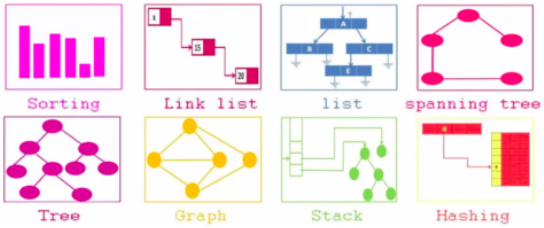
What are Data Structures?

-A method of storage (memory) that provides through a set of operations (add, del, access, search, sorting) functionality to manipulate the data in some useful way.

-Efficiency? Space &/VS Time



-Difference programs have different optimal data structures and algorithms.

-Java Difference

-Java is safe, you don't have direct access to memory (Objects are how you allocate memory), unlike C.

-Selecting a Data Structure

-1. Analyze the problem to determine the resource constraints a solution must meet.

-2. Determine the basic operations that must be supported.

-3. Quantify the resource constraints for each operation.

-Some questions to ask

-Thread safe?

-Data deletable?

-Is access random or well-defined?

-Data Structure Philosophy

-Each data structure has costs and benefits

-Rarely is one data structure is way superior to the other.

-Each requires:

-Space for storage

-Time to perform actions

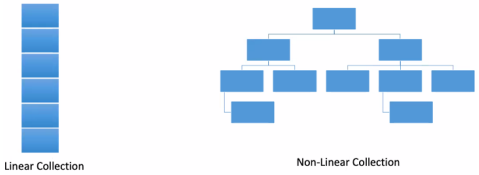
-Effort to code it in

-Time is another constraint

-Some actions takes way longer than another, like closing a bank account compared to deposit/withdraw

Collections

-A collection is an object that gathers and organizes other objects



-Linear (Static, can't reallocate memory)

-Array

-Non-Linear (Dynamic, can change memory)

-Map, LinkedList

-A collection is an abstraction that defines the interface operations through which the user can manage the objects within the collection.

-You can make a LinkedList with a set of an array or whatever, it is a correct collection if it behaves like a LinkList.

Abstract Data Types

-An abstract data type (ADT) is a description of a data structure with no implementation details.

-The API defines the different ADTs

-This description, often called an application programming interface (API), describes how the data structure is supposed to operate without suggesting any implementation details like architecture, language, or organization of the data and algorithms.

-If the API omits some details, the coder has to fill in the gaps, needing to satisfy the API.

-Efficiency & speed is preferred

-Implementations of a given data type can differ from one another

-They may make use of different architecture with different built in basic operations.

-They may be coded in different programming languages using different conventions.

-They may use different data organization and algorithms. (ie sorting algorithm be like)

Common Data Structures

- Lists
  - Arrays, Linked Lists, Stacks, Queues, Self-Organizing
- Trees
  - Binary Search Trees, Self-Balancing
  - B-Trees
  - Huffman Trees
  - Quad-Trees, kd-Trees
- Dictionary/Map
  - Hash tables
- Graph
  - Undirected, Directed, Weighted

-Invariant

-A property or set of properties that cannot vary anywhere in the data structure.

List

- What are the abstract expectations we have for a list?
  - List Invariant: A list is a collection of objects in *some* order.
    - Invariant: A property or set of properties that cannot vary or change in the data structure.
  - Every list will have the structure implicit in the list invariant.
    - This means every list will be a collection of objects that is stored in some order.
  - A list will be further defined by its access criteria through additional invariants:
    - Array (Vector) Invariant : index access
    - Linked List Invariant : iterator access
    - Stack Invariant: first-in last-out
    - Queue Invariant: first-in first-out

Algorithm Efficiency

-Time Complexity

-The number of primitive operations done in a block of code, which depends on the size of the input data.

