**Sorting**

1. **Insertion sort**
2. Algorithm

A = [5, 2, 4, 6, 1, 3]

// Loop from second position, and compare to sorted array from 0 to current position, if smaller, then loop until find the position that current item is larger or first position. Insert item to the sorted array

function insertionSort(a) {

for (var i = 1; i < a.length; i++) {

var key = a[i]

var j = i - 1

while (j >= 0 && a[j] > key) {

a[j + 1] = a[j]

j = j - 1

}

a[j + 1] = key

}

return a

}

console.log(insertionSort(A))

1. Analysis

The time taken by an algorithm grows with the size of the input

Worst case running time of θ(n2)

1. **Merge sort ( divide and conquer)**
2. algorithm

// divide

A = [5, 2, 4, 7, 1, 3, 2, 6]

function mergeSort(a) {

if(a.length < 2) {

return a

}

var middle = Math.floor(a.length/2)

var leftArr = a.slice(0, middle)

var rightArr = a.slice(middle, a.length)

console.log(leftArr)

console.log(rightArr)

return merge(mergeSort(leftArr), mergeSort(rightArr))

}

//Conquere

function merge(L, R) {

var result = []

while(L.length && R.length) {

if(L[0] <= R[0]) {

result.push(L.shift())

} else {

result.push(R.shift())

}

}

while(L.length) {

result.push(L.shift())

}

while(R.length) {

result.push(R.shift())

}

return result

}

console.log(mergeSort(A))

1. analysis

Worst case running time of θ(nlgn)

1. **Bubble sort**

Simply swapping 2 elements of array. It’s no efficiency

1. **Heap sort**
2. Algorithm

* Build max\_heap\_heap. ( create a function to maintain max-heap properties)
* Then get a max item at position 0, swap position to the last pos and decremeting the array size
* Repeat the process for the size n-2 down to 1. We get the sorted array

var arr = [ 9, 10, 2, 1, 5, 4, 3, 6, 8, 7, 13 ];

var arraylength = 0

function max\_heapify(a, i) {

var left = i\*2 + 1;

var right = i\*2 + 2;

var largest = i;

if (left < arraylength && a[left] > a[largest]) {

largest = left;

}

if (right < arraylength && a[right] > a[largest]) {

largest = right;

}

if (i != largest) {

swap(a, i, largest);

max\_heapify(a, largest)

}

}

function swap(input, index\_A, index\_B) {

var temp = input[index\_A];

input[index\_A] = input[index\_B];

input[index\_B] = temp;

}

function build\_max\_heap(a) {

arraylength = a.length

for(i = Math.floor(a.length/2); i >= 0; i--) {

max\_heapify(a, i)

}

}

function heapSort(a) {

build\_max\_heap(a)

for (var i = a.length - 1; i > 0; i--) {

swap(a, 0, i);

arraylength--;

max\_heapify(a, 0);

}

}

heapSort(arr);

console.log(arr)

1. Analysis

The HEAPSORT procedure, which runs in O(nlgn)

1. **QuickSort**
2. Algorithm

Divide: divide the array into 2 subarray a[p, q-1] and a[q, r] such that element of a[p, q – 1] is less than or equal to a[q], and element of a[q + 1, r] greater than a[q]. Compute the index q

Conquer: repeat sort 2 subarrays by recursive call quicksort

function quickSort(a, p, r) {

if(p < r) {

q = partition(a, p, r)

quickSort(a,p, q - 1)

quickSort(a,q + 1, r)

}

}

function partition(a, p ,r) {

var x = a[r]

var i = p - 1

for (var j = p; j < r; j++) {

if(a[j] <= x){

i = i + 1

swap(a, i, j)

}

}

swap(a, i+ 1, r)

return i + 1

}

quickSort(arr, 0, arr.length - 1)

1. Analysis

Worst-case partitioning : when 1 subarray with n-1 elements and one with 0 elements. The partition cost θ(n) time. With array size 0, the cost is T(0) = θ(1). So recursive running time is

T(n) = T(n-1) + T(0) + θ(n)

Solution is T(n) = θ(n2)

Best-case partitioning: 2 subarrays and each of size no more than n/2

T(n) = 2T(n/2) + θ(n)

Solution is T(n) = θ(nlgn)

**Data Structure**

1. Stacks and queues
2. Stacks: last in, first out.
3. Queues: first in, first out.
4. Linked list

It’s data structre in which the objects are arranged in a linear order. Linked list is a sequence of links which contains items. Each link contains a connection to another link.

