CS-260 : Notes

Charlie Stuart : src322

Winter 2019

Note: I made up the seciton order because I'm quirky

Based on my notes from CS-260 in the winter 2019 with Kurt, but then done in C.

Contents

List of Algorithms

1 Other Resources

Algorithm Visualization : https://www.cs.usfca.edu/%7Egalles/visualization/Algorithms.html

B-Tree Visualization : https://www.cs.usfca.edu/~galles/visualization/BTree.html

Gnarley Trees : https://people.ksp.sk/~kuko/gnarley-trees/

Red/Black Tree Visualization : https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

Sorting Video : https://www.youtube.com/watch?v=kPRA0W1kECg

Sorting Visualization : https://www.toptal.com/developers/sorting-algorithms

Sorting Visualization Again : https://www.hackerearth.com/practice/algorithms/sorting/bubble-sort/tutorial/

2 Math Review

2.1 Set Theory

From pages 3-7 in *Introduction to the Theory of Computation* by Michael Sipser

Set : A group of objects represented as a unit
Element : An object in a set
Member : An object in a set
Multi Set : An set containing an element that occurs multiple times
Subset : A set that consists of elements that exist in a different set
Proper Subset : A set that is a subset of another set, but not equal
Infinite Set : A set of infinitely many elements
Empty Set : A set of no elements
Singleton Set : A set of one elements
Unordered Pair : A set of two elements
Sequence : A set in a specific order
Tuple : A finite set
k-Tuple : A tuple of k elements
Ordered Pair : A 2-tuple
Power Set : All the subsets of A
$\in$: Is a member of
$\notin$: Is not a member of
$\subset$: Is a proper subset of
$\not\subset$: Is not a proper subset of
$\subseteq$: Is a subset of
$\not\subseteq$: Is not a subset of
$\cup$: Union of two sets
$\cap$: Intersection of two sets
$\times$: Cross product of two sets
$\mathbb{N}$: Set of natural numbers
$\mathbb{Z}$: Set of integers
$\mathbb{Q}$: Set of rational numbers
$\mathbb{A}$: Set of algebraic numbers
$\mathbb{R}$: Set of real numbers

A set is defined in a few ways

$S = \{7, 21, 57\}$	Finite Set
$S = \{1, 2, 3...\}$	Infinite Set of all natural numbers $\mathbb{N}$
$S = \{7, 7, 21, 57\}$	Multi Set
$S = \varnothing$	Empty Set
$S = \{5\}$	Singleton Set
$S = \{5, 3\}$	Unordered pair
$S = \{n \mid n = m^2 \text{ for some } m \in \mathbb{N}\}$	Set of perfect squares

The union of two sets is the same as an OR operator in boolean algebra. It's all the elements in both sets.

$$
\begin{aligned}
A &= \{1, 2, 3\} \\
B &= \{3, 4, 5\} \\
A \cup B &= \{1, 2, 3, 4, 5\}
\end{aligned}
$$

The intersection of two sets is the same as an AND operator in boolean algebra. It's all the elements that appear only in both sets.

$$
\begin{aligned}
A &= \{1, 2, 3\} \\
B &= \{3, 4, 5\} \\
A \cap B &= \{3\}
\end{aligned}
$$

The Cartesian product, or cross product, of two sets is the set of all ordered pairs where the first element is a member of the first set and the second element is a member of the second set for every combination.

$$
\begin{aligned}
A &= \{1, 2\} \\
B &= \{x, y, z\} \\
A \times B &= \{(1, x), (2, x), (1, y), (2, y), (1, z), (2, z)\}
\end{aligned}
$$

2.2 Functions

From pages 7-8 in *Introduction to the Theory of Computation* by Michael Sipser

Function : An objects that sets up an input-output relationship
Domain : The set of possible inputs to a function
Range : The set of possible outputs to a function

$f : D \to R$ Function f has domain D and range R

2.3 Summations

From CLRS Appendix A

REMEMBER : Summations are inclusive

Constants can be "taken out":

$$\sum_{i=1}^{n} cx_i = c\sum_{i=1}^{n} x_i$$

Addition can be broken up:

$$\sum_{i=1}^{n} (x_i + y_i) = \sum_{i=1}^{n} x_i + \sum_{i=1}^{n} y_i$$

Arithmetic Series :

$$\sum_{i=1}^{n} i = 1 + 2 + ... + n$$
$$\sum_{i=1}^{n} i = \frac{1}{2}n(n+1)$$
$$\sum_{i=1}^{n} i \in \Theta(n^2)$$

Sum of Squares :

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Sum of Cubes :

$$\sum_{i=0}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$

Geometric Series : When $x \neq 1$ and is real

$$\sum_{i=0}^{n} x^i = 1 + x + x^2 + ... + x^n$$
$$\sum_{i=0}^{n} x^i = \frac{x^{n+1} - 1}{x - 1}$$

Geometric Series : When the summation is infinite and $|x| < 1$

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1 - x}$$

Harmonic Series :

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}$$
$$H(n) = \sum_{i=1}^{n} \frac{1}{i}$$
$$H(n) = \ln n + O(1)$$

Logarithms :

$$S(n) = \sum_{i=1}^{n} \log(i)$$
$$S(n) = \log(1) + \log(2) + ... + \log(n-1) + \log(n)$$
$$S(n) = \log(1 * 2 * ... * (n-1) * n)$$
$$S(n) = \log(n!)$$

2.4 Limits

Indeterminate Forms : $\frac{\pm\infty}{\pm\infty}$, $\frac{0}{0}$

2.5 Logarithms

$$\log_b(XY) = \log_b(X) + \log_b(Y)$$
$$\log_b(\frac{X}{Y}) = \log_b(X) - \log_b(Y)$$
$$\log_b(X^y) = y \log_b(X)$$

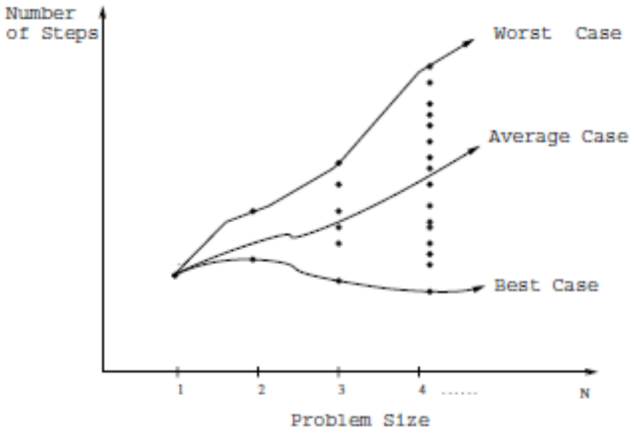
3 Big-Oh / Runtime

In order to make sure programs are efficient, they're timed. We analyze their rate of growth and compare them with other functions to describe them.

Worst Case : The slowest an algorithm can run on an input of problem size n. Eg: Insertion sort on a list of size n runs slowest when the list is sorted in the reverse order

Best Case : The fastest an algorithm can run on an input of problem size n. Eg: Insertion sort on a list of size n runs fastest when the list is sorted to start

Average Case : The average run time of an algorithm of all problem instances of size n



3.1 Asymptotic Notation

With asymptoic notations $o, O, \Theta, \Omega, \omega$ describe a set of functions. $O(f(n))$ describes all the functions bound above by $f(n)$.

3.1.1 Table of Informal Definitions

The "Kinda like Saying" isn't entirely correct, it's just to wrap my head around the bounds and relations

Name	Symbol	Informal Definition	Kinda like Saying
Little Omega	ω	Lower bound	$g(n) \in \omega(f(n))$ so $g(n) > f(n)$
Big Omega	Ω	Tight Lower bound	$g(n) \in \Omega(f(n))$ so $g(n) \geq f(n)$
Big Theta	Θ	Both an upper and lower bound	$g(n) \in \Theta(f(n))$ so $g(n) = f(n)$
Big Oh	O	Tight Upper bound	$g(n) \in O(f(n))$ so $g(n) \leq f(n)$
Little Oh	o	Upper bound	$g(n) \in o(f(n))$ so $g(n) < f(n)$

3.1.2 Table of Formal Definitions

Name	Symbol	Formal Definition
Little Omega	ω	$g(n) \in \omega(f(n)) \iff \exists c > 0, n_0 > 0 \ni g(n) > cf(n) \forall n > n_0$
Big Omega	Ω	$g(n) \in \Omega(f(n)) \iff \exists c > 0, n_0 > 0 \ni g(n) \geq cf(n) \forall n > n_0$
Big Theta	Θ	$g(n) \in \Theta(f(n)) \iff \exists c_1 > 0, c_2 > 0, n_0 > 0 \ni c_1 \leq g(n) \leq c_2 f(n) \forall n > n_0$
Big Oh	O	$g(n) \in O(f(n)) \iff \exists c > 0, n_0 > 0 \ni g(n) \leq cf(n) \forall n > n_0$
Little Oh	o	$g(n) \in o(f(n)) \iff \exists c > 0, n_0 > 0 \ni g(n) < cf(n) \forall n > n_0$

3.1.3 Table of Limit Definitions

Name	Symbol	Proving with Limits
Little Omega	ω	$\lim_{n\to\infty} f(n)/g(n) = \infty$
Big Omega	Ω	$\lim_{n\to\infty} f(n)/g(n) \neq 0$
Big Theta	Θ	$\lim_{n\to\infty} f(n)/g(n) \neq 0, \infty$
Big Oh	O	$\lim_{n\to\infty} f(n)/g(n) \neq \infty$
Little Oh	o	$\lim_{n\to\infty} f(n)/g(n) = 0$

3.1.4 Properties

Transitivity :

- $f(n) \in O(g(n))$ and $g(n) \in O(h(n)) : f(n) \in O(h(n))$
- $f(n) \in \Omega(g(n))$ and $g(n) \in \Omega(h(n)) : f(n) \in \Omega(h(n))$

Reflexivity : $f(n) \in \Theta(f(n))$

Transpose Symmetry : $f(n) \in O(g(n) \iff g(n) \in \Omega(f(n))$

Symmetry : $f(n) \in \Theta(g(n)) \iff g(n) \in \Theta(f(n))$

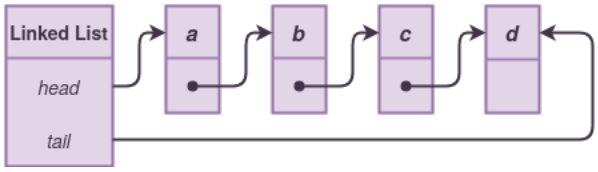
Trichotomy : For any two real numbers a and b: $a > b$ or $a = b$ or $a < b$

4 Linked Lists

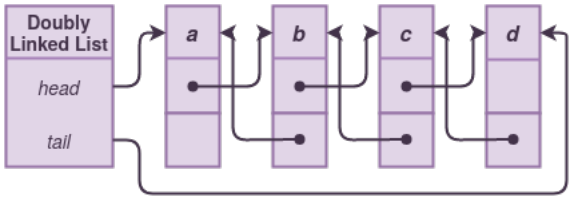
In Racket in CS-270, a list only had two elements. To create longer lists, you needed to nest lists.

```
(1, (2, (3, (4, 5))))
```

The structure of a linked list is similar. Each link has two parts, the data stored and a pointer to the next link.