-aka Running Time / Order of Growth

-depends on the time it takes to solve a problem

-depends on the size of the data

-Ex:

-Sorting a list of length n

-Searching a list of length n

-Multiplying a n×n matrix by an n×1 vector

-Cases

-Average-case time

-Best-case time (shortest), the lower bound / Omega

-Worst-case time (longest), the upper bound / Big-O

-Order Notation (eww 321)

- **BIG-O** $T(n) = O(f(n))$
  - Upper bound
  - Exist constants c and n_0 such that
    $T(n) \le c\, f(n)$ for all $n \ge n_0$
- **OMEGA** $T(n) = \Omega(f(n))$
  - Lower bound
  - Exist constants c and n_0 such that
    $T(n) \ge c\, f(n)$ for all $n \ge n_0$
- **THETA** $T(n) = \theta(f(n))$
  - Tight bound
  - $\theta(n) = O(n) = \Omega(n)$

-We keep track of "growth rates" of the computational workload as a function of problem dimension, so Big-O is most important.

Given two functions, *f* and *g*, say that "*f* is of order *g*" if

- there is a constant *c*, and
- a value n_0

such that $f(n) \le c.g(n)$ for all $n \ge n_0$

So, apart from a fixed multiplicative constant, the function *g* is an

- upper bound on the function *f*
- valid for large values of its argument.

Notation: write $f = O(g)$ to mean "*f* is of order *g*".

Sometimes we write $f(n) = O(g(n))$ to remind us what the arguments are labeled.

-Steps to determine efficiency

-1. Determine the time for each step / executable statement (like a constant c_1 for a statement)

-2. Sum up all independent statements / nested loops -> multiply (summation)

-Ex: Running Time: N-by-N matrix multiplying N-by-1 vector

```
Y = zeros(N,1);
for i=1:N                    i<N, i++
    Y(i) = 0.0;              c3
    for j=1:N                j<N, j++
        Y(i) = Y(i) + A(i,j)*x(j);   c4
    end                      c5
end                          c5
```

$$\text{Total} = c_1N + c_2N + N(c_3 + c_2N + N(c_4 + c_5) + c_5)$$
$$= (c_2 + c_4 + c_5)N^2 + (c_1 + c_2 + c_3 + c_5)N$$
$$= c_6N^2 + c_7N$$

-We count the constant steps like initializing the for loop counter and whatnot.

-Duplicate statements should be take same time, so some constant

-Different language have different statements (ie "end"), so the constant is only applicable where the compiler actually acknowledge it.

-That means constants are kinda useless.

Total = $\cancel{c_6}N^2 + \cancel{c_7}N$

- Problem size affects (is, in fact) N
- Processor speed, processor and language architecture, ie., technology, affect c_6 and c_7

Hence, "this algorithm of matrix-vector multiply has $O(N^2)$ complexity."

-Big O Sakpal way

-Disregard the constant/coefficients because it doesn't really matter

-Then show if $f(n) \le g(n)$ by showing $n \le n,\ n^2 \le n^2,\ n^3 \le n^3, \ldots$ whatever the highest exponent is.

$f(n) = 3n$
$g(n) = n$
$3n <= c \cdot n$

$n = 0$
$0 <= 0$

let's say $c = 5$
$f(n) <= c \cdot g(n)$

$n = 1$
$3 <= c \cdot 1$ → $3 <= 5$ T

$n = 2$
$6 <= c \cdot 2$ → $6 <= 10$ T

$n = 3$
$9 <= c \cdot 3$ → $9 <= 15$ T

→ (discard the constant)

let's say $c = 2$
$f(n) >= c\, g(n)$
$3 <= 2$ F

$f(n) = n$
$g(n) = n$ → disregarded
$f(n) <= 1 \cdot g(n)$
$f(n) <= g(n)$
$n <= n$
$n = 1$ $1 <= 1$
$n = 2$ $2 <= 2$
$n = 3$ $3 <= 3$
$n = 4$ $4 <= 4$
$f(n) <= g(n)$ $\therefore f(n) = O(g(n)) = 3n = O(n)$

$f(n) = 3n$
$g(n) = n^2$
$n <= n^2$?
$n = 1$ $1 <= 1$
$n = 2$ $2 <= 4$
$n = 3$ $3 <= 9$
$f(n) <= g(n)$
$\therefore f(n) = O(n^2)$

Example: $n^2 + n = O(n^3)$

$\le c\, n^3$

Proof:

- Here, we have $f(n) = n^2 + n$, and $g(n) = n^3$
- Notice that if $n > 0,\ n \le n^3$ is clear.
- Also, notice that if $n > 0,\ n^2 \le n^3$ is clear.