1. Singly node

function LinkedList(){

this.head = null;

}

LinkedList.prototype.push= function(val) {

var node = {

value: val,

next: null

}

if(!this.head) {

this.head = node

} else{

var current = this.head

while(current.next) {

current = current.next

}

current.next = node

}

}

LinkedList.prototype.printList = function() {

if(!this.head) {

console.log('empty list')

} else{

var current = this.head

while(current) {

console.log(current.value)

current = current.next

}

}

}

LinkedList.prototype.insertAfter = function(pre\_node, new\_data) {

if(pre\_node === null) {

console.log('given previous node cannot be null')

return

} else{

var node = {

value = new\_data

next = pre\_node.next

}

pre\_node.next = node

}

}

LinkedList.prototype.insertAfter = function(pre\_node, new\_data) {

if(pre\_node === null) {

console.log('given previous node cannot be null')

return

} else{

var node = {

value = new\_data,

next = pre\_node.next

}

pre\_node.next = node

}

}

LinkedList.prototype.insertFirst = function (new\_data) {

var node = {

value = new\_data,

next = null

}

if(!this.head) {

this.head = node

return

}else {

node.next = this.head

this.head = node

}

}

LinkedList.prototype.deleteNode = function (val) {

var current = this.head;

//case-1

if(current.value == val){

this.head = current.next;

}

else{

var previous = current;

while(current.next){

//case-3

if(current.value == val){

previous.next = current.next;

break;

}

previous = current;

current = current.next;

}

//case -2

if(current.value == val){

previous.next == null;

}

}

}

var lls = new LinkedList();

lls.push(2)

lls.push(3)

lls.push(4)

//traver list

lls.printList()

1. Doubly node

A doubly linked list contains an extra pointer, typically called previous pointer, together with next pointer and data which are there in singly linked list.

function DoubleLinkedList() {

this.head = null

}

DoubleLinkedList.prototype.append = function(val){

var node = {

value: val,

pre: null,

next: null

}

if(!this.head) {

this.head = node

} else {

var current = this.head

while(current && current.next) {

current = current.next

}

current.next = node

node.pre = current

}

}

DoubleLinkedList.prototype.push = function(val){

var node = {

value: val,

pre: null,

next: null

}

if(!this.head) {

this.head = node

} else {

var current = this.head

current.pre = node

node.next = current

this.head = node

}

}

DoubleLinkedList.prototype.insertAfter = function(pre\_node, val){

var node = {

value: val,

pre: null,

next: null

}

if(pre\_node === null) {

console.log('not a node')

}else {

node.next = pre\_node.next

node.pre = pre\_node

pre\_node.next = node

if(node.next != null) {

node.next.pre = node

}

}

}

DoubleLinkedList.prototype.printList = function(){

if(!this.head) {

console.log('empty list')

} else {

var current = this.head

while(current) {

console.log(current.pre)

current = current.next

}

}

}

var dll = new DoubleLinkedList();

dll.append(2);

dll.append(3);

dll.append(4);

dll.append(5);

dll.printList()

1. Rooted tree (binary tree)

In binary tree, each node has pointer parent, left, right to pointer to other nodes respectively. If x.p = nil, then x is the root. If node x has no left child or right child then x.left or x.right = nil. The root of the entire tree T is pointed to by the attribute T.root. If T.root = nil, then tree empty

1. **Hash Table**

A hash table is a data structure that maps keys to values for highly efficient lookup. In a very simple implementation of hash table, we use an array of linked list and a hash code function.

To insert a key and value:

* Compute the key’s hash code
* Then, map the hash code to an index in the array. This could be done with something like hash(key) % array\_length.
* At this index, there is a llinked list of key and values. Store the key and value in this index.

1. **Linked list:**

A linked list is a data structure that represents a sequence of nodes. In a singly linked list, each node points to the next node in the linked list. A doubly linked list gives each node pointers to both the next and previous node.