Doubly Linked List : A linked list with a pointer to the previous link in addition to the pointer to the next

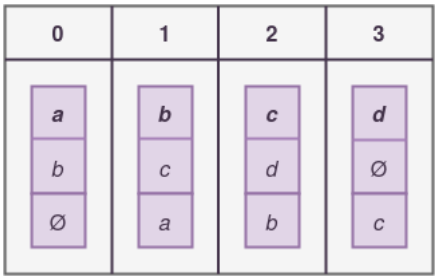


4.1 Functions

Function	Runtime
$\textsc{End}(L) \to p$	Linear
$\textsc{First}(L) \to p$	Constant
$\textsc{Next}(p, L) \to p$	Constant
$\textsc{Prev}(p, L) \to p$	Constant*
$\textsc{Retrieve}(p, L) \to x$	Constant
$\textsc{Append}(x, L)$	Linear
$\textsc{Insert}(x, p, L)$	Constant
$\textsc{Remove}(p, L) \to x$	Constant
$\textsc{Find}(x, L) \to p$	Linear
$\textsc{Size}(L) \to x$	Constant*

4.2 Implementations

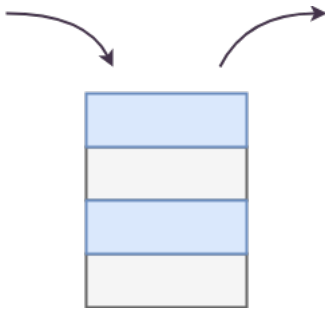
A Vector of Structs:



Parallel arrays:

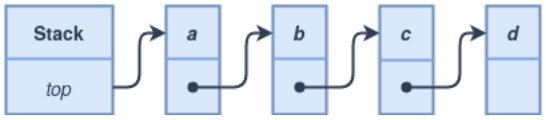
5 Stacks

Last On, First Off abstract data structure. It's like stacking plates, whatever was put on most recently has to come off before getting things under it.

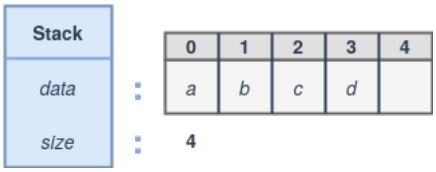


Can be implemented over a linked list or an array:

Linked List : All operations occur on the head, so operations only need to adjust the *head* pointer



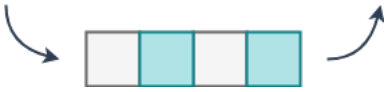
Array : All operations occur at the end of the array, as long as the *size* variable is kept up to date, operations are constant



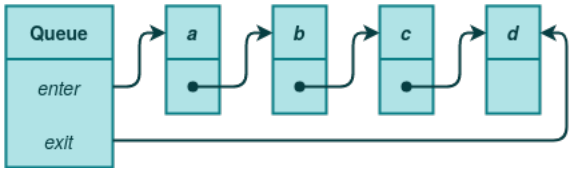
Function	Linked List Runtime	Array Runtime
INIT()	Constant	Constant
EMPTY?(S)	Constant	Constant
PUSH(x, S)	Constant	Constant
POP(S) $\rightarrow x$	Constant	Constant
PEEK(S) $\rightarrow x$	Constant	Constant

6 Queues

First In, First Out abstract data structure. It's like a fast food line. Whatever was added most recently has to wait until the things before it are removed.



Implementing over a linked list allows for constant time operations. Adding elements happens at the head by adjusting the head pointer. Removing elements happens at the tail by adjusting the tail pointer.



Functions	Runtime
INIT()	Constant
EMPTY?(Q)	Constant
ENQUEUE(x, Q)	Constant
DEQUEUE(Q) $\rightarrow x$	Constant
PEEK(Q) $\rightarrow x$	Constant

6.1 Priority Queues

Not exactly a queue. It behaves similarly, but isn't implemented as one or has the same quick runtimes for it's functions. Elements are inserted one at a time, then removed in order.

```
Insert 4
Insert 2
Insert 5
Insert 3
Remove 2 Min Queue
       5 Max Queue
Insert 1
Remove 1 Min Queue
       4 Max Queue
```

Can be implemented a few ways

Ordered Array : Insert via comparison. Remove the first element then shift the rest. Insertion and removal are both linear.

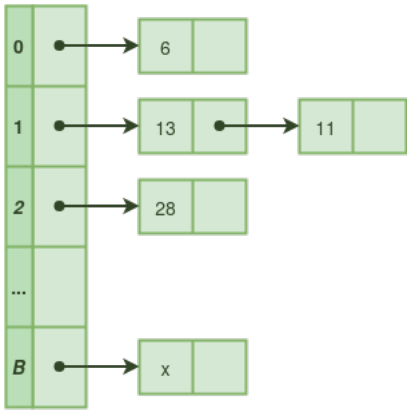
Unordered Array : Insert at the end of the array. Remove via comparison. Insertion is constant time, but removal is linear.

7 Hash Tables

An abstract data structure composed of buckets to hold data. There's a hash function that decides which data goes in which bucket. Hash functions tend to be one way. Given a piece of data, it can be hashed to get a new value. The new value cannot be unhashed to get the data. A good hash table with a good hash function has an even distribution of data across buckets, and there is some connection from the bucket to the data in it.

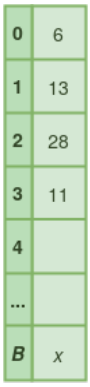
Open Hash Table

We have an array of buckets. Each array has a pointer to a linked list where elements are stored. The hash function sorts elements into a bucket where the element is appended to the linked list. Buckets don't have a size cap, but more elements in a bucket leads to more inefficient hashing.



Closed Hash Table

Implemented over an array. Each bucket can only hold one element. If we're trying to insert an element but its bucket is full, the element is inserted ito the next available bucket. If all buckets are filled, the hash table needs to be resized.



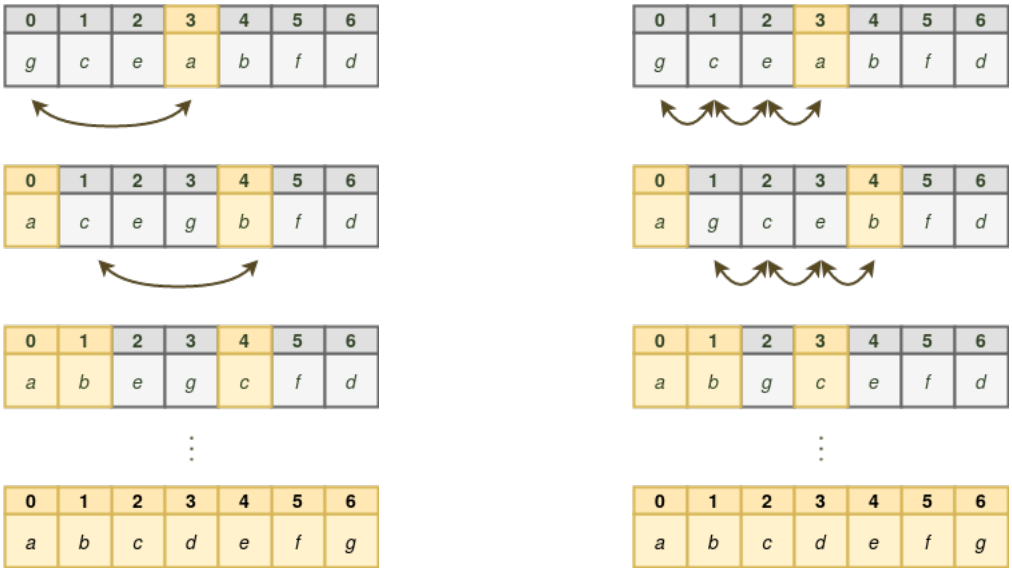
Resizing a Hash Table

1. Create a new hash table twice the size of the old one
2. Update the hashing function based on the new size
3. Rehash all data into the new table using the new function

8 Sorting

8.1 Selection Sort

The general idea is very straight forward. Basically, sort the array by finding the minimum element and making it the next element. It's slow, but fast to implement. It's $\in O(n^2)$ since finding the minimum of an array is linear, then you need to do that for every element in the array.



Algorithm 1 Selection Sort

```
1:  function SelectionSortA(A) // ∈ O(n^2)
2:      for i := 1 → A.length do
3:          key := i
4:          for j := i + 1 → A.length do
5:              if A[j] < A[key] then
6:                  key := j
7:              end if
8:          end for
9:          t := A[key]
10:         A[key] := A[i]
11:         A[i] := t
12:     end for
13: end function
```

Pretending I hadn't taken CS-457 and don't know how to properly solve recurrence relations, we can make an educated guess about function runtimes by unwinding recurrence relations. This isn't a formal proof, just an educated guess, but we'll play along with what Kurt gave since there's a lot more to proving a recurrence relation.

