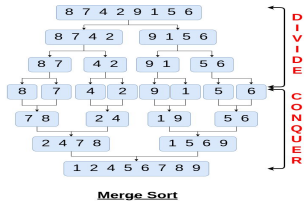
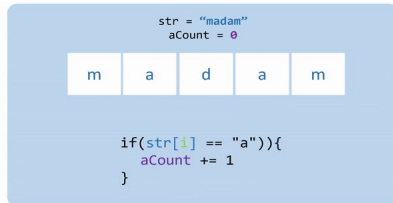


problem solving patterns

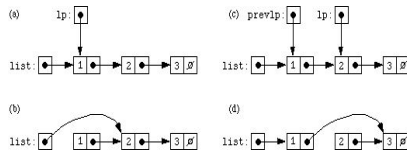
-- **divide and conquer** - divide in half, initialize a min max then move the pointer until the result is found.



-- **frequency counter** - making object counters then compare them. i.e. anagrams -> {a: 0, n: 0, g: 0, r: 0, m: 0, s: 1}

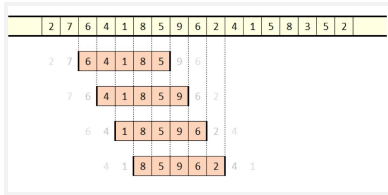


-- **multiple pointers** - move multiple pointers. arr[i], arr[j]



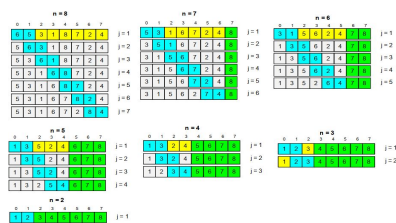
-- **sliding windows** - create temporary variable then update it according to logic. i.e. temp_var = max_var or

a,
b = a, b+a

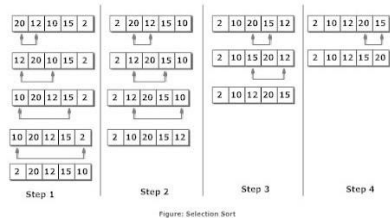


search algo

bubble sort -- brute force, comparing current and next element linearly

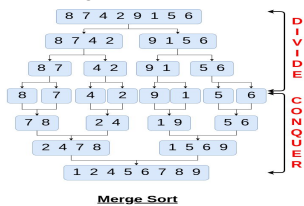


selection sort - compare current element to next then move the pointer to the next then compare again and so on..

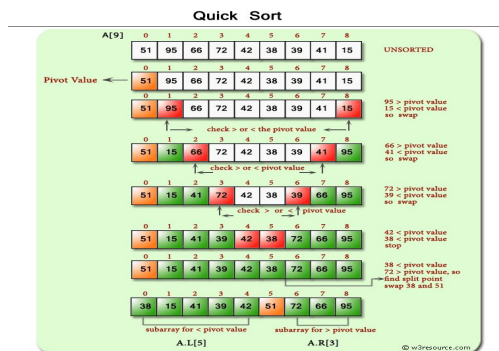


Advanced sorting:

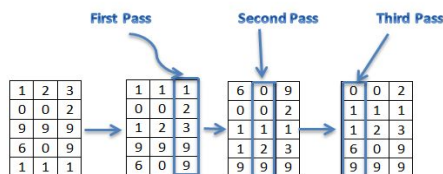
merge sort - divide array in half until there's one element per array then compare the next array element then merge them



quick sort - pick any element (use pivot helper) then find all elements that are less than it, put it to the left then greater than elements to the right, repeat. Pivot helper --where the rest of the sorted element will depend



radix sort - looking at the digit from the right then group them in every element bucket and



data structures

singly linked list - object consists of head and tail node with next attribute node and a tail that has a next attribute of null.

```
-- SinglyLinkedList {head: Node { val: 100, next: Node { val: 201, next: [Node] } },  
                    tail: Node { val: 999, next: null }, length: 5}
```

doubly linked list - object consist of head and tail node with next and prev attribute node and a tail that has a next attribute of null

```
-- DoublyLinkedList { head: Node { val: 'Harry', next: Node { val: 'Ron', next: [Node], prev: [Circular] },  
prev: null },          tail: Node {val: 'Hermione', next: null, prev: Node { val: 'Ron', next: [Circular],  
prev: [Node] } },      length: 3 }
```

stacks - FILO

```
-- Stack {  
  first: Node { value: 3, next: Node { value: 2, next: [Node] } },  
  last: Node { value: 1, next: null },  
  size: 3}
```

queues - FIFO

```
-- Queue {  
  first: Node { value: 1, next: Node { value: 2, next: [Node] } },  
  last: Node { value: 3, next: null },  
  size: 3}
```

binary search tree - parents nodes are greater than children, left child is always less than the parent and right are greater than the parent

```
-- BinarySearchTree {  
  root: Node {  
    value: 10,  
    left: Node { value: 5, left: [Node], right: [Node] },  
    right: Node { value: 13, left: [Node], right: [Node] }  
  }  
}
```

tree traversal bfs - searching the tree per each row, from left to right

```
--data [ 10, 6, 15, 3, 8, 20 ]
```

