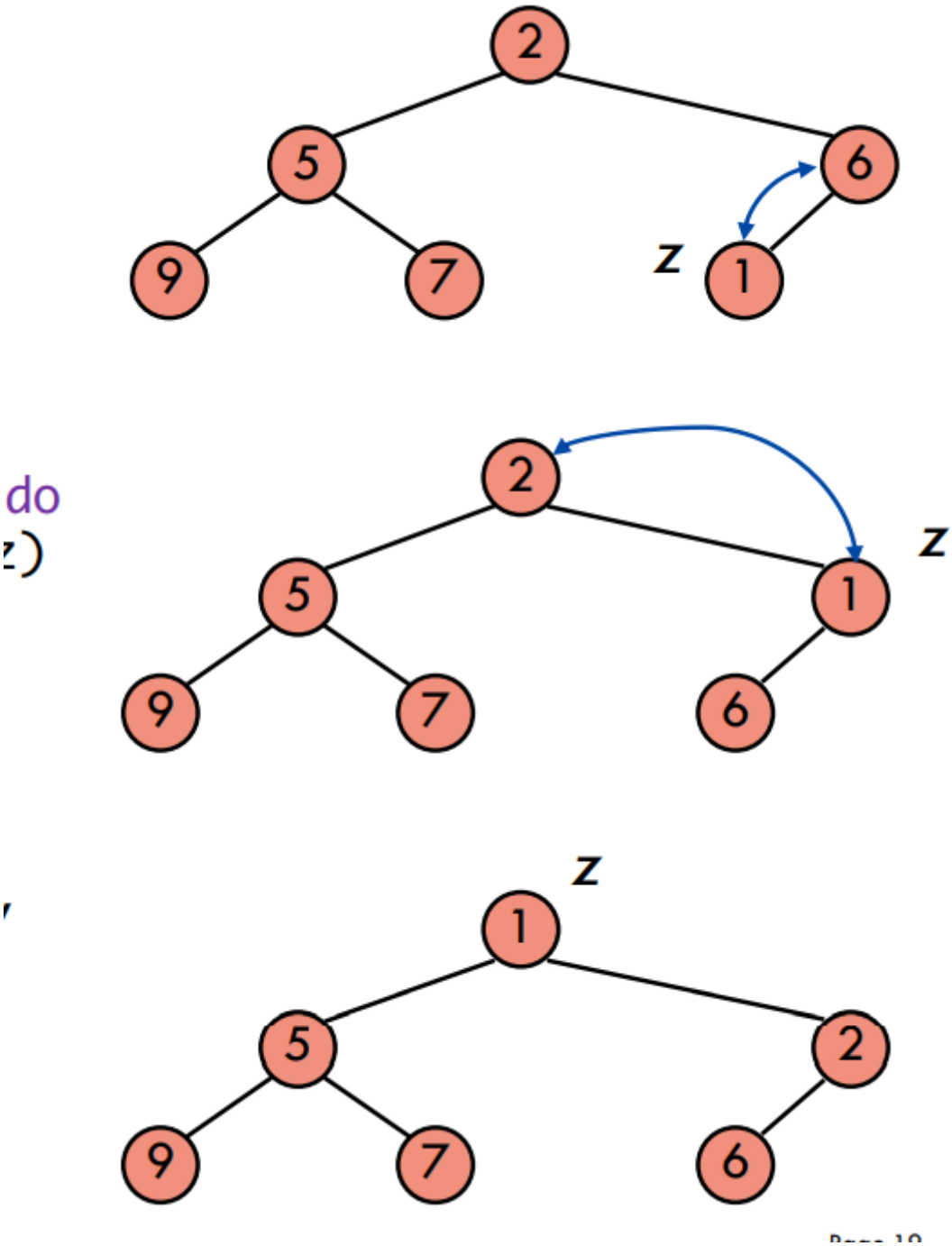
The last node is the rightmost node of maximum depth

# Retaining log(n) height

Restore heap-order property by swapping keys along upward path from insertion point
Complexity: O(log n) time because the height of the heap is log n

```
def up_heap(z):
        while z ≠ root and key(parent(z)) > key(z) do
                swap key of z and parent(z)
                z ← parent(z)
```

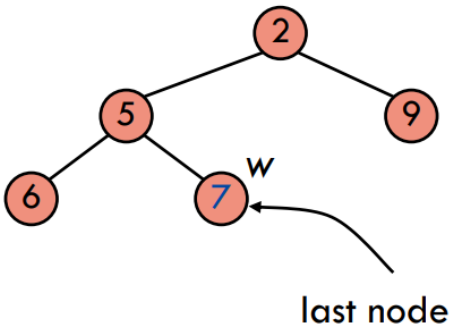
in this example the newest inserted node would go to the bottom then switch

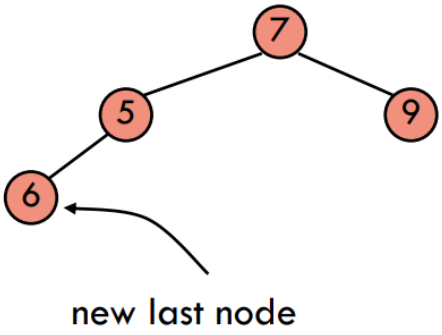


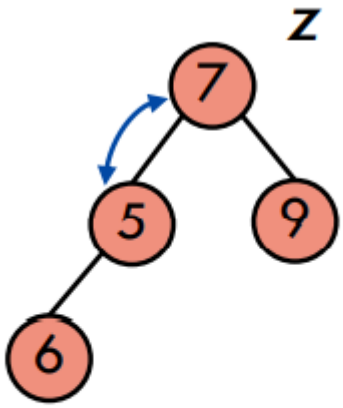

Restore heap-order property by swapping keys along downward path from the root

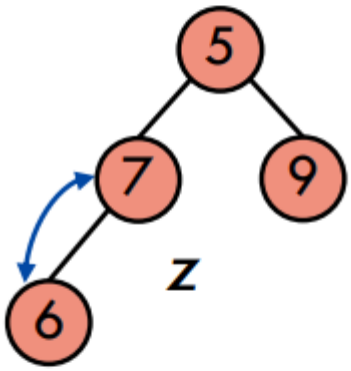
```
def down_heap(z):
        while z has child with key(child) < key(z) do
                x ← child of z with smallest key
                swap keys of x and z
                z ← x
```

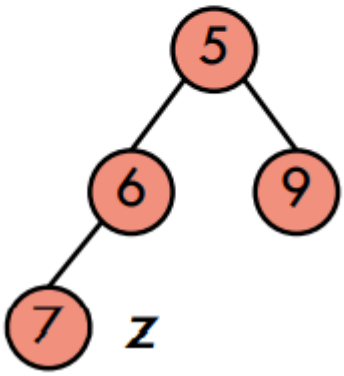
when there is a node from the top that is removed. first get the last node inserted? then make that the top and keep switching



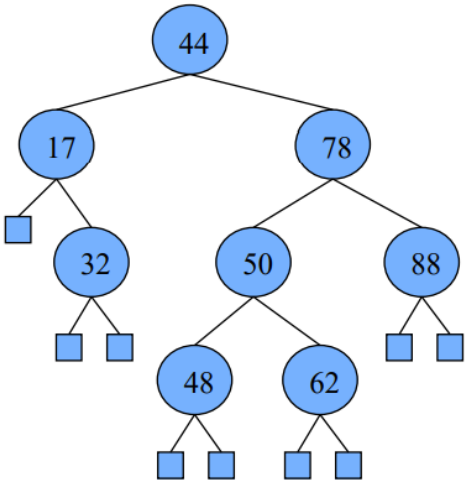



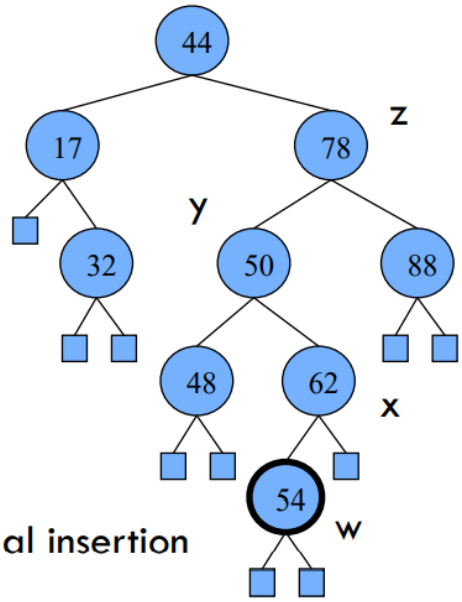

## AVL Tree

AVL trees are rank-balanced trees, where r(v) is its height of the subtree rooted at v

