1 Big O Notation

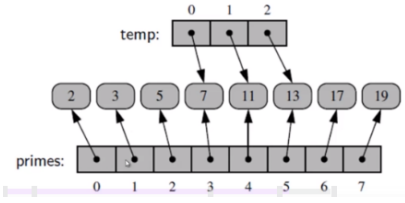
- Asymptotic Notation - measures time and space complexity of an algorithm

Name	Magnitude
Constant	$O(1)$
Logarithmic	$O(logn)$
Linear	$O(n)$
Superlinear	$O(nlogn)$
Polynomial	$O(n^c)$
Exponential	$O(c^n)$

- Big O - upper bound (worst case)
- Big Ω - lower bound (best case)
- Big Θ - tight bound (avg case)

2 Array Sequences

- Introduction
  - Byte is a memory address
  - Byte = 8 bits
  - Unicode char = 16 bits
  - Computer's main memory performs as random access memory (RAM). RAM allows any byte of main memory to be efficiently accessed.
- Low Level Arrays
  - Array - group of related variables stored in a contiguous portion of memory.
  - Each cell uses the same number of bytes. Hence, any cell can be accessed in constant time. $value = start + (cellsize)(index)$
  - Referential Array - each element refers to the object, this way each element is still the same size. Multiple elements can refer to the same object. An object can be shared by multiple lists.



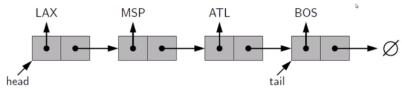
- Dynamic Array
  - Don't need to specify how large an array is beforehand. A list has greater capacity than current length. New array should have twice the capacity of the existing array.

3 Stacks, Queues, and Deques

- Stack (LIFO, FILO)
  - ordered collection of items where addition/removal of items takes place at the same end
  - items closest to base have been there longest
  - LIFO - (last-in, first-out) most recent items, removed first
  - Applications: browser back button, back button pops out webpages pushed onto stack
- Queue (FIFO)
  - ordered collection of items where addition/removal of items takes place at opposite ends
  - FIFO - (first-in, first-out) oldest item is removed first
  - Enqueue - add a new to queue
  - Dequeue - remove front item of queue
  - Applications: movie ticket line, grocery line, ...
- Deque
  - ordered collection of items with two ends
  - new items added/removed at both ends

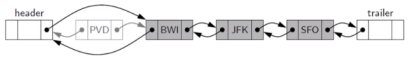
4 Linked Lists

- Singly Linked Lists
  - Collection of nodes that form a linear sequence.
  - Each node stores a reference to an object that is an element of the sequence and a reference to the next node of the list.



  - No predetermined fixed size
  - Pros: insertions/deletions $O(1)$
  - Cons: access/search takes $O(n)$
- Doubly Linked Lists
  - linked list in which each node keeps a reference to the node before/after it
  - sentinel - header/trailer nodes

  - inserting a node between two nodes is easy

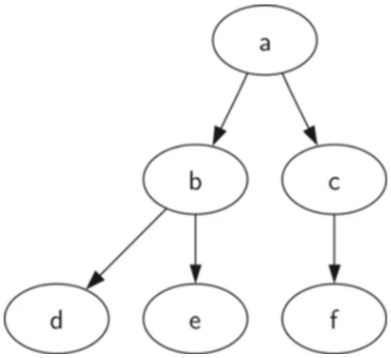


5 Recursion

- Two types
  - Function that calls itself
  - Data structure represented with same data structures
  - Always start with the base case
- Memoization
  - Remembering result of method calls by inputs and returning cached results rather than re-computing.

6 Trees

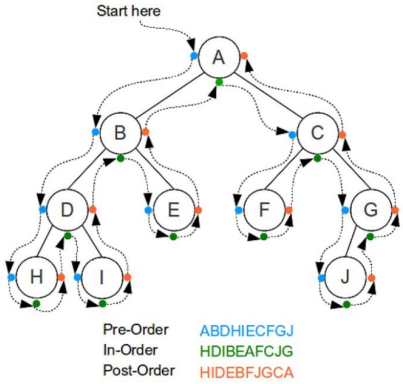
- What is a Tree?
  - Has roots, branches, and leaves
  - Ex: animal kingdom, file system, html, ...
  - Node - has a key (name) and payload (attributes)
  - Edge - connects two nodes to form a relationship. The root node is the only one with no incoming edges.
  - Path - ordered list of nodes connected by edges
  - Children - nodes that have incoming edges from the same node.
  - Parent - node with outgoing edges to children
  - Leaf - node with no children
  - Level - num of edges on path from root node to n
  - Trees - set of nodes and edges connected to each other. (1) One node is designated as root node, (2) every node except root is connected by at least one other node, (3) a unique path traverses from root to each node (4) binary tree - each node has max of two children
  - Recursive Definition - tree is empty or consists of nested subtrees
- Implement Trees as List of Lists
  - Store as [root, left subtree, right subtree]