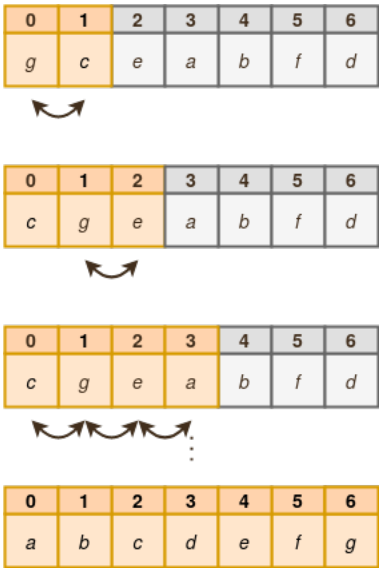
$$
\begin{aligned}
S(n) &= \begin{cases} d & n \in \{0,1\} \\ S(n-1) + cn & n > 1 \end{cases} \\
S(n) &= S(n-1) + cn && i = 1 \\
S(n) &= (S(n-2) + c(n-1)) + cn && i = 2 \\
&= S(n-2) + 2cn - c \\
S(n) &= (S(n-3) + c(n-2)) + 2cn - c && i = 3 \\
S(n) &= S(n-3) + 3cn - (1+2)c \\
S(n) &= S(n-k) + kcn - c\sum_{i=1}^{k-1} i && i = k \\
S(n) &= S(0) + c^2 - c\frac{n(n-1)}{2} && n = k \\
S(n) &= d + \frac{c^2}{2} - \frac{cn}{2} \in O(n^2)
\end{aligned}
$$

Basically, unwind the recurrence k times until the base case is reached. Based on the patterns seen, make an educated guess about the runtime.

8.2 Insertion Sort

Inefficient, but another quick implementation. Traverse the array. As I reach a new element, put it in the correct position in the sorted portion of the array.

This is less useful when trying to sort an existing array and more useful when you're inserting an element into an already existing sorted array. For example, a priority queue.



Algorithm 2 Insertion Sort

```
1:  function InsertionSort(A) // ∈ O(n^2)
2:      for j := 2 → A.length do
3:          key := A[j]
4:          // Insert A[j] into the sorted sequence A[1...j − 1]
5:          i := j − 1
6:          while  i > 0 and A[i] > key  do
7:              A[i + 1] := A[i]
8:              i := i − 1
9:          end while
10:         A[i + 1] := key
11:     end for
12: end function
```

8.3 Bubble Sort

Very inefficient, but to the point. It sorts elements by running through the array many times and swapping elements that are out of order.

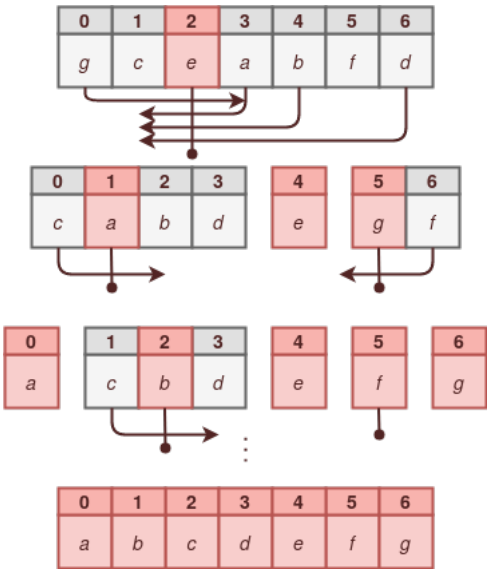
Algorithm 3 Bubble Sort

```
1:  function BubbleSort(A) // ∈ O(n^2)
2:      for i := 1 → A.length − 1 do
3:          for j := A.length → i do
4:              if A[j] < A[j − 1] then
5:                  t := A[j]
6:                  A[j] := A[j − 1]
7:                  A[j − 1] := t
8:              end if
9:          end for
10:     end for
11: end function
```

8.4 Quick Sort

A quick sort is fairly simple. Pick a random "partition". This partition is used as a post. We compare the rest of the elements in the array to the partition and group them based on whether they are less than or greater than the partition. We then recursively perform a quick sort on each of the new resulting arrays.

Quick sort is a fickle one. It's runtime is dependent on the partition and how "good" it is. A good partition cuts the array in half with an equal number of elements on each side. This makes our problem reduced by half. A bad partition is one where all the elements are less than or greater than the partition. This gives us two new problem sets. One where the problem is 0 elements. The other where the problem has $n - 1$ elements. The partition is chosen at random each iteration of the quick sort.



Algorithm 4 Quick Sort

```
function QuickSort(A, p, r) // Worst case ∈ O(n^2), Best case ∈ O(n log(n))
    if p < r then
        q := Partition(A, p, r)
        QuickSort(A, p, q − 1)
        QuickSort(A, q + 1, r)
    end if
end function
function Partition(A, p, r)
    x := A[r]
    i := p − 1 // i keeps track of where the "less than or equal to x" partition ends
    for j := p → r − 1 do
        if A[j] ≤ x then
            // Since we found another element less than the parition, increase the size of "less than" partition
and put the element in that parition
            i := i + 1
            t := A[i]
            A[i] := A[j]
            A[j] := t
        end if
    end for
    // Put the pivot between the two partitions
    t := A[i + 1]
    A[i + 1] := A[r]
    A[r] := t
    return i + 1
end function
```

Once again, pretending CS-457 wasn't a class I took and enjoyed and understood, we can unwind the recurrence relation and make an educated guess about its runtime. With Quick Sort however, we have a best case and worst case possibility we need to analyze.

Best Case:

$$Q_b(n) = \begin{cases} d & n \leq 1 \\ 2Q_b(\frac{n}{2}) + n & n > 1 \end{cases}$$

$$Q_b(n) = 2Q_b(\frac{n}{2}) + n \qquad i = 1$$

$$Q_b(n) = 2(2Q_b(\frac{n}{4}) + \frac{n}{2}) + n \qquad i = 2$$

$$Q_b(n) = 4Q_b(\frac{n}{4}) + 2n$$

$$Q_b(n) = 2^k Q_b(\frac{n}{2^k}) + kn \qquad i = k$$

$$Q_b(n) = nQ_b(1) + n\log(n) \qquad k = \log(n)$$

$$Q_b(n) = nQ_b(1) + n\log(n) \in O(n\log(n))$$

Worst Case:

$$Q_w(n) = \begin{cases} d & n \leq 1 \\ Q_w(n-1) + n & n > 1 \end{cases}$$

$$Q_w(n) = Q_w(n-1) + n \qquad i = 1$$

$$Q_w(n) = (Q_w(n-2) + n - 1) + n \qquad i = 2$$

$$Q_w(n) = Q_w(n-2) + 2n - 1$$

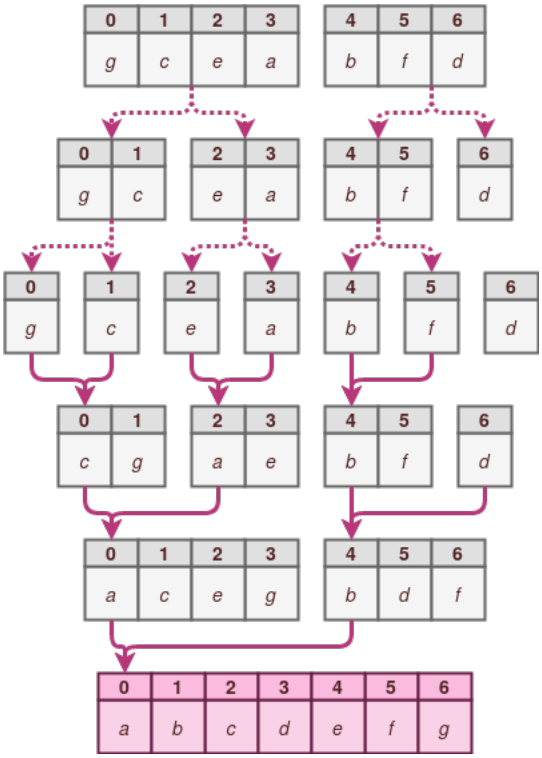
$$Q_w(n) = Q_w(n-k) + kn \qquad i = k$$

$$Q_w(n) = Q_w(0) + n^2 \qquad k = n$$

$$Q_w(n) = d + n^2 \in O(n^2)$$

8.5 Merge Sort

Merge sort is consistently efficient in terms of time, not very effiecient in terms of space. We consistently split the array in half until each array is sorted. We know for a fact that an array is sorted when there's only one element. So split the array until it's only one element long, then, merge them. Once we have these "sorted" one element arrays, we merge them back into full length arrays. When merging, since we know the arrays we're merging are sorted, we only need to compare the first elements of each array with each other, then add the smaller element to the sorted array. This leads to a consistent $O(n \log(n))$ runtime.



Algorithm 5 Merge Sort

```
1: function MergeSort(A, p, r) // ∈ O(n log(n))
2:     if p < r then
3:         q := ⌊(p+r)/2⌋
4:         MergeSort(A, p, q)
5:         MergeSort(A, q + 1, r)
6:         Merge(A, p, q, r)
7:     end if
8: end function
```

Algorithm 6 Merge Sort helper function

```
1: function MERGE(A, p, q, r)
2:     n_1 := q − p + 1
3:     n_2 := r − q
4:     L := [1...n_1 + 1] // A new array
5:     R := [1...n_2 + 1] // A new array
6:     for i := 1 → n_1 do
7:         L[i] := A[p + i − 1]
8:     end for
9:     for j := 1 → n_2 do
10:        R[j] := A[q + j]
11:    end for
12:    L[n_1 + 1] := ∞
13:    R[n_2 + 1] := ∞
14:    i := 1
15:    j := 1
16:    for k := p → r do
17:        if L[i] ≤ R[j] then
18:            A[k] := L[i]
19:            i := i + 1
20:        else
21:            A[k] := R[j]
22:            j := j + 1
23:        end if
24:    end for
25: end function
```

8.6 Radix Sort

Given the following set of Base 4 numbers of length 3, we can sort them digit by digit and sort them. This can be easily implemented with a hash table.