- Let w be location of newly inserted node
- Let z be lowest ancestor of w, whose children ***heights*** differ by 2 (their subtrees would differ)
- Let y be the child of z that is ancestor of w (taller child of z, the one with a bigger depth/larger subtree)
- Let x be child of y that is ancestor of w




before inserting 54

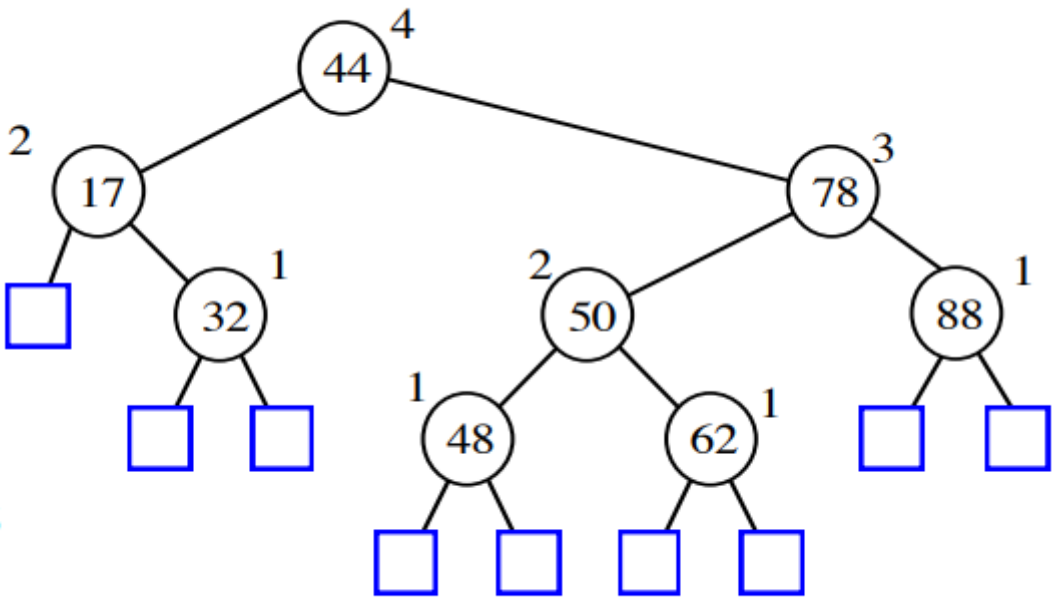



after initial insertion

ceq

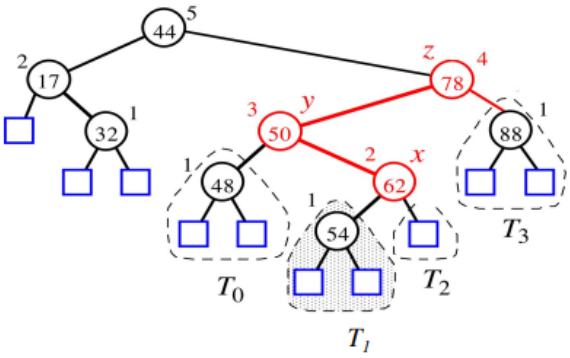
ght

/

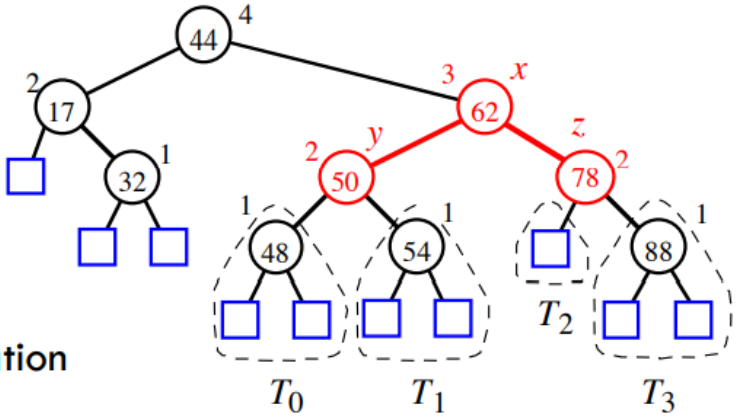



nks

ery




If tree does not have AVL property, do a trinode restructure at x, y, z

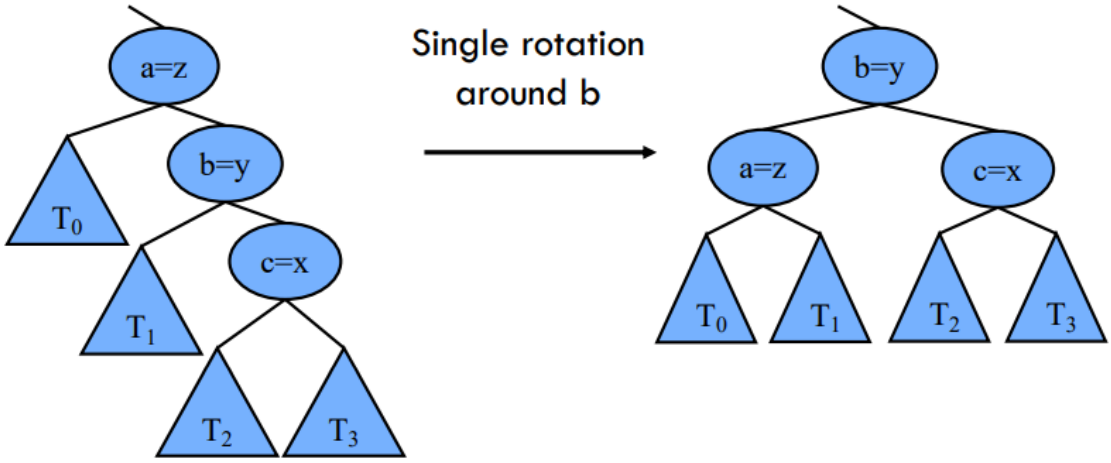



It can be argued that tree has AVL property after operation

Let x, y, z be nodes such that x is a child of y and y is a child of z.

Let a, b, c be the inorder listing of x, y, z

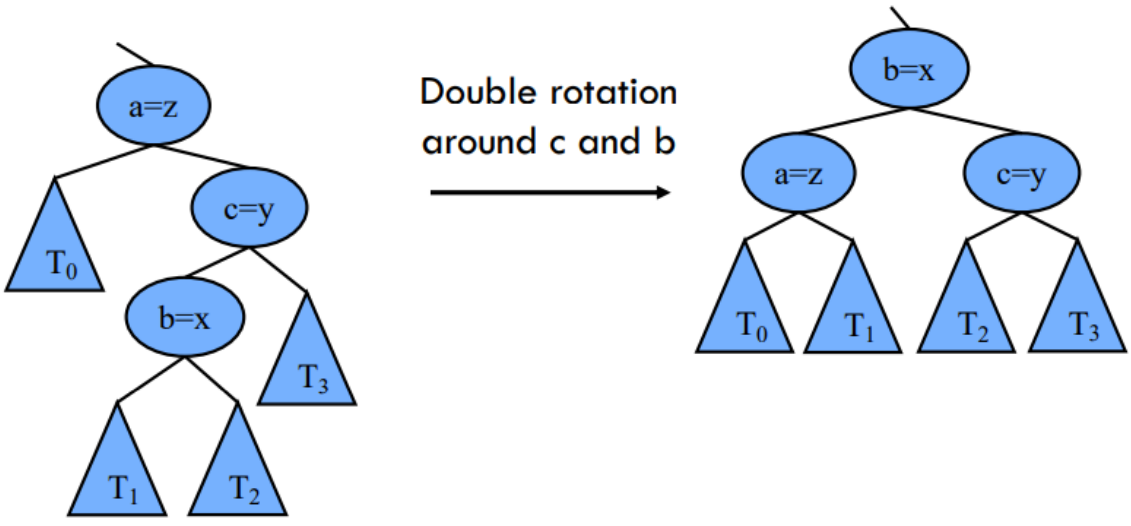
Perform the rotations so as to make b the topmost node of the three.