Compare to array:

* Pros: you can add and remove items from the beginning of the list in constant time.
* Cons: a linked list does not provide constant time access to a particular “index” within the list.

Remember to use 2 pointers if the question is to find the middle of linked list or to travel the whole list.

1. **Stacks and queues:**
2. Stack: is a data structure that stores the data in a stack rather than in an array.

A stack uses LIFO (last-in first out) ordering.

Unlike an array, a stack does not offer constant-time access to the ith item. However, it does allow constant-time adds and removes, as it doesn’t require shifting elements around.

* Implementing stack:

var stack = []

stack.push

stack.pop

or using object

var Stack = function () {

this.count = 0

this.storage = {}

}

Stack.prototype.push(val) {

this.storage.[this.count]

this.count ++

}

Stack.prototype.pop() {

If(this.count === 0 {  
 return -1

} else {

Var I = this.storage.[this.count]

Delete this.storage.[this.count]

this.count –

return i

}

}

Stack.prototype.isEmpty

1. Queue

A queue implements FIFO ordering.

Operations:

* Add: add an item to the end of the list
* Remove: remove the first item in the list
* Peek(): return the top of the queue
* isEmpty(): return true if and only if the queue is empty

1. **Trees and Graphs**
2. Trees:

* Types of trees:

+ Each tree has a root node.

+ The root node has zero or more child nodes

+ Each child node has zero or more child nodes, and so on

The tree cannot contain cycles. The nodes may or may not be in particular order, they could have any data type as value, and they may or may not have links back to their parent node

Definition of Node:

Node = function () {

This.value

This.child = []

}

1. Trees and Binary Trees

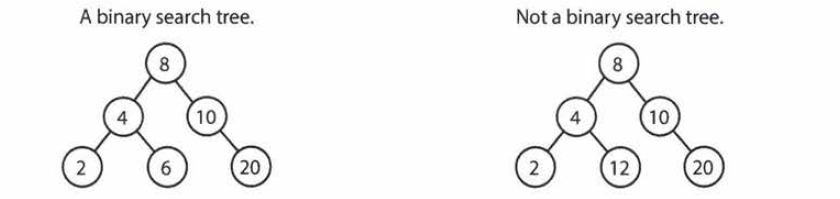
A binary tree is a tree in which each node has up to two children. Not all trees are binary trees.

A node is called a “leaf” node if it has no children.

1. Binary Tree vs Binary Search Tree

A binary search tree is a binary tree in which every node fits a specific ordering property: all left descendants <= n < all right descendants. This must be true for each node n. The duplicate values will be on the right or can be on either side or don’t exist.

Note that this inequality must be true for all of a node’s descendants, not just its immediate children. The tree on the right is not a binary tree since 12 is to the left of 8

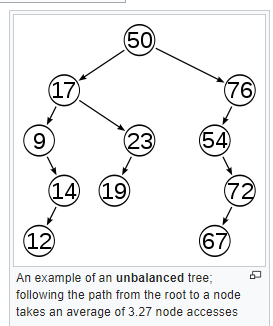
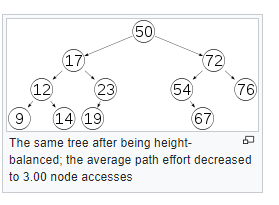


Once again, a binary search tree imposes the condition that, for each node, its left descendants are less than or equal to the current node, which is less than the right descendants.

1. Balanced and Unbalanced

Note that balancing a tree does not mean the left and right subtrees are exactly the same size. Balanced tree really means something more like “not terribly imbalanced”. It’s balanced enough to ensure O(log n) times for insert and find.

a B-tree is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time.



It’s desirable to keep the height small. A binary tree with height h can contain at most 20+ 21 + …+ 2h = 2h+1 – 1 nodes.

So it follows that for a tree with n nodes and height h

N <= 2h+1- 1

* h >= log2n

1. Complete Binary Trees

A complete binary tree is a binary tree in which every level of the tree is fully filled, excepts for perhaps the last level. To the extent that the last lever is filled, it’s filled left to right.