0	1	2	3	4	5	6	7	8	9	10	11
231	211	112	133	321	203	000	312	001	021	302	210

	0	1	2	3
XX**X**	00**0** 21**0**	23**1** 21**1** 32**1** 00**1** 02**1**	11**2** 31**2** 30**2**	11**3** 20**3**
X**X**X	0**0**0 0**0**1 3**0**2 2**0**3	2**1**0 2**1**1 1**1**2 3**1**2	3**2**1 0**2**1	2**3**1 1**3**3
XXX	**0**00 **0**01 **0**21	**1**12 **1**33	**2**03 **2**10 **2**11 **2**31	**3**02 **3**12 **3**21

0	1	2	3	4	5	6	7	8	9	10	11
000	001	021	112	133	203	210	211	231	302	312	321

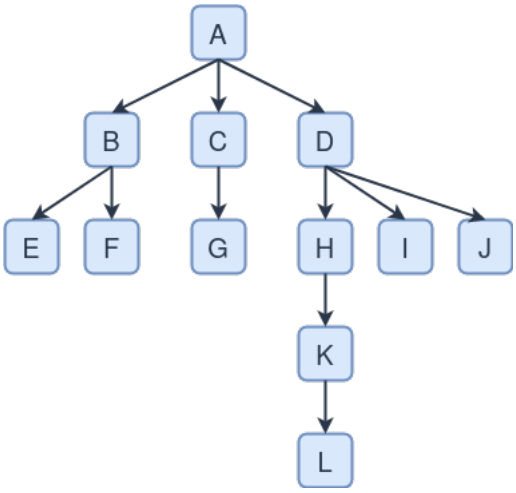
9 Trees

A tree is a rooted abstract data structure with hierarchy. A tree has a root, which contains data, then x children that "branch" off the root and also contain data. These children can be subtrees with children of their own or leaves, which are trees without children.

Depth of a Vertex : The number of edges from the root to the vertex

Height of a Vertex : The number of edges from a leaf to the vertex

Height of a Tree : The number of edges from the deepest leaf to the root



Node	A	B	C	D	E	F	G	H	I	J	K	L
Depth	0	1	1	1	2	2	2	2	2	2	3	4
Height	4	1	1	3	0	0	0	2	0	0	1	0

9.1 Tries

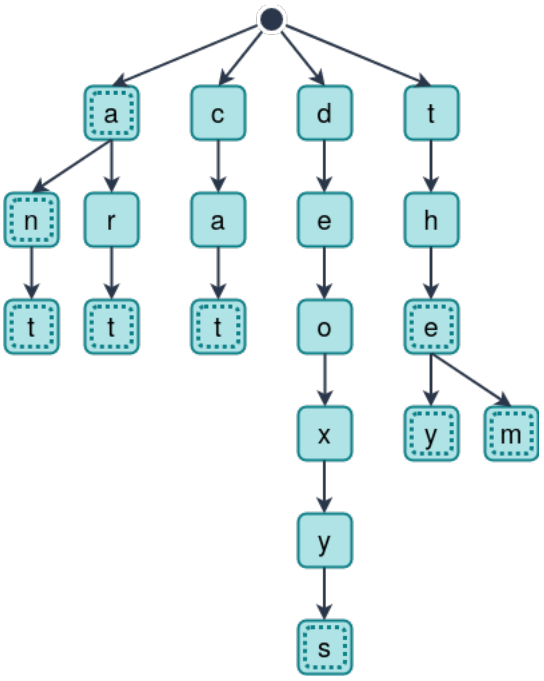
I think the example Kurt used here was autocorrect. A trie has an alphabet, A string/word in a trie is denoted by the extra dotted line in the diagram below. ~~I really used the name of a pokemon as a string huh~~ If we traverse in pre-order (basically a depth first search) and print a word everytime we reach a "dotted box" that denotes a word, we'll print all the words in the trie's dictionary in alphabetical order.

λ : The alphabet. The characters used in each string. In the case of the trie below, the alphabet is just the 26 letter English alphabet.

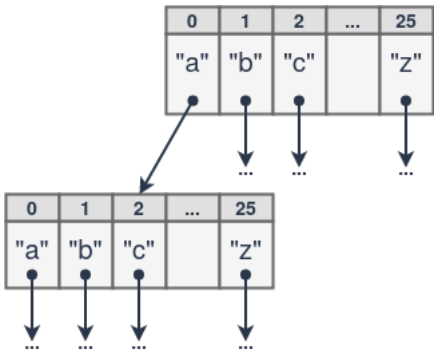
s **:** The length of the alphabet. $||\lambda||$

L **:** The average string length

n **:** The number of strings in a trie

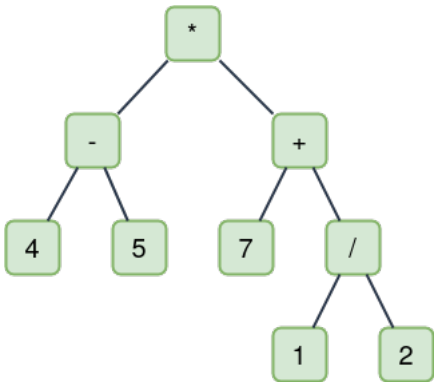


This can be stored as an array, but gets unwieldy. Each subtree needs s children.



9.2 Parse Trees

A parse tree, also called an expression tree, is mostly related to how compilers parse code and translate expressions into mathematical expressions. A parse tree is almost always a binary tree meaning each root has two children, a designated left and right child. Assume we have the following parse tree:



We can read this tree using three different notations.

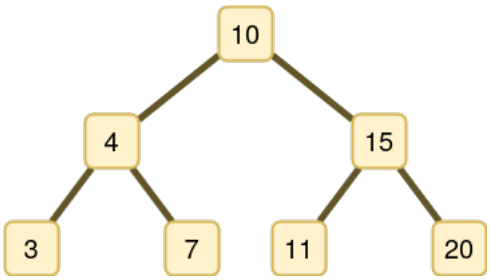
Prefix : Parse in the order root, left, right. `(* (- 4 5) (+ 7 (/ 1 2)))`

Infix : Parse in the order left, root, right. `((4 - 5) * (7 + (1 / 2)))`

Postfix : Parse in the order left, right, root. `((4 5 -) (7 (1 2 /) +) * )`

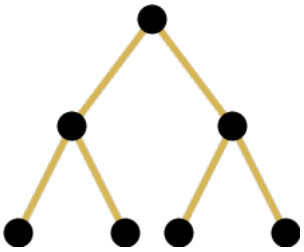
9.3 Binary Search Tree

A binary tree is a tree where there's only two children. One is a distinct left child, the other is a distinct right child. We use this property to our advantage in a binary search tree. We can use the left child to denote all elements less than the root and the right child to denote all elements greater than the root.



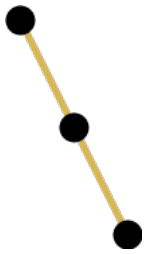
In a binary tree, the height has a best case and worst case growth depending on how filled a depth is.

Best Case:
Before incrementing the height, we fill each height as much as possible. This way, we don't increase the height unless we have no other choice. In this case, the height grows in $\log(n)$.



h	n
0	1
1	3
2	7
3	15
h	$2^{h+1} - 1$

Worst Case:
When we try to put only the required nodes to get us to the next height at each height, the height grows linearly.



h	n
0	1
1	2
2	3
3	4
h	$h + 1$

Now that we have this "left less than, right greater than" structure defined for storing data, we can using this to make searching easier. Given a binary search tree (or just a sorted array) we can do a binary search by checking the middle element and comparing it against what we're looking for until we either reach the element we're looking for or we fall off the tree.

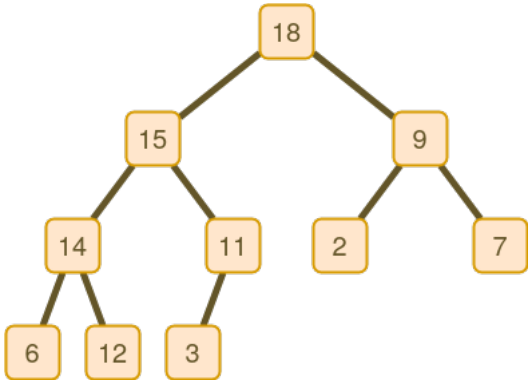
We can make an educated recurrence relation guess again:

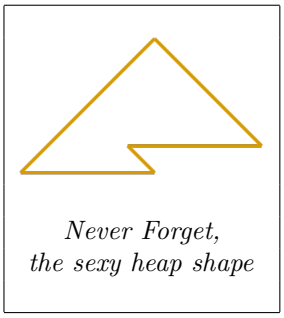
$$
\begin{aligned}
B(n) &= \begin{cases} d & n \in \{0,1\} \\ B(\frac{n}{2}) + c & n > 1 \end{cases} \\
B(n) &= B(\frac{n}{2}) + c && i = 1 \\
B(n) &= (B(\frac{n}{4}) + c) + c && i = 2 \\
&= B(\frac{n}{4}) + 2c \\
B(n) &= (B(\frac{n}{8}) + c) + 2c && i = 3 \\
&= B(\frac{n}{8}) + 3c && i = 3 \\
B(n) &= B(\frac{n}{2^k}) + kc && i = k \\
1 &= \frac{n}{2^k} \\
n &= 2^k \\
k &= \log(n) \\
B(n) &= B(1) + \log(n)c && k = \log(n) \\
B(n) &= d + \log(n)c \in O(\log(n))
\end{aligned}
$$

9.4 Heaps

A binary tree with two extra properties:

1. It's full. It fills left to right
2. Children are not X than parent
   - **Min Heap :** Minimum element is at the top of the heap
   - **Max Heap :** Maximum element is at the top of the heap





Can be implemented easily over an array *Note: Indexed at 1*:

1	2	3	4	5	6	7	8	9	10
18	15	9	14	11	2	7	6	12	3

Algorithm 7 Heap Implementation Functions

```
1:  function PARENT(i)
2:      return ⌊i/2⌋
3:  end function
4:
5:  function LEFT(i)
6:      return 2i
7:  end function
8:
9:  function RIGHT(i)
10:     return 2i + 1
11: end function
```

Algorithm 8 Heap Helper Functions

```
1: function UPHEAP(i, H) // ∈ O(log(n))
2:     // For a min heap, replace the > with a <
3:     while i > 1 and H[i] > H[Parent(i)] do
4:         swap(i, Parent(i), H)
5:         i := Parent(i)
6:     end while
7: end function
8:
9: function DOWNHEAP(i, H) // ∈ O(log(n))
10:    if Left(i) ¿ H.size then return
11:    end if
12:    li := Left(i) // The index of the larger child
13:    // For a min heap, replace the < with a >
14:    if Right(i) ≤ H.size and H[li] < H[Right(i)] then
15:        li := Right(i)
16:    end if
17:    // For a min heap, replace the > with a <
18:    if H[i] < H[li] then
19:        swap(i, li, H)
20:        Downheap(li, H)
21:    end if
22: end function
23: function INSERT(x, H) // ∈ O(log(n))
24:    ResizeArray?()
25:    H.size := H.size + 1
26:    H[H.size] := x
27:    Upheap(H.size, H)
28: end function
29:
30: function REMOVE(H) // ∈ O(log(n))
31:    if Empty?() then return
32:    end if
33:    ResizeArray?()
34:    rv := H[1]
35:    H[1] := H[H.size]
36:    H.size := H.size - 1
37:    Downheap(1, H) return rv
38: end function
39:
40: function BUILDMAXHEAP(A) // ∈ O(n log(n))
41:    A.size := A.length
42:    for i := ⌊A.length/1⌋ down to 1 do
43:        Downheap(i, A)
44:    end for
45: end function
```

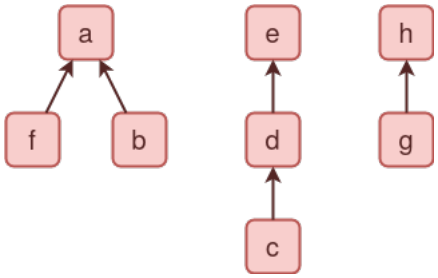
Since we know the heap has this property where the maximum/minimum element is always the root, by putting a bunch of elements in a heap, the continually grabbing the root, we basically sort those elements.

Algorithm 9 Heap Sort

```
1: function HeapSort(A) // ∈ O(n log (n))
2:     BuildMaxHeap(A)
3:     for i := A.length down to 2 do
4:         swap(1, i, A)
5:         A.size := A.size - 1
6:         Downheap(1, A)
7:     end for
8: end function
```

9.5 Merge-Find Sets

Not technically a tree, but it's with the tree stuff since it has a parent structure. It's a set of data that can be implemented over a map of parent pointers. In the map, the key is the node and the value is the parent of the key.



a	b	c	d	e	f	g	h
-1	a	d	e	-1	a	h	-1

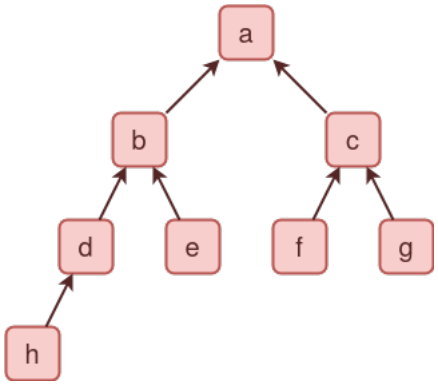
Algorithm 10 Merge-Find Set

```
1:  function FIND(x, S) // ∈ O(Height)
2:      // Finds x in set S and returns the parent
3:      p := x
4:      while S[p] ≠ −1 do
5:          p := S[p]
6:      end while
7:      return p
8:  end function
9:  function MERGE(x, y, S) // ∈ O(Height)
10:     // As long as we point the smaller tree to the larger one, the height doesn't grow
11:     c_1 := FIND(x, S)
12:     c_2 := FIND(y, S)
13:     S[c_2] := c_1
14: end function
```

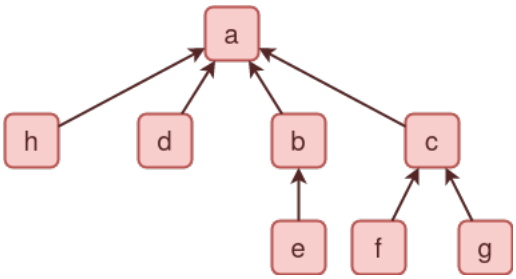
9.5.1 Path Compression

During a FIND we walk up a tree to find the "ultimate parent". As we climb, we point all the nodes we pass to this "ultimate parent". Eventually all nodes point to the root and look up is constant.

Before Compression



After FIND(h, S)



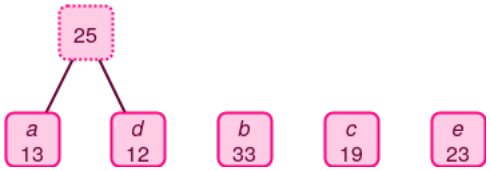
Algorithm 11 Merge-Find Set with Path Compression

```
1: function FIND(x, S) // ∈ O(Height)
2:     // Finds x in set S and returns the parent
3:     p := x
4:     while S[p] ≠ −1 do
5:         p := S[p]
6:     end while
7:     t := x
8:     u := t
9:     while S[t] ≠ −1 do
10:        u := S[t]
11:        S[t] := p
12:        t := u
13:    end while
14:    return p
15: end function
```

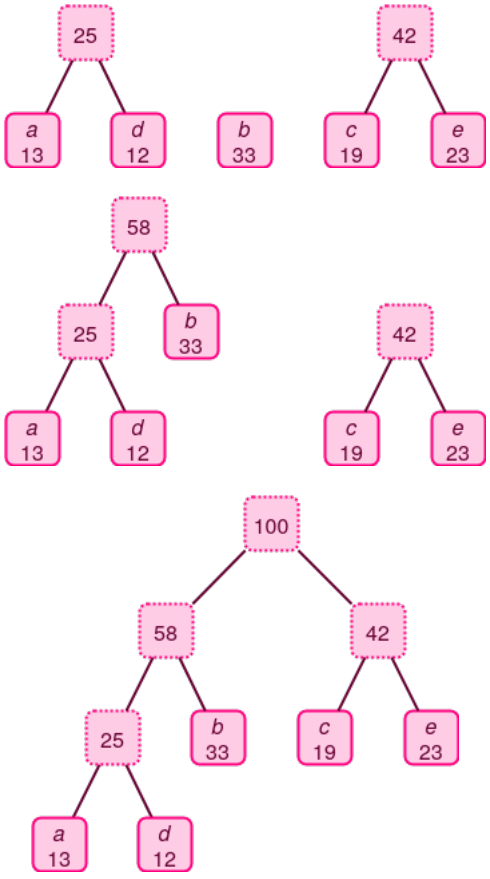
9.6 Huffman Codes

We can use Huffman trees to encode probabilities. Given the following pieces of data:

We then combine the two smallest pieces of data:



Using this new value, we repeat the process until we have a full tree:



In the final tree, each probability is a leaf. This means we will not pass probabilities as we fall down the tree searching for probabilities. As we fall down the tree, we give each probability a binary code. If we fall **left, 0**. If we fall **right, 1**. This gives the following table:

Node	Probability	Code	Code Length
a	0.13	000	3
b	0.33	01	2
c	0.19	10	2
d	0.12	001	3
e	0.23	11	2

Since all the letters are leaves, no probability is a prefix of another probability. We can concatenate all the codes together and get the following encoding:

Encoding : *adbce*

Encoding : 000001011011

Then we can calculate the average code length using the lengths and probabilities:

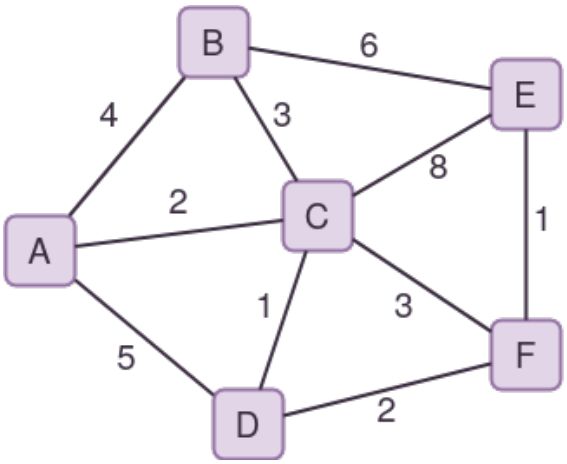
$$A = (0.13 * 3) + (0.33 * 2) + (0.19 * 2) + (0.12 * 3) + (0.23 * 2)$$
$$A = 2.25$$

10 Graphs

Graph : A collection of nodes, called vertices, containing data connected through edges. Edges can be directed which allows for one way travel and hierarchy, or undirected where travel is allowed in both directions.

Path : A string of vertices connected by edges

Cycle : A path with repeated vertices



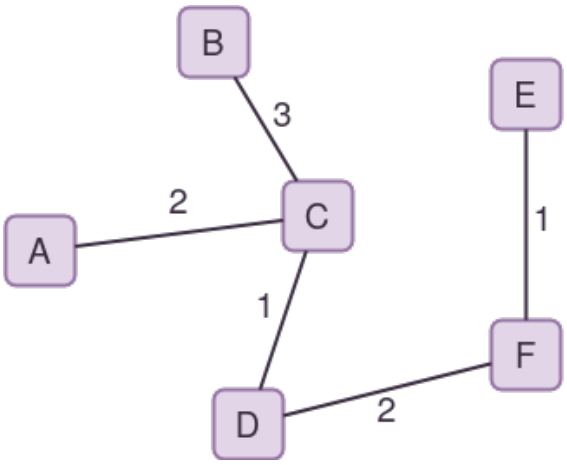
Predecessor Matrix
From u to v, the predecessor of v is u
These make more sense in a directed graph and when building paths and spanning trees.
Currently, from **A** to **B**, since the graph is undirected, **A** can be the predecessor of **B** and vice versa. The matrix below only contains the edges, but by applying various algoriths, we can fill this in with paths. Currently, from **D** to **E**, there is no predecessor. We can find a path between them through **C**. This would make **C** the predecessor of **E** from **D**.