Let x, y, z be nodes such that x is a child of y and y is a child of z.

Let a, b, c be the inorder listing of x, y, z

Perform the rotations so as to make b the topmost node of the three.




## Binary tree Traversals

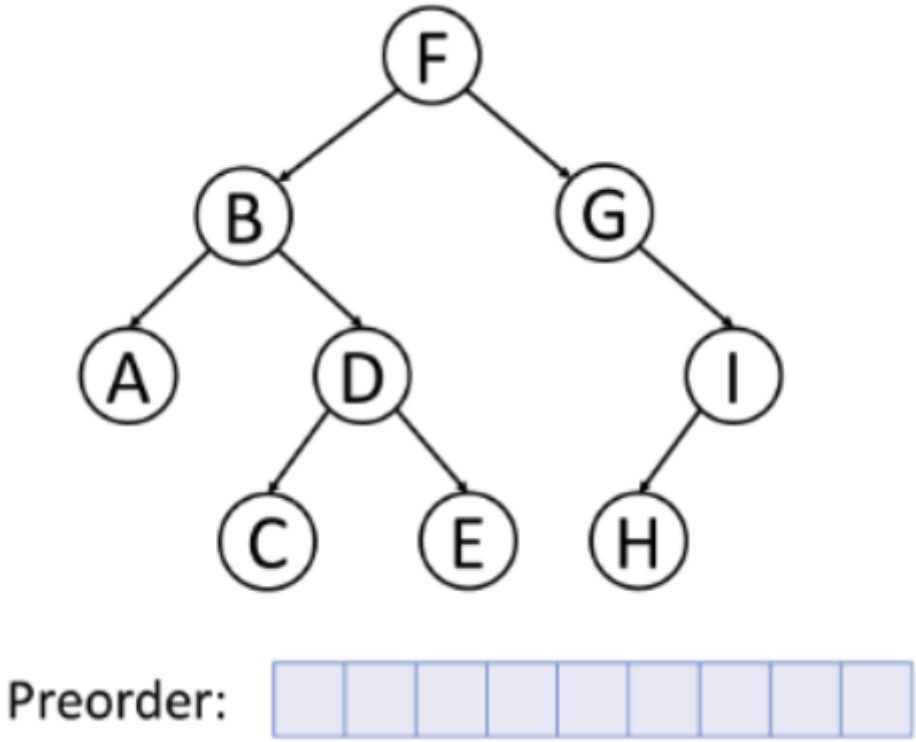
a binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.

where the left child is lesser than (or equal to in some cases) the parent and the right child is greater than the parent

```
def find_vertex(current_vertex, u):
        if current_vertex == u:
                return u

        if current_vertex == NULL
                find_vertex(current_vertex.right)
                find_vertex(current_vertex.left)
```

## Preorder Traversal Example

keeps following a path to attempt to going to the left untill it cannot left anymore



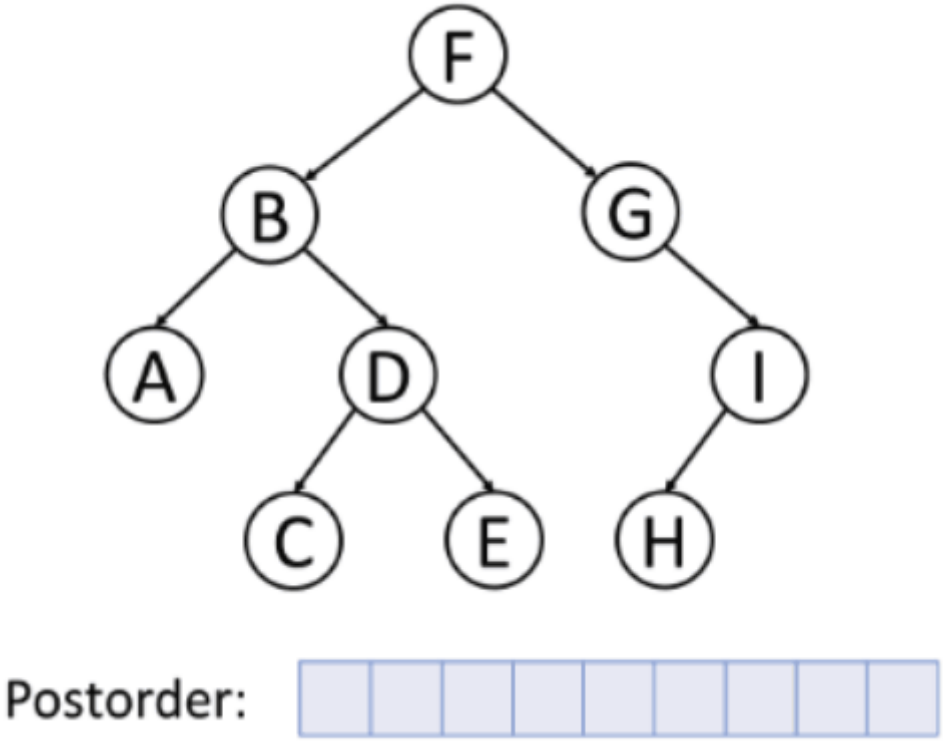

**explores the left subtree first, then the right goes to the left as much as possible**

- keeps following a path to attempt to going to the left untill it cannot left anymore
- in that case it would go back up to its parent and go to the right
- in the case where there is no more to be traversed it would go all the way up

```
def pre_order(current_node, visited):
        if current_node != NULL
                visited.add(current_node)
                visited.add(pre_order(current_node->left, visited))
                visited.add(pre_order(current_node->right, visited))
```

## Postorder Traversal

will always try to get the left most element if possible



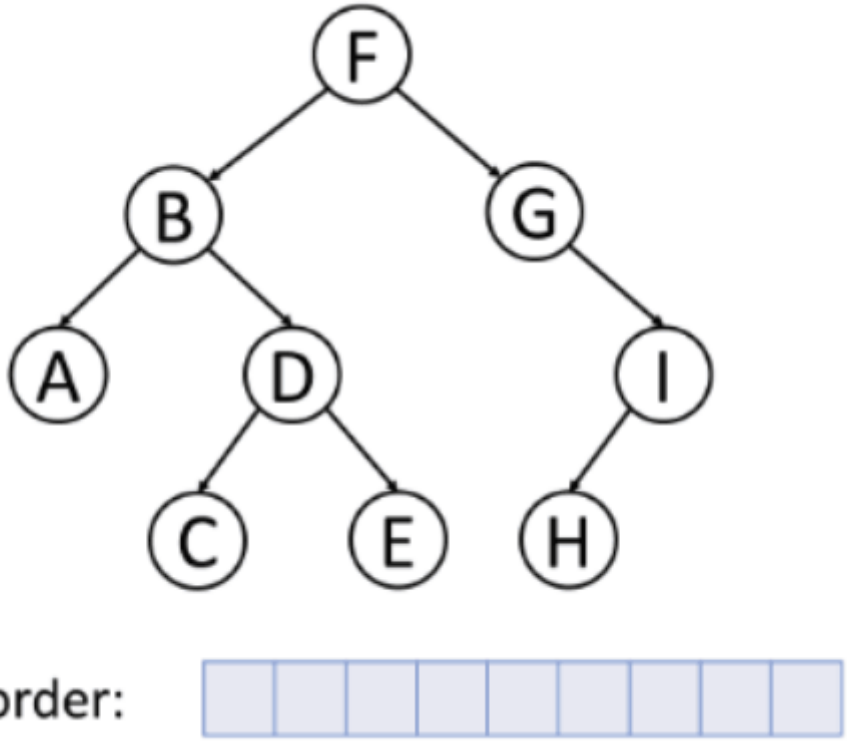




```
def post_order(current_node, visited):
        if current_node != NULL
                visited.add(post_order(current_node->left, visited))
                visited.add(post_order(current_node->right, visited))
                visited.add(current_node, visited)
```