Side Note: In general, if $a \le b$, then $n^a \le n^b$ whenever $n > 0$. This fact is used often in these types of proofs.

- Therefore,

$$n^2 + n \le n^3 + n^3 = 2n^3$$ Donald's way!

- We have just shown that $n^2 + n \le 2n^3$ for all $n > 0$

Thus, we have shown that $n^2 + n = O(n^3)$
(by definition of Big-O, with $n_0 = 0$, and $c = 2$.)

$n^2 + n = O(n^2)$
$f(n) = n^2 + n$
$g(n) = n^2$
$n <= n^2$
$1 <= 1^2$
$2 <= 4$
$3 <= 9$
$n^2 <= n^2$
$1 <= 1$
$2^2 <= 2^2$
$3^2 <= 3^2$
$n^2 + n <= n^2 + n^2$
$n^2 + n <= 2n^2$
$f(n) <= c\, g(n)$
$\therefore \boxed{n^2 + n = O(n^2)}$

$n^2 + n = O(n)$
$f(n) = n^2 + n$
$g(n) = n$
$n <= n$ T
$1 <= 1$
$2 <= 2$
$3 <= 3$
$n^2 <= n$? F
$1 <= 1$
$2^2 > 2$
Thus $n^2 + n \not<= n$
$n^2 + n \neq O(n)$

-How to talk about time complexity

- Algorithmic complexity is usually expressed in 1 of 2 ways.
  - The first is in terms of running time - "logarithmic", "linear", etc.
  - The other is called Big-O notation. This is a more mathematical way of expressing running time, and looks more like a function.

For example,

- a "linear" running time can also be expressed as O(n).

- Similarly, a "logarithmic" running time can be expressed as O(log n).

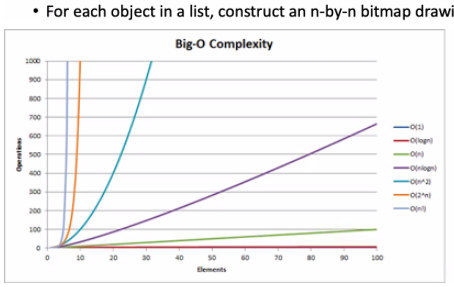
Constant	O(1)
Logarithmic	O(log n)
Linear	O(n)
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$O(2^n)$

-Common Time Complexity and Examples

- A constant-time algorithm is one that takes the same amount of time, regardless of its input. Here are some examples:
  - Given two numbers* , report the sum
  - Given a list, report the first element
  - Given a list of numbers* , report the result of adding the first element to itself 1,000,000 times
- A linear-time algorithm is one that takes a number of steps directly proportional to the size of the input.
- In other words, if the size of the input doubles, the number of steps doubles.
- Examples:
  - Given a list of words, print each item of a list
  - Given a list of numbers, add each pair of numbers together (item 1 + item 2, item 3 + item 4, etc.)
  - Given a list of numbers, multiply every 3rd number by 2
- A cubic-time algorithm is one that takes a number of steps proportional to n^3 . In other words, if the input doubles, the number of steps is multiplied by 8.
- Similarly to the quadratic case, this could be the result of applying an n^2 algorithm to n items, or applying a linear algorithm to n^2 items.
- Examples:
  - Fill in a 3D board (or environment)
  - For each object in a list, construct an n-by-n bitmap drawing of the object