1. Full Binary Tree

A full binary tree is a binary tree in which every node has either zero or two children. That’s, no nodes have only one child 

1. Perfect binary tree

A perfect binary tree is one that is both full and complete. All leaf nodes will be at the same level, and this level has the maximum number of nodes.

1. Binary tree Traversal
2. In-order Traversal

In-order traversal means to visit the left branch, then the current node, and finally the right branch

Function inOrderTraversal (treenode node) {

If(node != null) {

inOrderTraversal(node.left);

visit(node)

inOrderTraversal(node.right);

}

}

1. Pre-order Traversal

Pre-order traversal visits the current node before its child nodes

Function preOrderTraversal (treenode Node) {

If(node != null) {

Visit(node)

preOrderTraversal(node.left);

preOrderTraversal(node.right);

}

}

In a pre-order traversal, the root is always the first node visited.

1. Post-Order traversal

Post-order traversal visits the current node after its child node

Function postOrderTraversal (treenode Node) {

If(node != null) {

postOrderTraversal (node.left);

postOrderTraversal (node.right);

Visit(node)

}

}

1. Binary Heaps (min heaps and Max heaps)

A min heap Is a complete binary tree ( that’s totally filled other than the rightmost elements on the last level) where each node is smaller than its children. The root, therefore, is the minimum element in the tree

2 Operations:

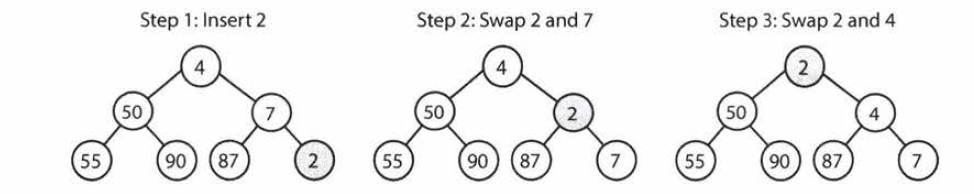
+ Insert

+ extract\_min

* Insert:

When we insert into a min-heap, we always start by inserting the element at the bottom. We insert at the rightmost spot so as to maintain the complete tree property

Then, we fix the tree by swapping the new element with its parent, until we find an appropriate spot for the element.



This takes O(log n) time, where n is the number of nodes in the heap

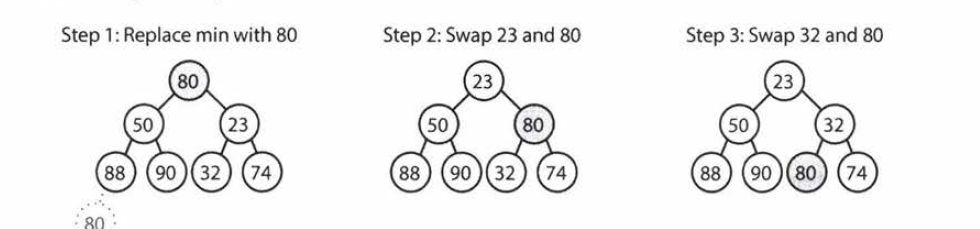
* Extract Minimum Element

The root is the minimum element. How to remove it?

First, remove the root and swap it with the last element of the heap (the bottommost, rightmost element).

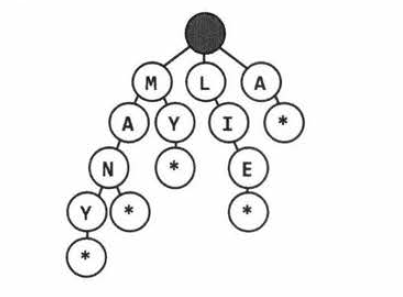
Then, we bubble down this element, swapping it with one of its children until the min-heap property is restored.

Always swap the smallest one to maintain the main-heap ordering.



1. Tries (prefix Trees)

A trie is a variant of an n-ary tree in which characters are stored at each node. Each path down the tree may represent a word.

The \* node (null nodes) are often used to indicate complete words. For example, the fact that there is a \* node under MANY indicates that MANY is a complete word.  
The implementation of these \* nodes might be a special type of child or we could use just a Boolean flag terminates within parent node

A trie is used to store the entire (English) language for quick prefix lookups.