## Inorder Traversal

left, parent, right.
this will be in ascending order in a binary tree




```
def inorder(current_node, visited):
        if current_node != NULL
                visited.add(inorder(current_node->left))
                visited.add(current_node)
                visited.add(inorder(current_node->right))
```

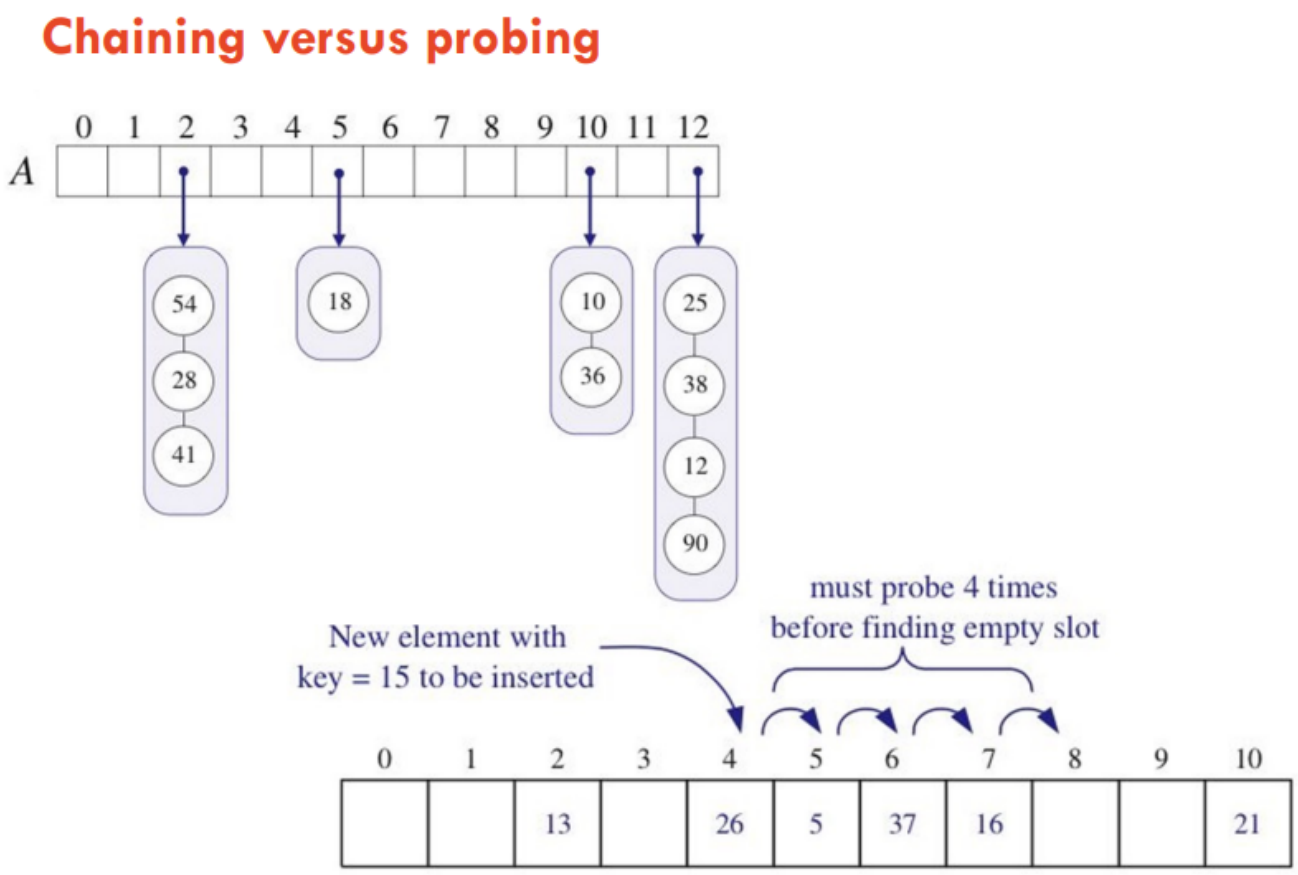
to find an element within a binary search tree, you compare the value of the node to the value you are searching for, if it is more than, then go to the right, if it is less than then go to the left

when inserting you do the same, and follow the path

# Hash Functions

Hash functions try to create a number as random as possible so it can insert it into an array with as little collision a possible. they wrap around using modulo. the array should be pretty big

linear chaining top, probing bottom




# Collisions

- **Seperate chaining** is when you have a collision you add another value to the same position worst case is O(n) if everything is put in the same
- **linear probing** is getting to the next empty spot to be inserted
- **cuckoo hashing** is using 2 hashtables and 2 hash functions (explained more later)

The time complexity for seperate chaining and linear probing is

| Function | Expected Case | Worst Case |
|---|---|---|
| Search | O(1) | O(n) |
| Insert | O(1) | O(n) |
| Delete | O(1) | O(n) |

because in the very small case where every single value is collided with each other, it would need to traverse through all the previous ones to get to where it wants.

(off the book but i think for seperate chaining you could keep all the collided values as a stack, and push a new one onto the stack, which would take O(1) instead)

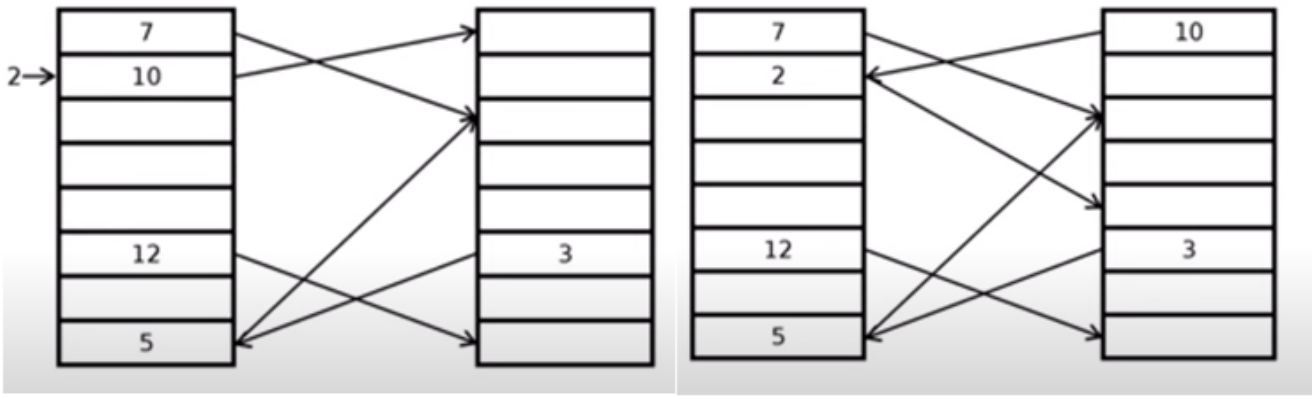
## Cuckoo hashing

often takes much longer than linear probing or seperate chaining but is guarenteed to have O(1) lookup and insertion

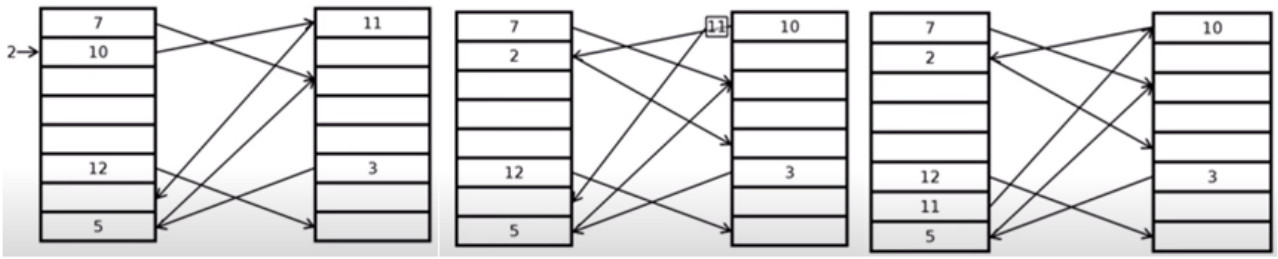
| Function | Expected Case | Worst Case |
|---|---|---|
| Search | O(1) | O(1) |
| Insert | O(1) | O(1) |
| Delete | O(1) | O(1) |

- Use two hash tables, T1 and T2, each of size N
- Use two hash functions, h1 and h2, for T1 and T2 respectively
- For an item with key k, there are only two possible places where we are allowed to store the item: T1[h1(k)] or T2[h2(k)]
- This restriction, simplifies lookup dramatically, while still allowing worst-case O(1) running time for get and remove.