- A logarithmic-time algorithm is one that requires a number of steps proportional to the log(n).
- In most cases, we use 2 as the base of the log, but it doesn't matter which base because we ignore constants. Because we use the base 2, we can rephrase this in the following way: every time the size of the input doubles, our algorithm performs one more step.
- Examples:
  - Binary search
  - Searching a tree data structure (we'll see what this is later)
- A quadratic-time algorithm is one takes a number of steps proportional to n^2. That is, if the size of the input doubles, the number of steps quadruples.
- A typical pattern of quadratic-time algorithms is performing a linear-time operation on each item of the input

  (n steps per item * n items = n^2 steps).
- Examples:
  - Compare each item of a list against all the other items in the list
  - Fill in a n-by-n game board
- An exponential-time algorithm is one that takes time proportional to 2^n . In other words, if the size of the input increases by one, the number of steps doubles.
- Note that logarithms and exponents are inverses of each other.
- Algorithms in this category are often considered too slow to be practical, especially if the input is typically large.
- Examples:
  - Given a number n, generate a list of every n-bit binary number



Running Time	Growth Order – Big O()	Label
t(n) = 17	O(1)	constant
t(n) = 3log n	O(log n)	logarithmic
t(n) = 20n – 4	O(n)	linear
t(n) = 12n log n + 100n	O(n log n)	n log n
t(n) = $3n^2 + 5n - 2$	$O(n^2)$	quadratic
t(n) = $8n^3 + 3n^2$	$O(n^3)$	cubic
t(n) = $2^n + 18n^2 + 3n$	$O(2^n)$	exponential

Task	Growth rate
Matrix/vector multiply	$O(n^2)$
Getting a specific element from a list	$O(1)$
Dividing a list in half, dividing one halve in half, etc	$O(\log_2 n)$
Binary Search	$O(\log_2 n)$
Scanning (brute force search) a list	$O(n)$
Nested **for** loops (k levels)	$O(n^k)$
MergeSort	$O(n \log_2 n)$
BubbleSort	$O(n^2)$
Generate all subsets of a set of data	$O(2^n)$
Generate all permutations of a set of data	$O(n!)$

-Ex: Time Complexity where input being odd/even changes time complexity.

-But it's Big-O, so we look at the worst-case scenario.

Exercise

What is the time complexity of following code:

```
int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
    for (j = 2; j <= n; j = j * 2) {
        k = k + n / 2;
    }
}
```

$\rightarrow c = 1$

$\left(\frac{n}{2}+2\right)$ $(\log n)$

$\boxed{\left(\frac{n}{2}+2\right)}$

$c=1$

even $n = 8$

$\left(\frac{n}{2}+1\right)$

$i = 4$	$j = 2, 4, 8$
$i = 5$	$j = 2, 4, 8$
$i = 6$	$j = 2, 4, 8$
$i = 7$	$j = 2, 4, 8$
$i = 8$	$j = 2, 4, 8$

odd $n = 7$

$i = 3$	$j = 2, 4$
$i = 4$	$j = 2, 4$
$i = 5$	$j = 2, 4$
$i = 6$	$j = 2, 4$
$i = 7$	$j = 2, 4$

$n = 16$

$j = 2, 4, 8, 16$

$= 2^1, 2^2, 2^3, 2^4 \ldots$

$2^k = n$

$\therefore k = \boxed{\log_2 n}$

$f(n) = \left(\frac{n}{2}+2\right)(\log n)$

$= \frac{n}{\cancel{2}} \log n + \cancel{2} \log n$

$O(n \log n)$

-Ex: Time Complexity with While Loop

-Let n be even/odd, find them patterns

```
int a = 0, i = N;
while (i > 0) {
    a += i;
    i /= 2;
}
```