1. Graph

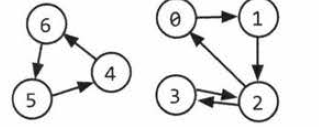
A tree is actually a type of graph, but not all graphs are trees. A tree is a connected graph without cycles.

A graph is a collection of nodes with edges between them

A graph can be either directed (1 way street) or undirected (2 way street)

A graph might consist of multiple isolated subgraphs. If there is a path between every pair of vertices, it’s called a “connected graph”

The graph can also have cycles. An acyclic graph is one without cycles



In programing, there are two common ways to represent a graph

* Adjacency List

Every vertex( or node) stores a list of adjacent vertices. In an undirected graph, an edge like (a, b) would be stored twince

Graph = function () {

Var nodes = []

}

Node = function {

Var string name

Var nodeChildren = []

}

The graph class is used because, unlike in a tree, you can’t necessarily reach all the nodes from a single node.

You don’t necessarily need any additional classes to represent a graph. An array of list can store the adjacency list.



* Adjacency Matrices

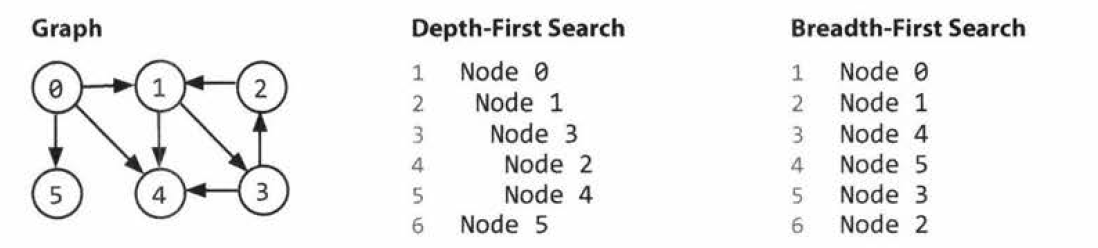
An adjacency matrix is an NxN Boolean matrix ( where N is the number of nodes), where a true value at matrix[i][j] indicated an edge from node I to node j.

In an undirected graph, an adjacency matric will be symmetric.



The same graph algorithms that are used on adjacency list can be performed with adjacency matrices, but they may be somewhat less efficient. In the adjacency list representation, you can easily iterate through the neighbors of a node. In the adjacency matrix representation, you will need to iterate through all the nodes to identify a node’s neighbors.

1. Graph Search
2. ways to search a graph are depth-first search and breadth-first search
3. Depth-first search (DFS), we start at the root and explore each branch completely before moving on to the next branch. That’s, we go deep first before we go wide
4. Breadth-first search (BFS), we start at the root and explore neighbor before going on to any of their children. That’s, we go wide before we go deep



Breadth-first search and DFS tent to be used in different scenarios.

DFS is often preferred if we want to visit every node in the graph. DFS is a bit simpler.

BFS is preferred if we want to find the shortest path between two nodes. BFS is generally better.

* DFS, we visit a node a and then iterate through each of a’s neighbors. When visiting a node b that is a neighbor of a we visit all of b’s neighbors before going on to a’s other neighbors. That’s, a exhaustively searches b’s branch before any of its other neighbors

Search = function (node) {

If(node == null) {

return

}

Visit (node)

node.visited = true

for each( node n in node.adjacent) {

if(n.visisted = false)

search (n)

}

}

* BFS is a bit less intuitive, and BFS is not recursive. It uses a queue. In BFS, node a visits each of a’s neighbors before visiting any of their neighbors. Thinking of this as searching level by level out from a. An iterative solution involving a queue usually works best.