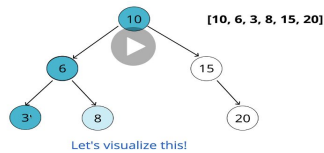
```
BinarySearchTree {  
  root: Node {  
    value: 10,  
    left: Node { value: 6, left: [Node], right: [Node] },  
    right: Node { value: 15, left: null, right: [Node] }  
  }  
}
```

tree traversal dfs preorder - traverse from root to the deepest child then its parent then the deepest child and so on.

data [10, 6, 15, 3, 8, 20]

```
BinarySearchTree {  
  root: Node {  
    value: 10,  
    left: Node { value: 6, left: [Node], right: [Node] },  
    right: Node { value: 15, left: null, right: [Node] }  
  }  
}
```

DFS - PreOrder

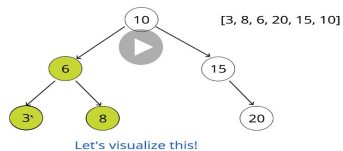


tree traversal dfs postorder - traverse from the deepest child then its parent then the deepest child and so on

data [3, 8, 6, 20, 15, 10]

```
BinarySearchTree {  
  root: Node {  
    value: 10,  
    left: Node { value: 6, left: [Node], right: [Node] },  
    right: Node { value: 15, left: null, right: [Node] }  
  }  
}
```

DFS - PostOrder



tree traversal dfs preorder - traverse the deepest child, sibling then its parent then the child and so on

data [3, 6, 8, 10, 15, 20]

```
BinarySearchTree {  
  root: Node {  
    value: 10,  
    left: Node { value: 6, left: [Node], right: [Node] },  
    right: Node { value: 15, left: null, right: [Node] }  
  }  
}
```

data [3, 8, 6, 20, 15, 10]

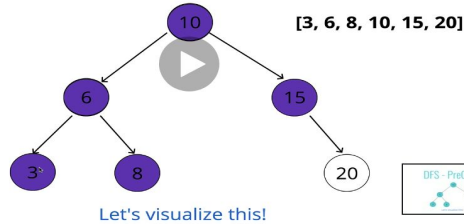
```
BinarySearchTree {  
  root: Node {
```

```

value: 10,
left: Node { value: 6, left: [Node], right: [Node] },
right: Node { value: 15, left: null, right: [Node] }
}
}

```

DFS - InOrder

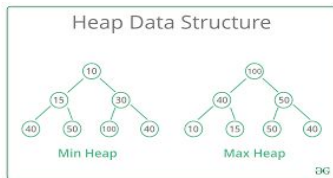


binary heaps - maxbh-> parents > children, minbh -> parents < children, no order for children

```

MaxBinaryHeap { values: [
  55, 39, 41, 18,
  27, 12, 33
]}

```

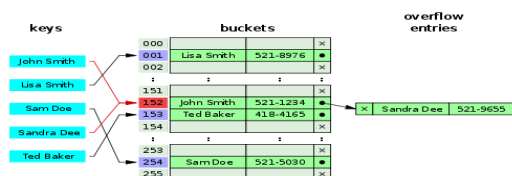


hash table/maps - used to store key value pairs

```

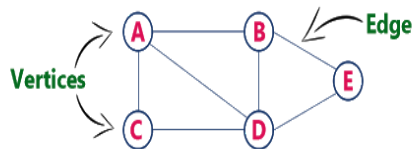
HashTable {
  keyMap: [
    [ [ 'plum', '#DDA0DD' ] ],
    <2 empty items>,
    [ [ 'salmon', '#FA8072' ] ],
    [ [ 'violet', '#DDA0DD' ] ],
    <2 empty items>,
    [ [ 'purple', '#DDA0DD' ] ],
    [ [ 'maroon', '#800000' ], [ 'yellow', '#FFFF00' ] ],
    <1 empty item>,
    [ [ 'olive', '#808000' ] ],
    <2 empty items>,
    [ [ 'lightcoral', '#F08080' ] ],
    <2 empty items>,
    [ [ 'mediumvioletred', '#C71585' ] ]
  ]
}