π	A	B	C	D	E	F
A	∅	A	A	A	∅	∅
B	B	∅	B	∅	B	∅
C	C	C	∅	C	C	C
D	D	∅	D	∅	∅	D
E	∅	E	E	∅	∅	E
F	∅	∅	F	F	F	∅

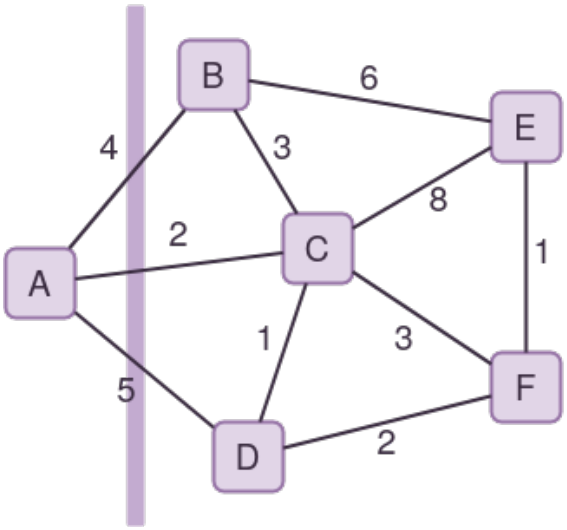
Distance Matrix
The full path distance from u to v.
The matrix below only contains the edges. This shows that an edge (which is just a very short path) between **A** and **B** exists and its of length 4. Once again, by applying algorithms we can find longer paths between vertices. From **D** to **E**, there currently is a infinite distance, meaning there is no path yet. We can find the path through **C** again and find the distance from **D** to **E** to be 9.

d	A	B	C	D	E	F
A	∞	4	2	5	∞	∞
B	4	∞	3	∞	6	∞
C	2	3	∞	1	8	3
D	5	∞	1	∞	∞	2
E	∞	6	8	∞	∞	1
F	∞	∞	3	2	1	∞

Minimum Spanning Tree : A tree that connects all the vertices in a graph with the lowest possible weighted edges

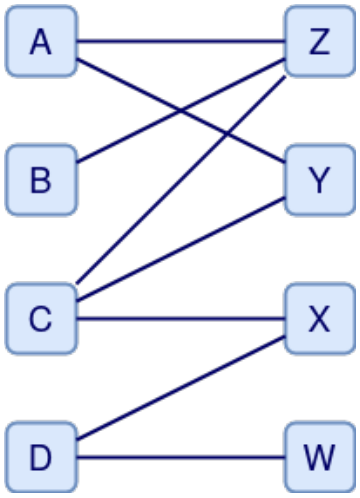


Cut : A parition of vertices. Shown as a vertical highlight below.



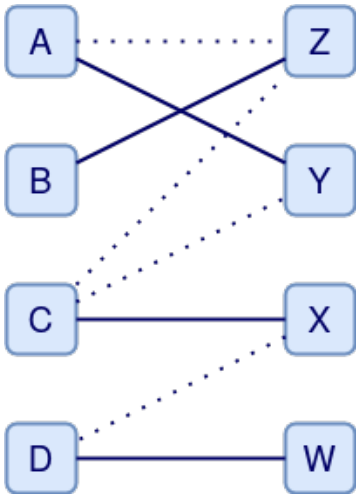
10.1 Bipartide Graphs

A specific type of graph with two "sides". Nodes on the same "side" have no edges connecting them.



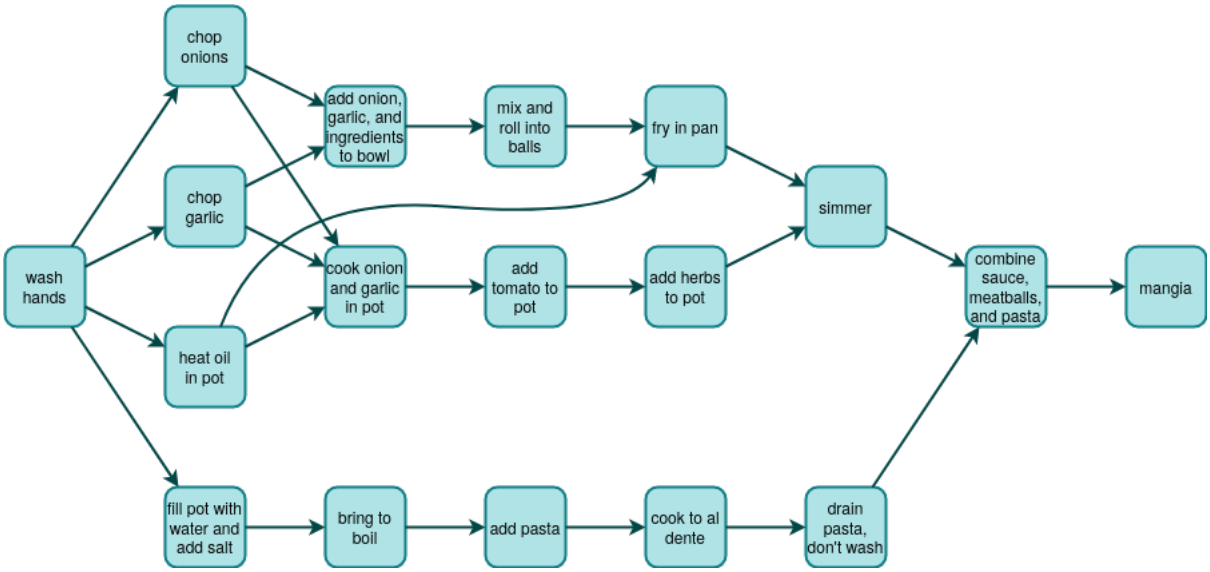
Matching : A node on one side has a node on the other. Shown below as dotted edges.

Perfect Matching : Every node has a unique partner on the other



10.2 Directed Acyclic Graphs (DAG)

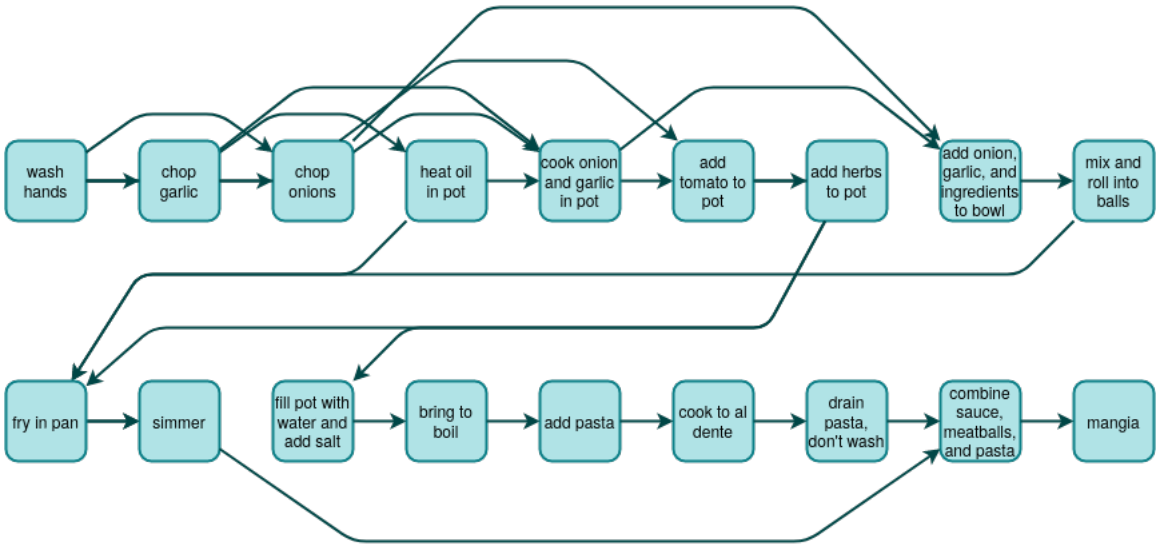
A DAG is a fairly self explanatory type of graph. It's directed and acyclic. This allows for it to act as a graph of dependencies. The DAG below is about making pasta, sauce, and meatballs. You can't cook the onions and garlic in a pan without first chopping the onions and garlic and also heating the oil. The boiling of the pasta isn't dependent on whether or not the sauce is cooked, so it gets its own branch. I could've added a vertex for reheating sauce that was previously cooked, but I chose not to make the graph more complicated.



Source : A vertex without any in edges

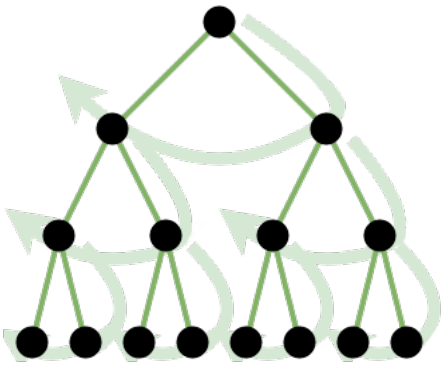
Sink : A vertex without any out edges

A DAG can be **topologically sorted** using a depth-first search. This sorts the vertices in order such that there are no backedges. I did my best to topologically sort my recipe DAG, but it got a bit unwieldy.