Search = function (node root) {

Queue queue = new queue()

Root.marked = true

Queue.enqueue(root)

While(!queue.isEmpty() ) {

Node r = queue.dequeue();

Visit(r);

Foreach (node n in r.adjacent) {

If(n.marked == false) {

n.marked = true

queue .enqueue(n)

}

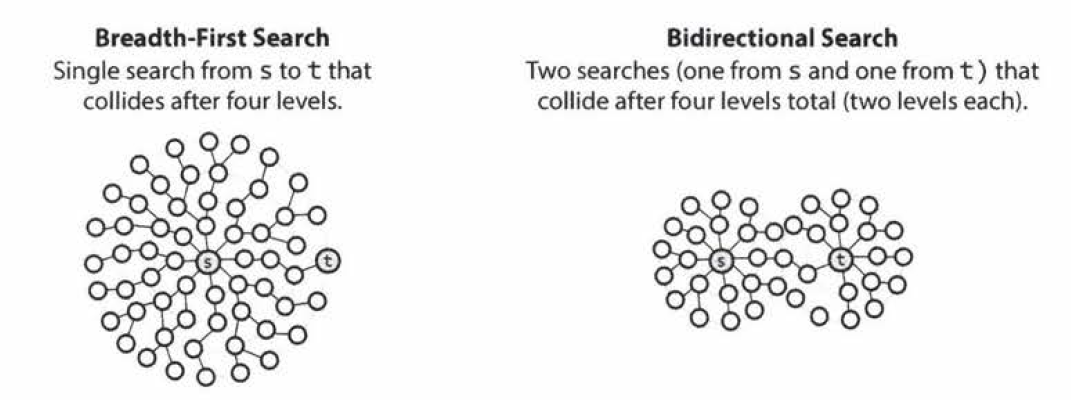
}

}

}

* Bidirectional search

BS is used to find the shortest path between a source and destination node. It operates by essentially running two simultaneous bfs, one from each node. When their searches collide, we have found a path.



+ BFS, consider where every node has at most k adjacent nodes and the shortest path from node s to node t has length d. We would search up to k nodes in the first “level” of the search. In the second level, would search up to k nodes for each of those first k nodes, so k2 nodes total(thus far). We would do this d times, so that’s O(kd) nodes

+ In bidirectional search, we have to searches that collide after approximately d/2 levels (the midpoint of the path). The search from s visits approximately Kd/2, as does the search from t. That’s approximately 2Kd/2 or O(kd/2), nodes total.

It’s huge different.

1. **Graph and tree advance**
2. Querying a binary search tree:

* Search:

Tree-search (x, k) {

If( x == NIL or k = x.key) {

Return x

}

If ( k < x.key) {

Return (Tree-search(x.left, k))

} else {

Return (Tree-search(x.right, k))

}

}

* Minimum:

Tree-Minimum (x) {

While x.left != NIL {

x = x.left

}

Return x

}

* Maximum:

Tree-Maximum (x) {

While x.right != NIL {

x = x. right

}

Return x

}

* Successor

Successor of a node x is the node with the smallest key greater than x.key

Tree-Successor(x) {

If(x.right != NiL) {

Return Tree-Minimum(x.right)

}

y = x.p

while( y != NIL && x == y.right) {

x = y

y = y.p

}

Return y

}

* Predecessor

Predecessor of a node x is the node with the largest key smaller than x.key

Tree-Predecessor (x) {

If(x.left != NiL) {

Return Tree-Maximum(x. left)

}

y = x.p

while( y != NIL && x == y.left) {

x = y

y = y.p

}

Return y

}

1. Dijkstra’s Algorithm:

In some graph, we might want to have edges with weights. If the graph represented cities, each edge might represent a road and its weight might represent the travel time. In this case, we might want to ask, just as your GPS mapping system does, what’s the shortest path from your current location to another point?

Dijkstra’s Algorithm is a way to find the shortest path between two points in a weighted directed graph. All edges must have positive values.

1. Start off at s
2. For each of s’s outbound edges, clone ourselves and start walking. If the edge (s, x) has weight 5, we should actually take 5 minutes to get there
3. Each time we get to a node, check if anyone’s been there before. If so, then just stop. We’re automatically not as fast as another path since someone beat us here from s. If no one has been here before, then clone ourselves and head out in all possible directions
4. The first one to get to t wins.

* One person moves at a time, and it’s always the one with the lowest time\_so\_far. This is sort of how Dijkstra’s algorithm works

Find the shortest path from a to i.