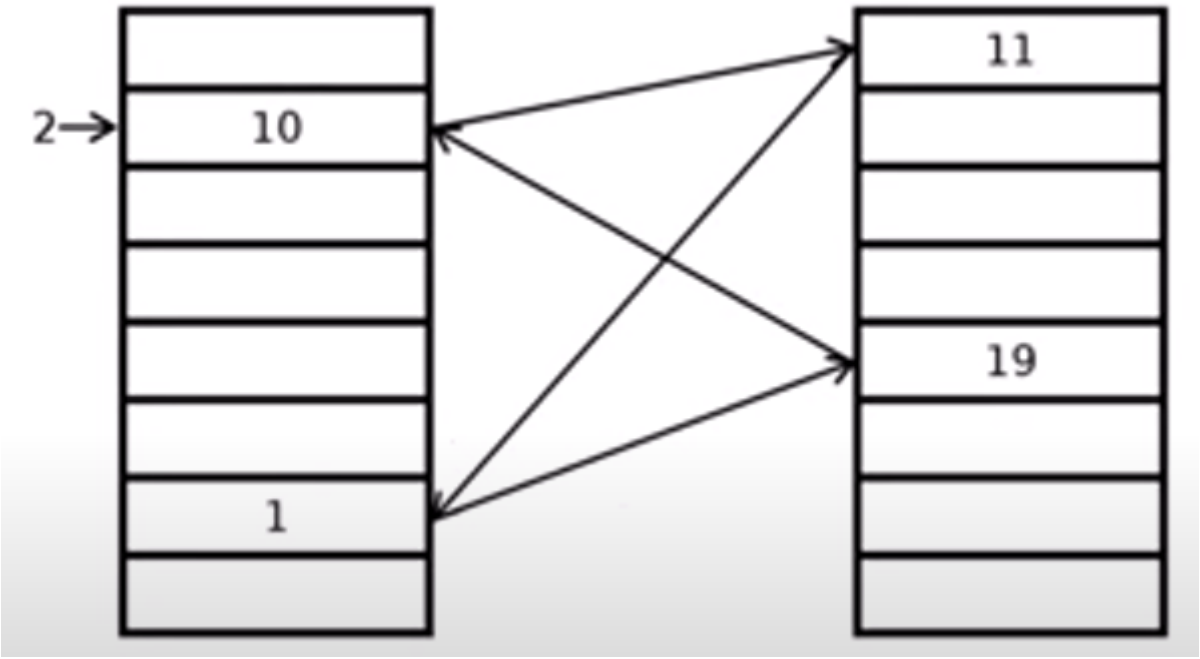
you can see that every value in T1, has a second hashing/pointer value in T2. so if there is a collilsion, it ejects the old values and put it in the T2 value. the new value is put in the T1




in this case, there is a function in 2 which points to 10 which goes to T2, which would point back to T1



Eviction cycle or infinite loops




if a loop is detected, it would rehash all of the functions? which is somehow still O(1)

to search for a value 10, you would search where it is in the first hash table, if its not there you go to the next hashtable

# Graphs

| Term | Definition |
| --- | --- |
| endpoints | the vertexes the edges are connecting |
| incident | the edges that are connected to a vertex |
| adjacent vertexes/neighbour | vertexes that are connected to each other |
| degree | number of edges a vertex has |
| parallel edges | when there are 2 edges that connect to the same vertexes |
| self loop | when it loops on itself |
| simple | no parallel or self loops |
| cut edge | In a connected graph G=(V, E), we say that an edge (u, v) in E is a cut edge if (V, E \ {(u, v)}) is not connected. Essentially if you remove that edge it wont be connected anymore |

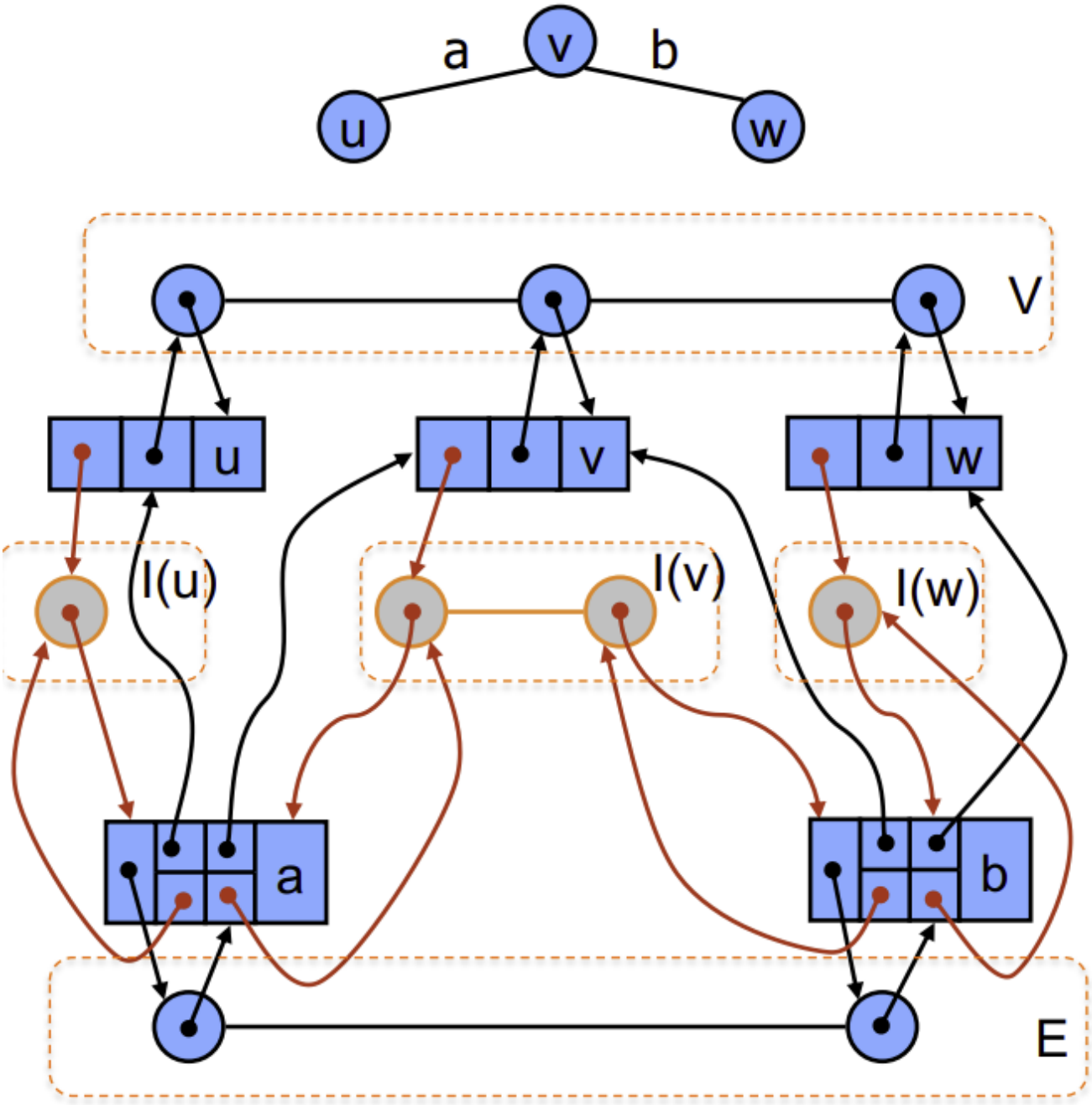
**directed graphs**

| Term | Definition |
| --- | --- |

| Term | Definition |
| --- | --- |
| Tail | where the edge goes to (the point of the edge) |
| Head | where the edge starts |
| Out-degree | is # of edges out of a vertex |
| In-degree | # of edges into a vertex |
| Parallel edges | share tail and head (have to have the arrows pointed at the same way) |