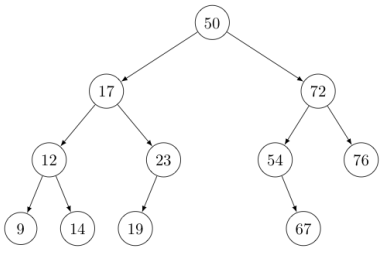

```
myTree = ['a', #root
            ['b', #left subtree
                ['d',[],[]],
                ['e',[],[]] ],
            ['c', #right subtree
                ['f',[],[]] ]
        ]
```

- Tree Traversals
  3 patterns to visit all nodes in a tree
  - Preorder: root, left preorder, right preorder
  - Inorder: left inorder, root, right inorder
  - Postorder: left postorder, right postorder, root
  - Remember "post/in/pre"

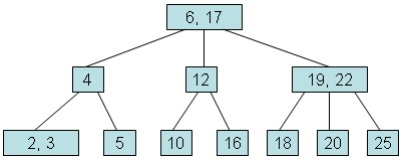
refers to the placement of processing the root node



- Priority Queues with Binary Heaps
  - Like a queue(i.e. FIFO) but order of items is determined by their priority
  - Highest at front, lowest at back
  - Binary Heap - $O(logn)$
    1. Create a Complete Binary Tree - Balanced Tree
    2. Order heap from smallest to largest
    3. If p = index of parent, index of children = (2p,2p+1)
- Binary Search Tree (BST)
  - 2 implementations of map ADT
    1. Binary Search on list
    2. Hash Tables
  - Three properties
    1. left subtree of a node contains only nodes with keys less than node's key
    2. right subtree of a node contains only nodes with keys greater than node's key
    3. left and right subtree each must also be a binary search tree

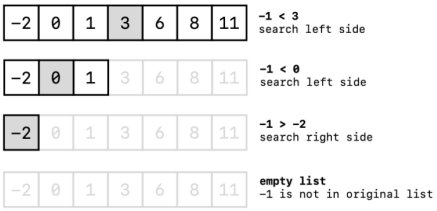


- B-Tree
  - a self-balanced tree in which every node has multiple keys and has more than two children
  - search/insertion is similar to BSTs