$C = 1$

$\rightarrow ?$

$C = 1$

$C = 1$

$n = 12$

$i = 12$

$i = 6$ $\frac{12}{2}$

$i = 3$ $\frac{6}{2}$

$i = 1$ $3/2$

$n = 9$

$i = 9 \rightarrow \frac{9}{2^1}$

$i = 4 \rightarrow \frac{9}{2^2}$

$i = 2 \rightarrow$

$i = 1 \rightarrow \frac{9}{2^3}$

$\frac{n}{2^1}$

$\frac{n}{2^2}$

$\frac{n}{2^3}$

$\frac{n}{2^k} := 1 \quad \therefore n = 2^k \quad \therefore k = \boxed{\log n}$

-So f(n) = O(logn)

-Ex: Time Complexity of Recurrence crap

```
T(n) = {  3T(n-1),   if n>0
       {  1,         otherwise
```

-Pick an n and simplify to generalize

$n = 4$

$T(4) = 3T(4-1) = 3T(3) \rightarrow 3T(n-1)$

$= 3[3T(3-1)] = 3^2 T(2) \rightarrow 3^2 T(n-1-1) = 3^2 T(n-2)$

$= 3^2[3T(2-1)] = 3^3 T(1) \rightarrow 3^3 T(n-2-1) = 3^3 T(n-3)$

$= 3^3[3T(1-1)] = 3^4 T(0) \rightarrow 3^4 T(n-3-1) = 3^4 T(n-4)$

$= 3^4$

$\vdots$

$3^n T(n-n) = \boxed{3^n}$

-Ex: 321 Big-O 2 functions cheese

Let $T_1(n) = 3n + 12$ and $T_2(n) = 4n^2 + 23n + 15$. Is $T_1(n) = O(T_2(n))$? Give brief explanation.

$T_1(n) <= T_2(n)$

-Fake induction

$f(n) <= c \cdot g(n)$

$n = 1$	
$3 + 12 <= 4 + 23 + 15$	T
$n = 2$	
$6 + 12 <= 4 \times 4 + 23 \times 2 + 15$	T
$n = 3$	
$18 + 12 <= 4 \times 9 + 23 \times 3 + 15$	T

(c=1, so k=1)

-Ex: Which is faster?

Algorithms A and B spend exactly $T_A(n) = 5n \log_{10} n$ and $T_B(n) = 25n$ microseconds, respectively, for a problem of size n. Which algorithm is better in the Big-O sense? For which problem sizes does it outperform the other ?

-T_B(n) is faster when $n \geq 10^5$. She did it by simplifying both sides to find n_o.

$T_B(n) <= T_A(n)$ $n \geq n_0$

$25n <= 5n \log_{10} n$

Divide by 5n on both sides

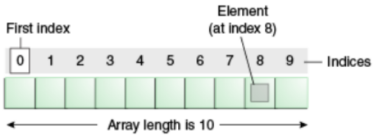
$5 <= \log_{10} n$

$10^5 <= n$

$\therefore$ $n \geq 10^5$ Algo B will outperform Algo A

Arrays Review

-Array be like



An array of 10 elements.

- Defining an Array – Size
- Accessing an element
- Searching for an element
- Adding(Insert) an element
  - Front, End, Middle
- Remove(Delete) an element
  - Front, End, Middle

-Static in memory, adjacent blocks of memory, fixed length/size

-Space: O(n), depending on size of the array.

-In Java, an array is an object. The 1st element of the array is at the memory address, so 0 offset and that's why index starts at 0.

(Suppose we have an array (arr[]) with size=10 and with 5 elements stored (n=5))

-Accessing Element O(1)

-Ez, get the element at the index. *array[i]*

-Search O(n)

```
search(T key) //Returns index of key element, or -1 if not found.
    for i -> 0 to n //From beginning to last element
        if key = arr[i] //if found,
            then return i //return index
    return -1 //not found
```

-If the array is sorted, use binary search for O(logn)

-Add O(1)

```
add(T element) //Adds the new element at the next available index.
    arr[n] : num
    n++ //increment counter of the amount of elements already in
```

-If we are implementing Java List, we need to implement capacity expansion, so time complexity will increase to O(n).