# Implementation

## Adjacency List



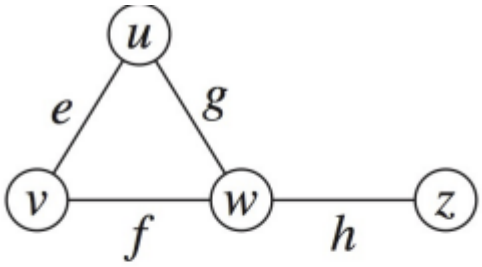
Additionally each vertex keeps a sequence of edges incident on it. Edge objects keep reference to their position in the incidence sequence of its endpoints

```
struct Node {
        *void value;
        *Node next;
}

struct vertex{
        void* value;
        Node* list_of_incident //a linked list of edges
}

struct edge{
        //not actual head and tails because it is not directed
        vertex* head;
        vertex* tail;
}
```

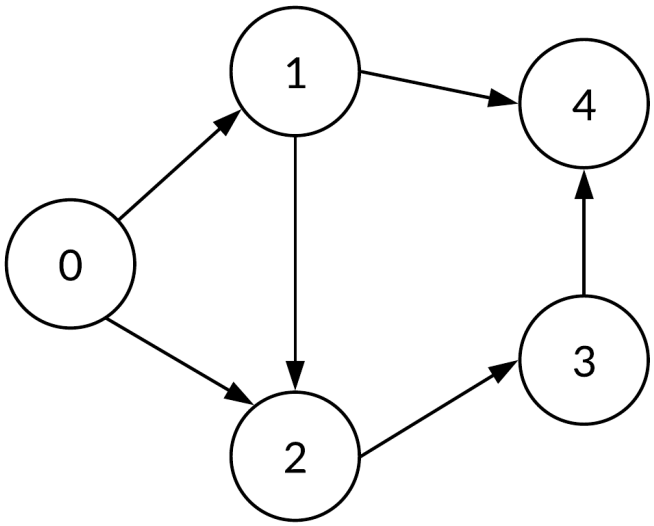
## Adjacency Matrix

A 2D array that stores the edges as such

the rows represent all of the edges going out of the vertex
columns represent all edges coming into the vertex




| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| u → 0 | | e | g | |
| v → 1 | e | | f | |
| w → 2 | g | f | | h |
| z → 3 | | | h | |




Adjacency Matrix

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 2 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 |
| 4 | 0 | 0 | 0 | 0 | 0 |

| ▪ $n$ vertices, $m$ edges<br>▪ no parallel edges<br>▪ no self-loops | Edge List | Adjacency List | Adjacency Matrix |
|---|---|---|---|
| Space | $O(n + m)$ | $O(n + m)$ | $O(n^2)$ |
| incidentEdges($v$) | $O(m)$ | $O(\deg(v))$ | $O(n)$ |
| getEdge($u$, $v$) | $O(m)$ | $O(\min(\deg(u), \deg(v)))$ | $O(1)$ |
| insertVertex($x$) | $O(1)$ | $O(1)$ | $O(n^2)$ |
| insertEdge($u$, $v$, $x$) | $O(1)$ | $O(1)$ | $O(1)$ |
| removeVertex($v$) | $O(m)$ | $O(\deg(v))$ | $O(n^2)$ |
| removeEdge($e$) | $O(1)$ | $O(1)$ | $O(1)$ |

# Graph Traversals

time complexity for most traversals are O(V + E). where v is the number of vertexes and e is the number of edges

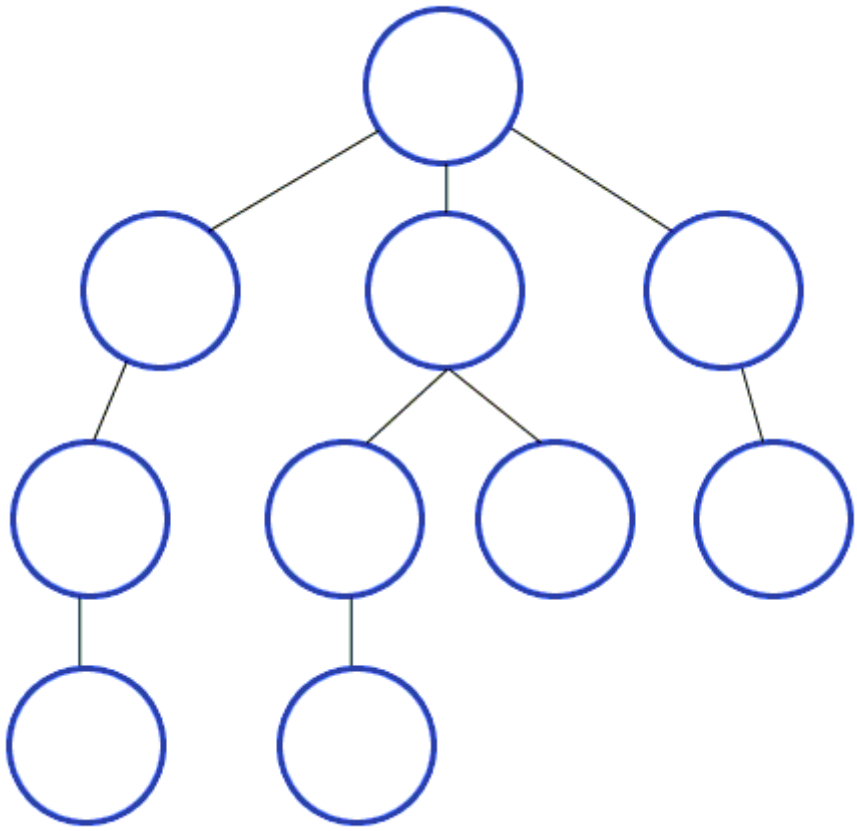
assume you are given the adjacency list representation

A systematic and structured way of visiting all the vertices and all the edges of a graph

## Depth First Search (DFS)

attempts to go as far into the tree as possible following a path, and backtracking when there are no more edges going down that path

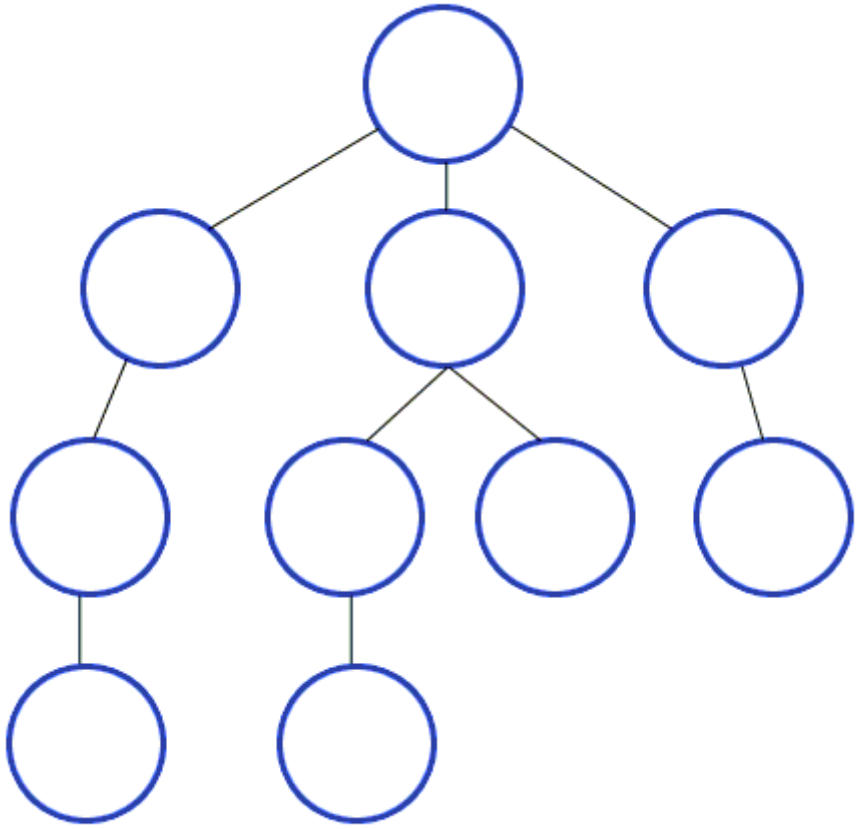
if done correctly, it should return a tree representation of the graph



```
def DFS(s):
    stack = [s]
    visited = set()
    # when current vertex is t then stop exploring
    while len(stack) > 0:
        # takes the top value of the stack and explores it
        current_vertex = stack.pop()

        # if it has already been explored then go to the next value in the
stack
        if current_vertex in visited:
            continue

        visited.add(current_vertex)
        for edge in current_vertex.edges:
            stack.append(edge)

    return path if path[len(path) - 1] == t else None
```

## Breadth First Search (BFS)

Explores the graph through layers, explores each level and goes one deeper until they have all been exhausted.