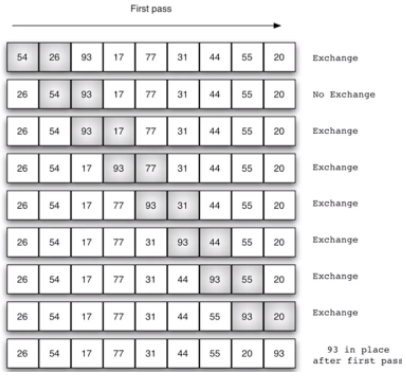


7 Searching and Sorting

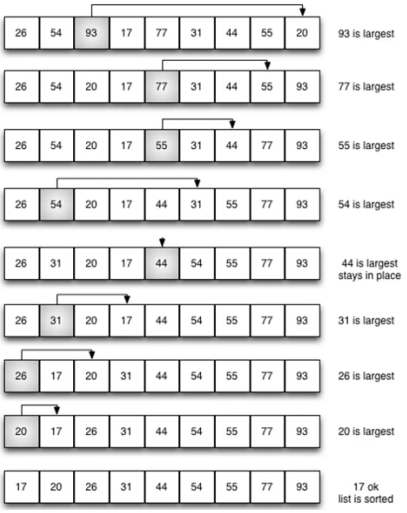
- Sequential Search
- Binary Search



  - Divide and Conquer - divide problem into smaller pieces, reassemble to get result
    iter 1, array length = n
    iter 2, array length = $n/2$
    iter 3, array length = $n/2 * (1/2)$
    after iter i, array length = $n/2^i$
    after i iter, array length = 1
    $1 = n/2^i$
    $2^i = n$
    $ilog(2) = log(n)$
    $i = log(n)$
- Hashing
  - hash table - collection of items stored in a way to make it easy to find later
  - hash function - mapping between item and slot where item belongs
  - perfect hash function - maps each item into a unique slot
  - load factor: tradeoff between time and space:
    $\lambda = numitems/tablesize$
    $\lambda = .75$ is a good rule of thumb
  - To search use hash function to lookup slot number and check
  - $O(1)$
- Hash Functions
  - Remainder
    $h(item) = item \,\%\, len(table)$
  - Folding
    1. Divide into equal-size pieces
    2. Add each piece
    3. Find remainder
    i.e. 436-555-4601
    (43,65,55,46,01)
    43+65+55+46+01 = 210
    210 % 11 = 1
  - Mid-Square
    1. Square item
    2. Extract middle r digits
    3. Find remainder
  - For strings
    1. Lookup ordinal value
    2. Add them up
    3. Find remainder
- Collision Resolution
  - linear probing - move sequentially and reassign collision item to first empty slot
  - chaining - items chained in same location
  - searching is difficult with either resolution technique
- Sorting
  - Visualization Resources
    - www.sorting-algorithms.com
    - visualgo.net/sorting.html
    - en.wikipedia.org/wiki/
  - Bubble sort
    - multiple passes through list
    - multiple exchanges per pass
    - compares adjacent items
    - largest bubbles to its place

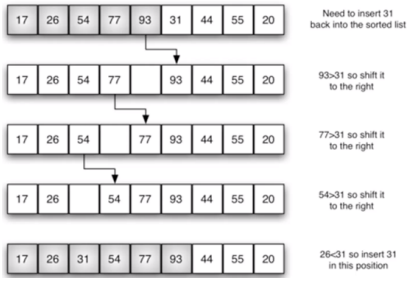


  - Selection sort
    - one exchange per pass
    - puts largest into proper place

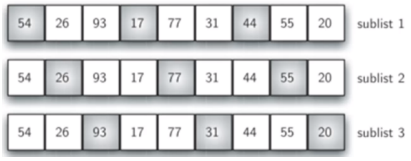


  - Insertion sort
    - check item with sorted sublist
    - shift items greater to the right
    - insert item when can't shift

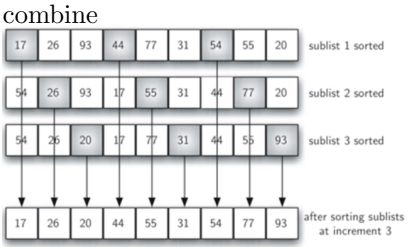
- how does insertion look like?



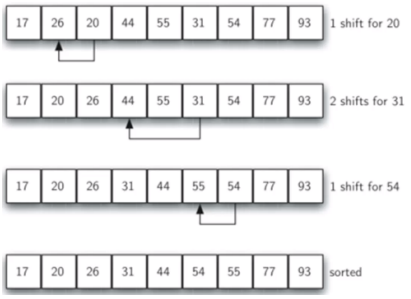
– Shell sort
- like insertion sort but with more sublists
- perform earlier sublist sorts
- reduces number of shifting operations
- i.e. sublist increment of 3



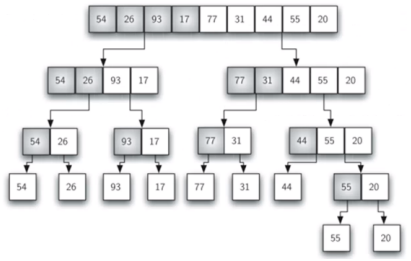
- i.e. insertion sort sublists and combine