```

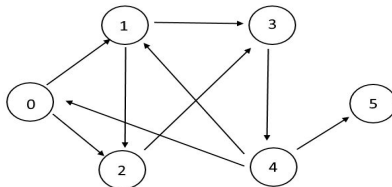


graph - consists of vertices(nodes) and edges

```
let g = new Graph();
g.addVertex("Dallas");
g.addVertex("Tokyo");
g.addVertex("Aspen");
g.addVertex("Los Angeles");
g.addVertex("Hong Kong");
g.addEdge("Dallas", "Tokyo");
g.addEdge("Dallas", "Aspen");
g.addEdge("Hong Kong", "Tokyo");
g.addEdge("Hong Kong", "Dallas");
g.addEdge("Los Angeles", "Hong Kong");
g.addEdge("Los Angeles", "Aspen");
Graph {
  adjacencyList: {
    Dallas: [ 'Tokyo', 'Aspen', 'Hong Kong' ],
    Tokyo: [ 'Dallas', 'Hong Kong' ],
    Aspen: [ 'Dallas', 'Los Angeles' ],
    'Los Angeles': [ 'Hong Kong', 'Aspen' ],
    'Hong Kong': [ 'Tokyo', 'Dallas', 'Los Angeles' ]
  }
}
```

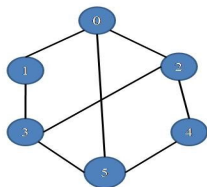


graph dfs - traverse through the deepest line then again on the next child



Depth First Traversal - 0 1 3 4 5 2

graph bfs - traverse through all children then the children of the next children



dijkstra - shortest way to travel from a path to another

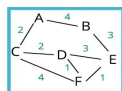
Dijkstra's Pseudocode

- This function should accept a starting and ending vertex
- Create an object (we'll call it distances) and set each key to be every vertex in the adjacency list with a value of infinity, except for the starting vertex which should have a value of 0.
- After setting a value in the distances object, add each vertex with a priority of infinity to the priority queue, except the starting vertex, which should have a priority of 0 because that's where we begin.
- Create another object called previous and set each key to be every vertex in the adjacency list with a value of null.
- Start looping as long as there is anything in the priority queue
 - dequeue a vertex from the priority queue
 - if that vertex is the same as the ending vertex - we are done!
 - Otherwise loop through each value in the adjacency list at that vertex
 - Calculate the distance to that vertex from the starting vertex
 - If the distance is less than what is currently stored in our distances object
 - update the distances object with new lower distance
 - update the previous object to contain that vertex
 - enqueue the vertex with the total distance from the start node

```
WeightedGraph {
  adjacencyList: {
    A: [ { node: 'B', weight: 4 }, { node: 'C', weight: 2 } ],
    B: [ { node: 'A', weight: 4 }, { node: 'E', weight: 3 } ],
    C: [
      { node: 'A', weight: 2 },
      { node: 'D', weight: 2 },
      { node: 'F', weight: 4 }
    ],
    D: [
      { node: 'C', weight: 2 },
      { node: 'E', weight: 3 },
      { node: 'F', weight: 1 }
    ],
    E: [
      { node: 'B', weight: 3 },
      { node: 'D', weight: 3 },
      { node: 'F', weight: 1 }
    ],
    F: [
      { node: 'C', weight: 4 },
      { node: 'D', weight: 1 },
      { node: 'E', weight: 1 }
    ]
  }
}
```

['A', 'C', 'D', 'F', 'E']

Step1: - pick the smallest



FIND THE SHORTEST PATH
FROM A TO E

Vertex	Shortest Dist From A
A	0
B	Infinity
C	Infinity
D	Infinity
E	Infinity
F	Infinity

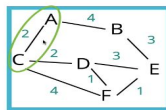
Visited:

[]

Previous:

```
{
  A: null,
  B: null,
  C: null,
  D: null,
  E: null,
  F: null
}
```

Step 2 - look at each of its neighbors then repeat on all its children



FIND THE SHORTEST PATH
FROM A TO E

Pick The Smallest...A

Vertex	Shortest Dist From A
A	0
B	Infinity, 4
C	Infinity
D	Infinity
E	Infinity
F	Infinity

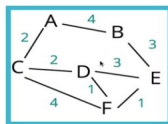
Visited:

[A]

Previous:

```
{
  A: null,
  B: A,
  C: null,
  D: null,
  E: null,
  F: null
}
```

Step 3 - look at the distance in the end



FIND THE SHORTEST PATH
FROM A TO E

Pick The Smallest...E

Vertex	Shortest Dist From A
A	0
B	Infinity, 4
C	Infinity, 2
D	Infinity, 4
E	Infinity, 7, 6
F	Infinity, 5

Visited:

[A, C, B, D]

Previous:

```
{
  A: null,
  B: A,
  C: A,
  D: C,
  E: F,
  F: D
}
```