-If we had a sorted array and want to add the new element in order

-Determine the order by binary search

-Add it in and shift elements

-Insert O(n)

```
insert(T element, int index) //Inserts new element at specified index, shifting others down
    for i -> n to index //Start from end,
        arr[i] = arr[i-1] //shift elements down until where you want to insert
    arr[index] = num //insert at new spot
    n++ //increment counter of the amount of elements already in
```

-Deletion O(n)

```
delete(int index) //Delete the element at the specified index, shifting elements upward to fill the gap.
    temp = arr[index] //temporary of the element to delete (optional)
    for i -> index to n //From deletion index,
        arr[i] = arr[i+1] //shift all following elements upward
    n-- //decrement counter of the amount of elements already in
    arr[n] = null //If working with objects, this sets up for garbage collection and you won't have duplicate references/elements
    return temp //Returns the deleted element (optional)
```

-Same time complexity with sorted array

-We need to check for underflow here, removing from an empty array. (At the start, if n=0, then throw error)

Linked Lists

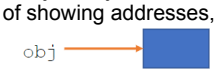
-What the link?

-A linked structure is a data structure that uses object reference variables to create links between objects.

-The following declaration creates a variable called obj that is only large enough to hold the numeric address of an object:

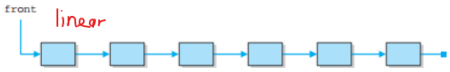
Object obj;

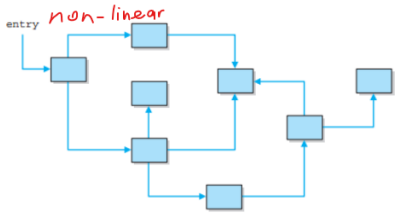
-Instead of showing addresses, we usually depict a reference variable as a name that "points to" an object.



-What the linked list?

-A linked list is a recursive dynamic data structure, in which the elements are not stored at contiguous (adjacent like array) memory locations. The elements in a linked list are linked using pointers as shown in the below image:





-Even though you can have a linear linked list, it's usually never linearly ordered in memory.

-Time Complexity (Un- and Sorted)

-Space: O(n), one for each node

-Access O(n)

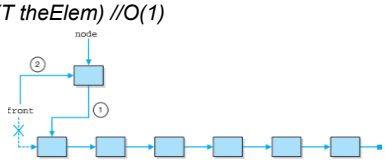
-Search O(n)
-Add Front/Read O(1)
-Insert O(n)
-Delete O(n)

-What a linked node

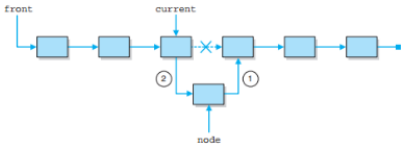
-A linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

-Defining the linked list (and node)

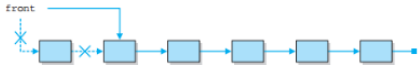
```
class SLL: //singly linked
        Node front = null
        Node rear = null
        int count = 0

        class Node: //inner class for encapsulation, so others can't mess with nodes which = memory.
                T elem
                Node next = null //Self referencing pointer
        public Node(T theElem)  //Constructor that assignes Node
                elem = theElem

        ////////////////////////Operations//////////////////////////
        addRear(T theElem) //O(1) since you know rear
                Node temp = new Node(theElem)
                if (count = 0) //If is empty
                        front = rear = temp //both front & rear is the new node
                else : //if not empty
                        rear.next = temp //add at end
                        rear = rear.next //update rear
                count++
        addFront(T theElem) //O(1)
```