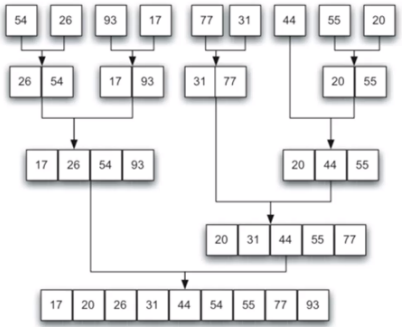
- i.e. insertion sort final list



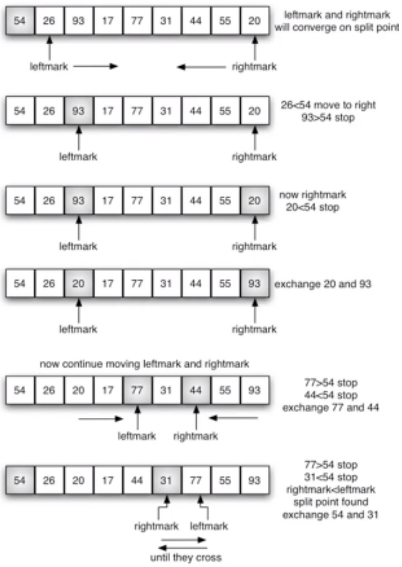
– Merge sort
- divide and conquer algorithm
- recursive splitting list in half
- sort each split
- merge back together
- i.e. recursive splitting



- i.e. merge and sort



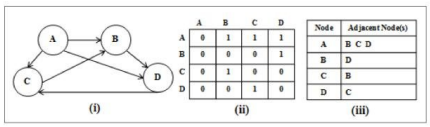
– Quick sort
- divide and conquer algorithm
- but doesn't use additional storage
- however list may not be divided in half



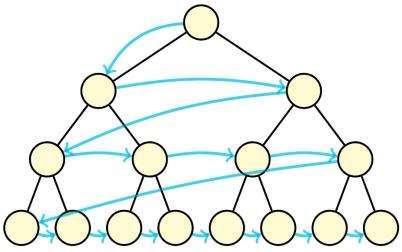
Name	Worst	Best
Bubble Sort	$O(n^2)$	$\Omega(n)$
Selection Sort	$O(n^2)$	$\Omega(n^2)$
Insertion Sort	$O(n^2)$	$\Omega(n)$
Shell Sort	$O(n*(log(n))^2)$	$\Omega(n*log(n))$
Merge Sort	$O(n*log(n))$	$\Omega(n*log(n))$
Quick Sort	$O(n^2)$	$\Omega(n*log(n))$

8 Graphs

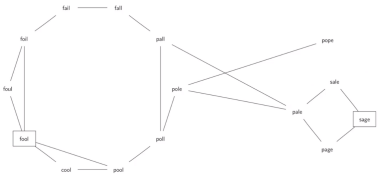
- Graphs have nodes(vertices), edges that connect the nodes, and optionally weights to show there is a cost to go from one vertex to another
- $G = (V, E)$ where w,v∈V
- A path is a sequence of vertices connected by edges
- They are represented in three ways: (from left to right: graph, adjacency matrix, adjacency list)



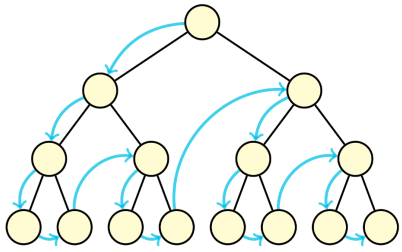
- Adjacency matrix is more time efficient. The adjacency matrix is simple and a good implementation when the number of edges is large. It is mostly empty (i.e. sparse).
- An adjacency list is more space efficient and allows us to compactly represent a sparse graph
- A cycle in a directed graph (digraph) is a path that starts/ends at the same vertex
- Breadth First Search (BFS)
  – Builds a search tree one level at a time
  – Great explanation: https://www.youtube.com/watch?v=s-CYnVz-uh4



  – Application: Word Ladder Problem
    * Transorm the word "FOOL" into the word "SAGE". You can only change one letter at a time. At each step, you must transform one word into another word. You cannot transform a word into a non-word
    * Solution: Use BFS to find an efficient path from starting to ending word. Get a list of all four letter words. Replace one of the letters with an underscore and move words that match the pattern into the appropriate bucket.

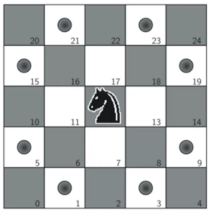
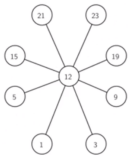


- Depth First Search (DFS)
  – DFS builds a search tree by exploring one branch of a tree as deeply as possible.