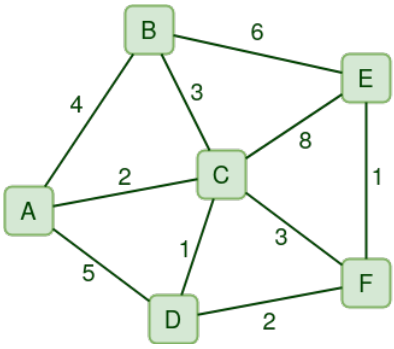


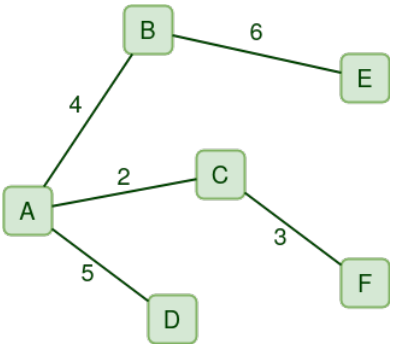
10.3 Breadth First Search

In a breadth first search (BFS), we first check each child, then we traverse each of the childrens' children. We don't move onto the next depth until we've already completed one.



We can use a BFS to make a spanning tree. It's not a minimum spanning tree, but it is a tree that spans. Below is a the result of a BFS starting at A.



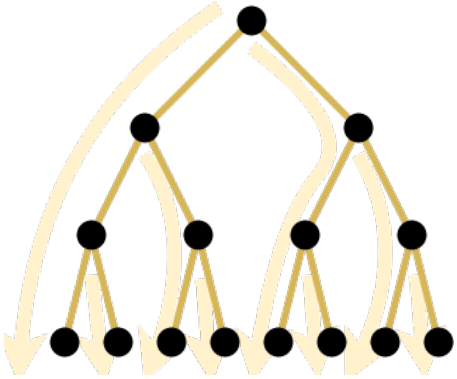


Algorithm 12 Breadth First Search

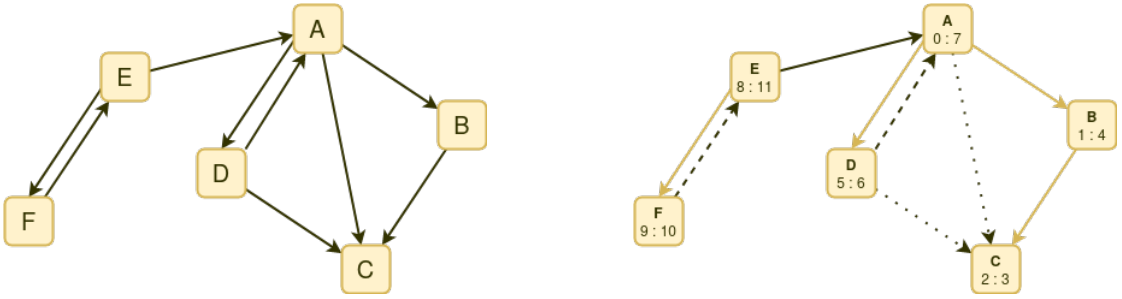
```
1:  function BFS(G(V, E), s) // ∈ O(V^2)
2:      n := ||V||
3:      vg := [] // A Map
4:      rv := {} // A Set
5:      for ∀v ∈ V do
6:          vg[v] := False
7:      end for
8:      vg[s] := True
9:      for ∀v ∈ adj(s, G) do
10:         if not vg[v] then
11:             vg[v] := True
12:             rv := rv ∪ (s, v)
13:         end if
14:     end for
15:     for ∀v ∈ adj(s, G) do
16:         BFS(G, v)
17:     end for
18:     return rv
19: end function
```

10.4 Depth First Search

A depth first search (DFS) is like the opposite of a breadth first search (BFS). Instead of trying to complete a depth and traversing all the children first, a DFS "falls" all the way to the furthest child/neighbor, then works its way back.



Like a BFS, a DFS makes a spanning tree. Once again, not minimum, just a tree that spans. Based on the discovery times, we get a variety of interesting properties about the graph and vertices. Part of this is what makes a DAG topologically sorted. Below shows the result of a DFS on a graph:



Discovery Time : When the vertex was "found" or first seen in the DFS. In the image above, the first number on the vertex.

Finishing Time : When all the children of the vertex were fully traversed in the DFS. In the image above, the second number on the vertex.

Tree Edge : An edge (u, v) that's a part of the spanning tree created during a DFS. In the image above, denoted by a highlighted yellow edge. u has been discovered and is "gray", v hasn't been discovered yet and is "white".

Cross Edge : An edge (u, v) where v has been finished and is "black" while u is still in progress and is "gray". In the above diagram, denoted by a solid black line.

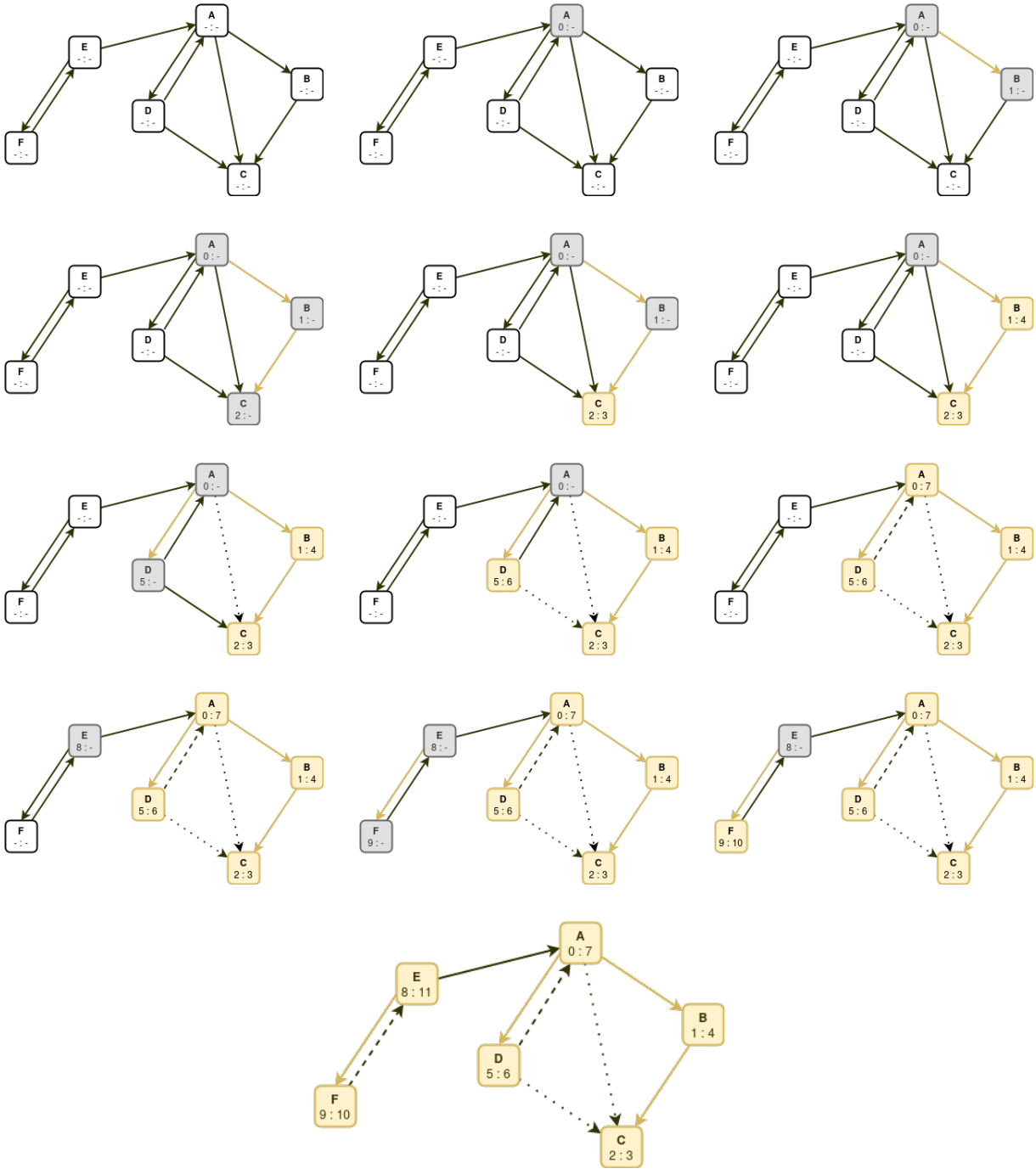
Forward Edge : An edge (u, v) where v is a child of u or one of it's children and v has already been discovered by another child of u. This is when v is "black" and u is still "gray". In the above diagram, denoted by a dotted black line.

Back Edge : An edge (u, v) where u is a child of v. Both u and v are "gray". In the above diagram, denoted by a dashed black line.

The colors of the node denote how discovered a vertex is.

- **White :** Not yet seen in the algorithm
- **Gray :** Has been seen in the algorithm, but it's children are still being traversed.
- **Black :** (Yellow in the step by step images below) The vertex and its children have been fully traversed.