# Weighted Graphs

## Dijkstra's algorithm

**calculates the shortest path to every node from a starting node**

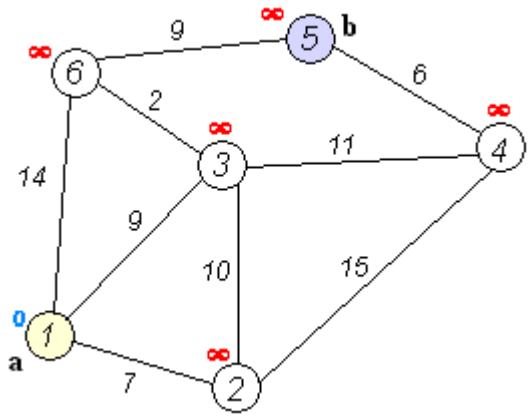
The base of the algorithm is that it gives every vertex a value on how much it "costs" to get there.

initially set every node to have a cost of infinite.
set the starting node as 0

it follows a path through the actual graph to decide where to go next and calculate how much it costs to get there.

as you traverse, you increment the current weight the vertex has, and then add the new edge weight to see how much it takes to get to the NEW vertex

if the current weight is less than the new weight, then change the parent




```
def Dijkstra(G, w, s):

  # initialize algorithm
  for v in V do                          ┐
    D[v] ← ∞                             │
    parent[v] ← ∅                        ├ O(n)
  D[s] ← 0                               ┘
  Q ← new priority queue for { (v, D[v]) : v in V }

  # iteratively add vertices to S
  while Q is not empty do
    u ← Q.remove_min()                   ┐
    for z in G.neighbors(u) do           │
      if D[u] + w[u, z] < D[z] then      │
        D[z] ← D[u] + w[u,z]             ├ O(deg(u)) for each u in V
        Q.update_priority(z, D[z])       │ plus update_priority work
        parent[z] ← u                    ┘
  return D, parent
```

**To calculate the actual smallest weight *OVERALL* then you use these algorithms**

Using a heap for PQ, Dijkstra runs in O(m log n) tim

if you use a fibonnacci heap for the priority queue you can get the decrease key and updating priority in O(1) expected.

O(m + n log n)

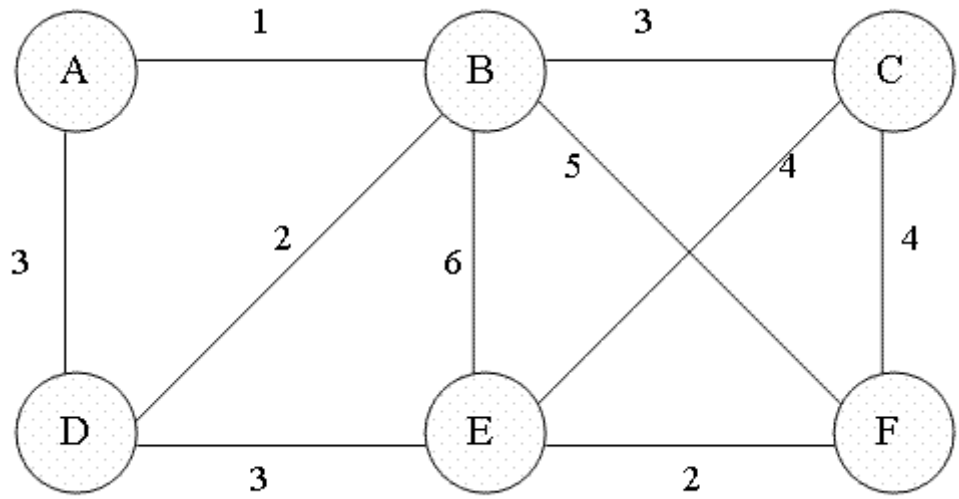
## Smallest weight overall

### Prim's algorithm

pick an arbitrary edge
then pick the smallest path to any vertex
if a vertex has already been encountered then there is no need to pick it

IT GOES DOWN A PATH OF VERTICIES AND DECIDES WHICH ONE IT SHOULD GO TO ***NEXT***

```
def prim(G, c):
  u ← arbitrary vertex in V
  S ← { u }
  T ← ∅
  while |S| < |V| do
    (u, v) ← min cost edge s.t. u in S and v not in S
    add (u, v) to T
    add v to S
  return T
```




similar to dijkstra

Using a heap for PQ, Dijkstra runs in O(m log n) tim

if you use a fibonnacci heap for the priority queue you can get the decrease key and updating priority in O(1) expected.

O(m + n log n)

## Kruskal's algorithm

always pick the smallest edge
keep adding edges untill all verticies are added to the MST
if there is an edge with the vertex already added down add it

IT PICKS THE SMALLEST EDGE WEIGHT REGARDLES OF WHERE THE ALGORITHM IS, IT DOESNT GO DOWN A PATH IT JUST KEEPS POPPING ANY EDGE THAT IS MINIMUM

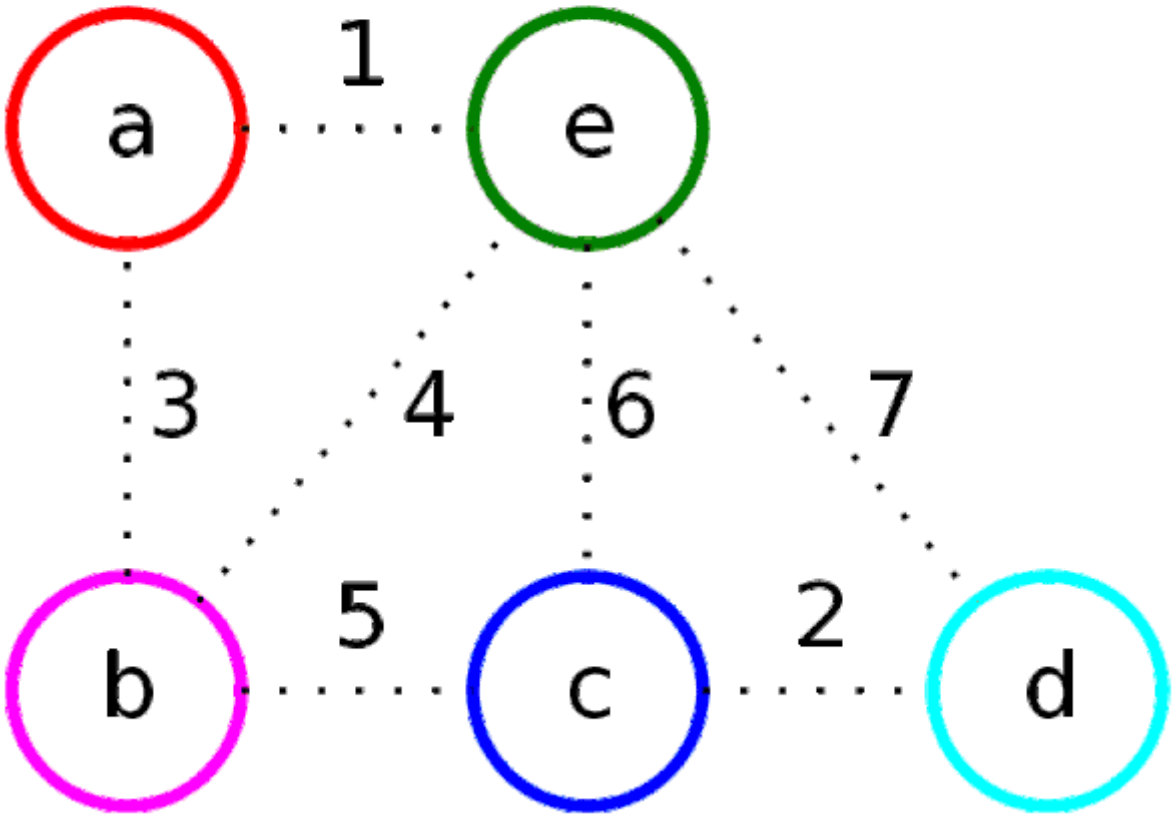
```
def Kruskal(G,c):

  sort E in increasing c-value
  answer ← [ ]
  comp ← make_sets(V)
  for (u,v) in E do
    if comp.find(u) ≠ comp.find(v) then
      answer.append( (u,v) )
      comp.union(u, v)
  return answer
```