* Initialize serval variables :

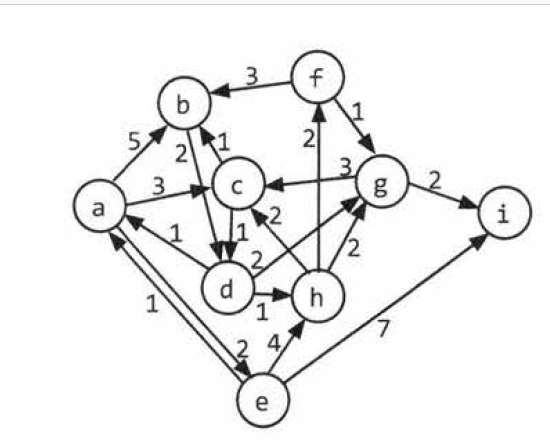
+ path\_weight [node]: maps from each node to the total weight of the shortest path. All values are initialized to infinity, except for path\_weight[a] which is 0

+ previous[node]: maps from each node to the previous node in the (current) shortest path

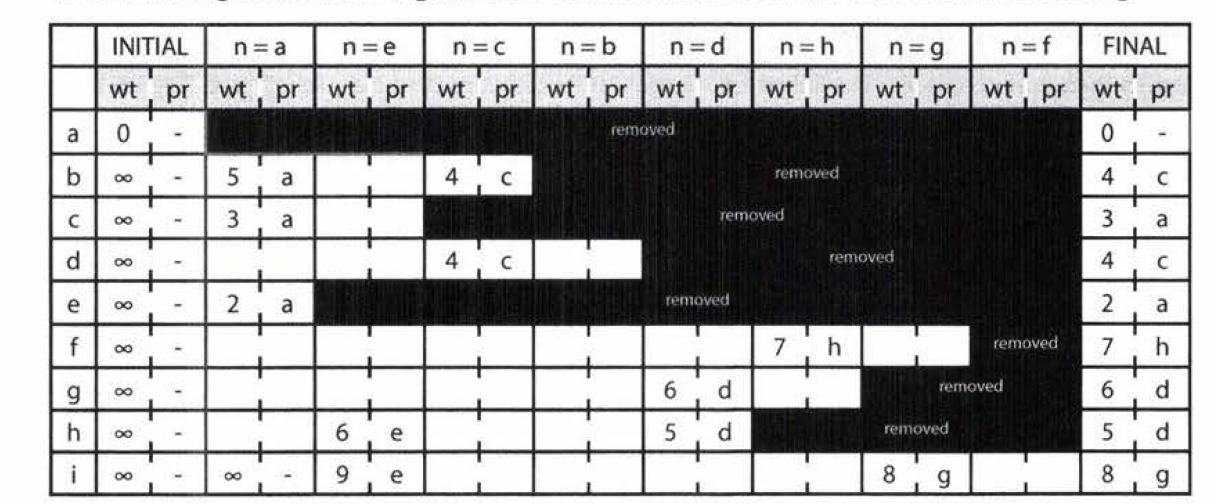
+ Remaining: a priority queue of all nodes in the graph, where each node’s priority is defined by its path\_weight

* We iterate through the nodes in remaining( until remaining is empty), doing the following:

1. Select the node in remaining with the lowest value in path\_weigth. Call this node n
2. For each adjacent node, compare path\_weight[x] (which is the weight of the current shortest path from a to x) to path\_weight[n] + edge\_weight[(n, x)]. That’s could we get a path from a to x with lower weight by going through n instead of our current path? If so, update path\_weight and previous
3. Remove n from remaining.



1. The first value of n is a. We look at its adjacent nodes (b, c and e), update the values of path\_weight (to 5,3 and 2) and previous (to a) then remove a from remaining.
2. Then, we go to the next smallest node, which is e. We previously updated path\_weight[e] to be 2. It’s adjacent nodes are h and I, so we update path\_weight( to 6 and 9) and previous for both of those.
3. The next smallest node is c, (3). Its adjacent nodes are b and d. We update b and d to 4 and 4



1. AVL trees

AVL tree is one of two common ways to implement tree balancing.

An AVL tree stores in each node the height of the subtrees rooted at this node. Then, for any node, we can check if it’s height balanced: that the height of the left subtree and the height of the right subtree differ by no more than one.