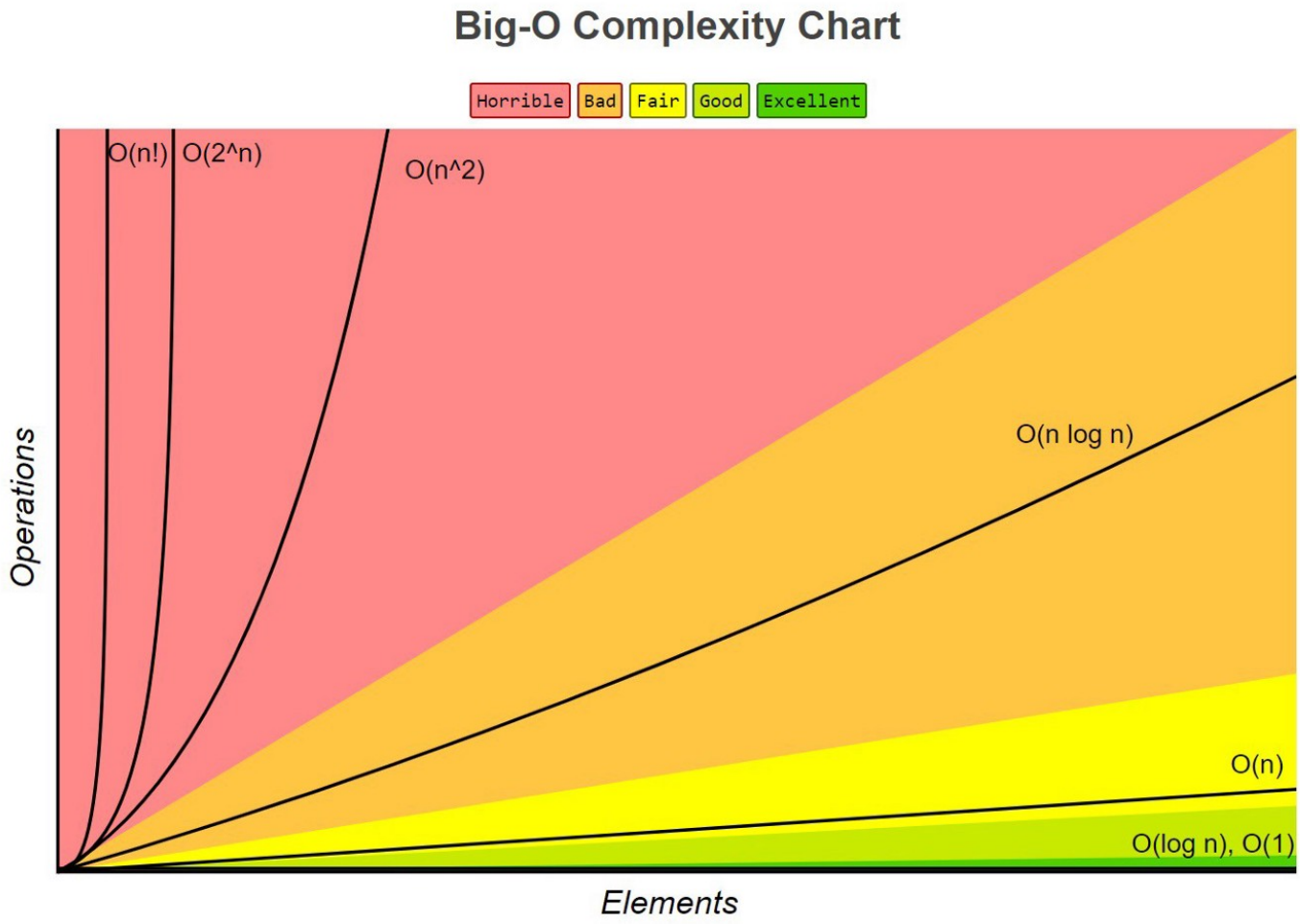
  – Knight's Tour Example Problem
    * Suppose you have a chess board with a single chess piece, the

knight. Find a sequence of moves that allows the knight to visit every square on the board exactly once.
    - Solution: Represent the moves of a knight on a chessboard as a graph. Use DFS to find a path of length rows x columns - 1

- Use BFS over DFS when ...
  - the solution is not far from the root
  - the tree is very deep and solutions are rare
- Use DFS over BFS when ...
  - the tree is very wide. BFS might need too much memory
  - solutions are frequent but located deep in the tree

9 Appendix



Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)