Union find operations:

- make_sets(A) : one call with |A| = |V|
- find(a) : 2m calls
- union(a,b) : n-1 calls



| Edge | ab | ae | bc | be | cd | ed | ec |
|---|---|---|---|---|---|---|---|
| Weight | 3 | 1 | 5 | 4 | 2 | 7 | 6 |



Sorting edges takes O(m log m) time

One option is to run DFS in each iteration to see if the number of connected components stays the same. This leads to O(m n) time for the main loop

Kruskal's algorithm would run in O(n^2) time after the edge weights are sorted.

# Minimum Spanning Trees

Minimum spanning trees are trees that have the least amount of edges possible

## Properties of an MST

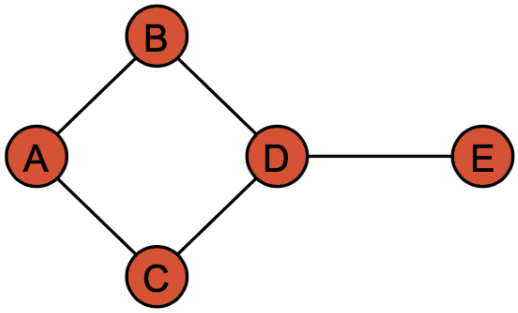
- **Cut property**
  - if one single edge is removed from the MST then the tree must no longer be connected
  - **EXPLAINATION:** the minimum spanning tree should really only need to take 1 direct path to every node. so if one thing is removed from this path, then there is no path to other nodes
- **Cycle property**
  - if a tree has a cycle it is not a minimum spanning tree
  - **EXPLAINATION:** if there is a cycle this means there is more than 1 way to get to a certain node, and if this is true that means it doesnt take the most efficient route (doesnt use the least number of edges to get to the tree)

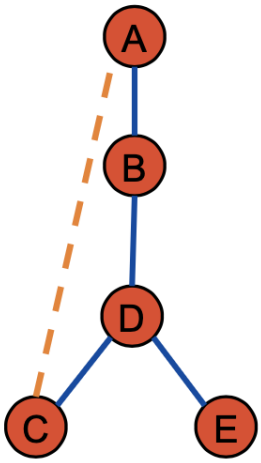
## Identifying cut edges in O(n+m) time

Compute a DFS tree of the input graph G=(V, E)

For every u in V, compute level[u], its level in the DFS tree

For every vertex v compute the highest level that we can reach by taking DFS edges down the tree and then one back edge up. Call this down_and_up[v]




| | level | d&u |
|---|---|---|
| A | 0 | 0 |
| B | 1 | 0 |
| C | 3 | 0 |
| D | 2 | 0 |
| E | 3 | 3 |

For down&up, we're trying to compute the highest level a vertex can reach by going down 0 or more edges and then coming back up using a single edge. For A, B and D, they can all traverse down to C and take that edge up to bring it back to level 0. In the case of C, since it's at the bottom, we can't move any further down but what we can do is take that back edge up to A, bringing it to the highest level :)

# Greedy Algorithm

A greedy algorithm constructs the solution in a greedy fashion, so after every element it processes, it makes some irrevocable choice (typically add or not add the processed element or something similar to that).

If you build a data structure, you're likely not making any choices while processing the elements, instead delaying the choices and the construction of the solution until the very end of the algorithm.

That's indeed basically the case. A greedy algorithm makes irrevocable decisions while creating the output. If you're creating a binary search tree from the input and then using that, you aren't doing this (as you aren't making any decisions while processing the input) and I wouldn't consider that approach greedy.

when a solution updates is answers when it considers more data then it is not a greedy algorithm

This is not really a greedy algorithm, as you sort your input, then iterate through your whole input and perform a bunch of computations and updates to your towers before finally producing them.

A greedy solution makes decisions without much knowledge of the broader data, and so should iterate through the intervals and make a decision on whether or not to open a tower at each step. Greedy algorithms are also much simpler (for a problem like this).

to be greedy it would need to

- loop through all the input and would always need to make the best decision based on that input
- those decisions cant be updated when you encouter new data

**Picks the largest**
**Picks the smallest weight**
**Picks best ratio**

# Divide and Conquer

1. Divide If it is a base case, solve directly, otherwise break up the problem into several parts.
2. Recur/Delegate Recursively solve each part [each sub-problem].
3. Conquer Combine the solutions of each part into the overall solution

essentially to do it is to assume that the "recursion fairy" (the recursion fairy works the same as the inductive hypothesis, where you have to take a leap of faith and believe it works before you finish) will already give you exactly what you need. then you decide what to do with that.

so assume you are at the very last step of the algorithm, what do you need to do to return the FINAL answer.

## Proving Correctness

**Inductive hypothesis:** what is the leap of faith you are taking? what are you assuming the "recursion fairy" returns

**Base Case:** the absolute last thing you are breaking the array down to. the smallest it gets

**Induction:**

- assume the inductive hypothesis for an arbitrary number/size k, then prove it for number/size k + 1
- because of the inductive hypothesis, every thing the algorithm returns will be correct (e.g Splitting the array in half gives us two array of size at most k, so by IH those are sorted correctly)
- because you run the inductive hypothesis smaller ones and those have already worked if you were to apply those to the result you currently have to create something bigger it should still work?

# Master Theorum

## Master Theorem

Let f(n) and T(n) be defined as follows:

$$T(n) = \begin{cases} a\,T(n/b) + f(n) & \text{for } n \geq d \\ c & \text{for } n < d \end{cases}$$

Depending on a, b and f(n) the recurrence solves to:

1. if $f(n) = O(n^{\log_b a - \varepsilon})$ for $\varepsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$,
2. if $f(n) = \Theta(n^{\log_b a} \log^k n)$ for $k \geq 0$ then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$,
3. if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ and $a\,f(n/b) \leq \delta\, f(n)$ for $\varepsilon > 0$ and $\delta < 1$ then $T(n) = \Theta(f(n))$,

$T(n) = aT(n/b) + O(1)$

$T(n) = T(n/2) + O(1)$

$a =$
$b =$
$f(n) =$

first calculate $n^{\log_b a}$ to find if it is > than f(n) = to f(n) or < than f(n), and thus find the case
$n^{\log_b a} =$

**MASTER THEORUM**

**Case 1:** $f(n) < n^{\log_b a}$ AKA $f(n) = O(n^{\log_b a - \epsilon})$ for $\epsilon > 0$
$T(n) = \Theta(n^{\log_b a})$

**Case 2:** $f(n) = n^{\log_b a} \cdot \log^k n$ AKA $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$ for $k \geq 0$
$T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1}(n))$

**Case 3:** $f(n) > n^{\log_b a}$ AKA $f(n) = \Omega(n^{\log_b a + \epsilon})$ and $a \cdot f(n/b) \leq \delta f(n)$ for $\epsilon > 0$ and $\delta < 1$ then
$T(n) = \Theta(f(n))$

We have case -insert-

which means:

$T(n) = O(n)$