In action, this looks like:

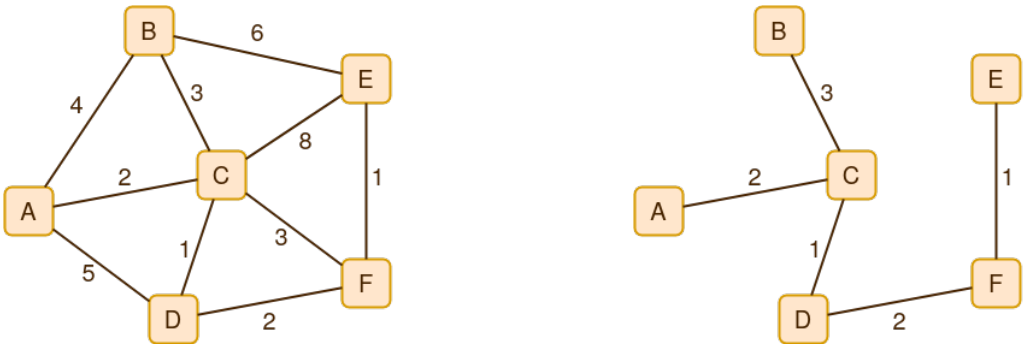


Algorithm 13 Depth First Search

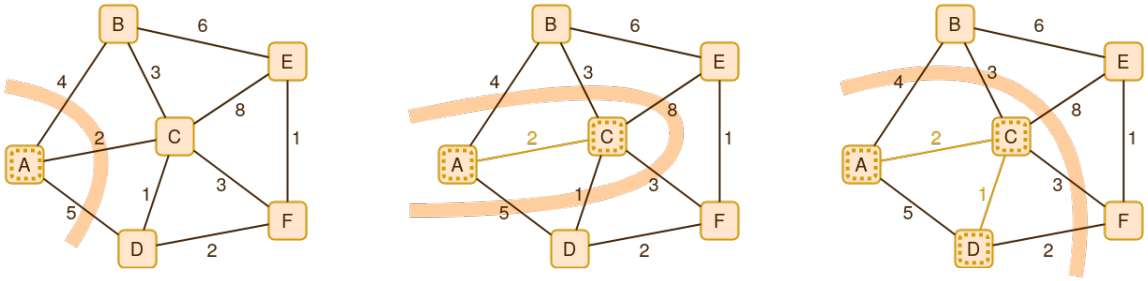
```
1: function DFS(G(V, E), s) // ∈ O(E)
2:     d[s] := time // Discovery time
3:     time := time + 1
4:     for ∀v ∈ adj(s, G) do
5:         if color(v) ≡ WHITE then
6:             color(v) := GRAY
7:             π[v] := s
8:             DFS(G, s)
9:         end if
10:    end for
11:    color(s) := BLACK
12:    f[s] := time // Finishing time
13:    time := time + 1
14: end function
15: function DFS-Exec(G(V, E)) // ∈ O(VE)
16:    time := 0
17:    for ∀v ∈ V do
18:        color(v) := WHITE
19:        π[v] := ∅
20:    end for
21:    for ∀v ∈ V do
22:        if color(v) ≡ WHITE then
23:            color(v) := GRAY
24:            DFS(G, v)
25:        end if
26:    end for
27: end function
```

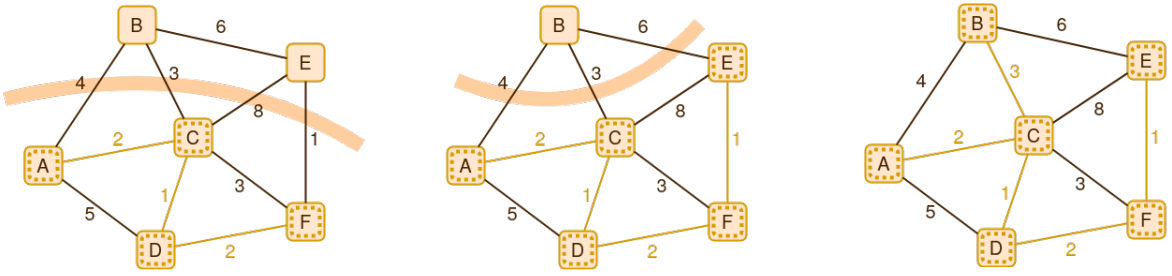
10.5 Prim's Spanning Tree

Prim's algorithm finds *a* minimum spanning tree in a graph. It is not *the most* minimum spanning tree, but it is **a** minimum spanning tree. It's a greedy algorithm, and only pays attention to what it sees in the moment, not the full picture.



We need a starting point, and make a cut in the graph isolating that vertex. Then, add the minimum edge that bridges that cut. Below, the process is shown step by step. The starting point is **A**. As edges are added the the minimum spanning tree, they get highlighted. Vertices in the tree are dotted.



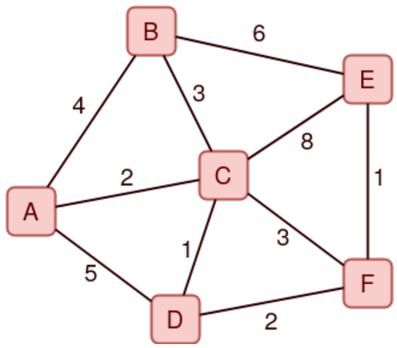


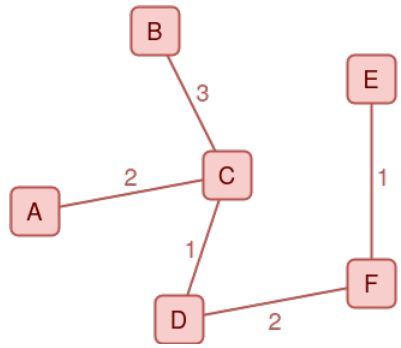
Algorithm 14 Prim's Algorithm

```
1: function PRIM(G(V, E), s) // Priority Queue ∈ O((V + E) log (V)) Unordered Array ∈ O(V^2)
2:     for ∀v ∈ V do
3:         key[v] := ∞ // Edge weights
4:         π[v] := ∅
5:     end for
6:     key[s] := 0
7:     PQ := MinQ(keys)
8:     while not EMPTY?(PQ) do
9:         x := POP(PQ) // PQ ∈ O(V log (V)) UA ∈ O(V^2)
10:        for ∀v ∈ adj(x, G) do
11:            if key[v] > wt(x, v) then
12:                key[v] := wt(x, v) // PQ ∈ O(E log (V)) UA ∈ O(E)
13:                π[v] := x
14:            end if
15:        end for
16:    end while
17: end function
```

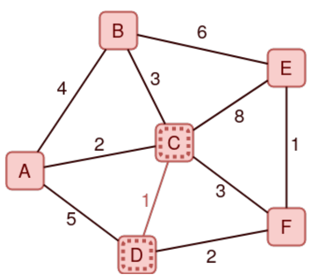
10.6 Kruskal's Spanning Tree

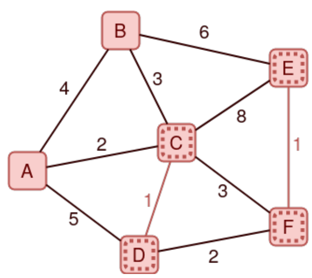
Kruskal's algorithm, like Prim's, finds *a* minimum spanning tree, not *the most* minimum spanning tree. Unlike Prim's, Kruskal's isn't greedy. It looks at the full picture instead of only the relevant nodes at a time.

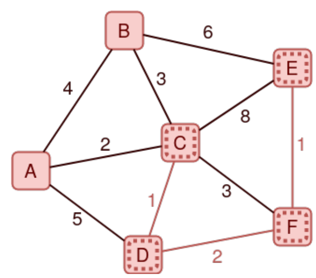


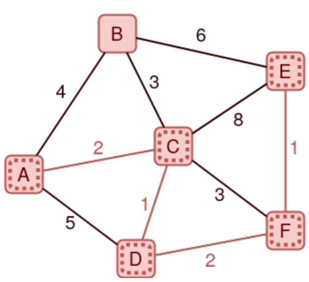


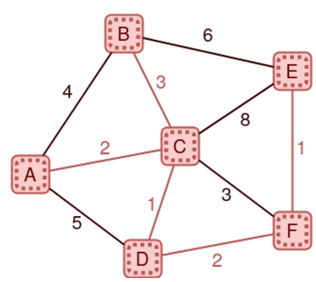
The algorithm doesn't need a starting point and instead takes all the edges and orders them by weight. As long as the most minimum edge isn't redundant, it gets added to the tree. Below, vertices in the tree have a dotted outline. As edges are added to the spanning tree, they get highlighted.











Algorithm 15 Kruskal's Algorithm

```
1: function KRUSKAL(G(V, E)) // ∈ O(E log (E))
2:     rv := {} // A Set
3:     S := MFSET(v)
4:     l := SORTED(E, wt, incr) // Sort the edged by weight increasing
5:     for ∀(u, v) ∈ l do
6:         if FIND(u, S) ≠ FIND(v, S) then
7:             rv := rv ∪ (u, v)
8:             MERGE(u, v, S)
9:         end if
10:    end for
11:    return rv
12: end function
```

10.7 Floyd-Warshaw's Shortest Path

This one wasn't covered so much. Kurt cared much more about Dijkstra. Floyd-Warshaw finds the shortest paths between vertices. It's straightforward and horribly inefficient. It also doesn't work with on negative cycles. To quote Kurt, "If you could walk in a circle and get younger, you'd do it forever."

Algorithm 16 Floyd-Warshaw Algorithm

```
1:  function FloydWarshaw(G(V, E)) // ∈ O(V^3)
2:      n := ||V||
3:      for i := 1 → n do
4:          for j := 1 → n do
5:              if (i, j) ∈ E then
6:                  d[i, j] := wt(i, j)
7:                  π[i, j] := i
8:              else
9:                  d[i, j] := ∞
10:                 π[i, j] := ∅
11:             end if
12:         end for
13:     end for
14:     for k := 1 → n do
15:         for i := 1 → n do
16:             for j := 1 → n do
17:                 if d[i, j] > d[i, k] + d[k, j] then
18:                     d[i, j] := d[i, k] + d[k, j]
19:                     π[i, j] := π[k, j]
20:                 end if
21:             end for
22:         end for
23:     end for
24: end function
```

10.8 Dijkstra's Shortest Path

Dijkstra, my beloved.

Dijkstra's shortest path algorithm finds the shortest path to all nodes in a graph from a source vertex. It's easy to modify this for a target, just stop the algorithm once the target is seen.

Dijkstra is greedy and similar to Floyd-Warshaw, it doesn't work with negative paths.

Dijkstra's runtime is heavily dependent on the data structures used to implement it. Below, a priority queue was used.

Algorithm 17 Dijksta's Algorithm

```
1:  function DIJKSTRA(G(V, E), s) // ∈ O(VE)
2:      for ∀v ∈ V do
3:          π[v] := ∅
4:          d[v] := ∞
5:      end for
6:      d[s] := 0
7:      PQ := MinQ(d)
8:      while not EMPTY?(PQ) do
9:          x := POP(PQ)
10:         for ∀v ∈ adj(x, G) do
11:             if d[x] + wt(x, v) < d[v] then
12:                 π[v] := x
13:                 d[v] := d[x] + wt(x, v)
14:             end if
15:         end for
16:     end while
17: end function
```