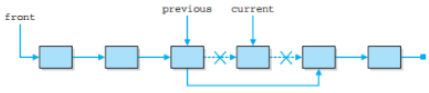

```
                Node temp = new Node(theElem)
                if (count = 0) //If is empty
                        front = rear = temp //both front & rear is the new node
                else: //if not empty
                        temp.next = front //link temp and front first
                        front = temp //update front (good rule is to not manipulate front until you really need to)
                        //Dummy Front Node? Can make handling front safer
        insert(T theElem, int theIndex) //O(n), still less than array insert because no shifting crap
```




```
                if (count = 0) //if empty, add to front/rear
                        addFront(theElem)
                        return;
                if (theIndex > (count - 1) or theIndex < 0) //Check bad index
                        throw error
                Node temp = new Node(theElem) //Node to insert
                Node current = front //Pointer
                        for i -> 0 to index //Go to index
                                current = current.next
                        //Inserting
                        temp.next = current.next
                        current.next = temp
                count++
        deleteByKey(T key) //O(n) (this method doesn't account if key is not found!)
                if (count = 0) //Check underflow
                        throw error //or whatver
                //Pointers
                Node current = front
                Node prev = null
                if (front.elem == key) //1st case, if front is key
```




```
                        front = front.next //change front to next, old front becomes garbage (unlike array which you need to manually do)
                else: //iterate and find element
```




```
                        while (current.elem != key && current != null)
                                prev = current
                                current = current.next
                        prev.next = current.next //found, then skip the node to delete
                        current.next = null //(optional) because it's garbage already
                count--
```

-Managing a linked list

-Accessing elements

```
get(int theIndex) //O(n) because the for loop
    if (theIndex > (count - 1) or theIndex > 0) //Check underflow and bad index
        throw error
    Node current = front //New pointer to iterate
    for i -> 0 to theIndex //loop getting next
        current = current.next
    return current //ez
search(T theElem) //O(n) because we iterate over entire list
    if (count = 0) //Check underflow
        throw error
    int returnIndex = -1
    Node current = front //New pointer to iterate
    //for i ->0 to count //alt using for loop
    while (current.elem != key && current != null)
        index++
        current = current.next
    if (current = null) //if reach past rear, key not found
        return -1
    return index
```

-LinkedList Leet Code kinda

Node reverse(Node head) //Given head, reverse the LinkedList O_t(n) & O_s(1) [Reverse a linked list - GeeksforGeeks](#)

```
//Pointers
Node curr = head
Node prev = null
while (curr != null) //Iterate through
    Node temp = curr.next //Remember curr.next for later
    curr.next = prev //Update curr.next to point to prev
    prev = curr //Move prev forward to curr
    curr = temp //Move curr forward to temp
head = prev //Update head to prev (because curr is null)
return head //Return head
```

boolean palindrome(Node next) //Stack cheese O_t(n) & O_s(n) [Function to check if a singly linked list is palindrome - GeeksforGeeks](#) *(reverse half, recursion)*

```
stack S
//Step 1: push all (because we don't know size)
Node curr = head
while (curr != null) //for i → 0 to count
    s.push(curr.elem)
//Step 2: Compare LinkedList (from head) with Stack (from lastest)
curr = head
while (curr != null)
    if (curr.element != s.pop()) //if not palindrome
        return false;
    curr = curr.next
```

void rotate(Node head, int k) //Rotate O_t(n) & O_s(1)

Given a singly linked list, write a function (pseudocode) to rotate the linked list counterclockwise by k nodes. Where k is a given positive integer. For example, if the given linked list is `10->20->30->40->50->60` and k is 4, the list should be modified to `50->60->10->20->30->40`. Assume that k is smaller than the count of nodes in the linked

```
Node curr = head
for i→0 to k //Go to index (k)
    curr = curr.next
Node temp = curr.next //Remeber the Node one after k
curr.next = null //Break k node's next
rear.next = head //Make the rear become head
head = temp //Update head to the Node one after k
rear = curr //Update rear to k Node
```

-Using fast and slow pointers

-You can use a pointer that moves through the list slower than the other.

-This usually means a ratio of speed. The fast pointer can be faster by 2x, doing .next.next

-With that 2:1 speed, the slow pointer will only be halfway when the fast one reaches end.

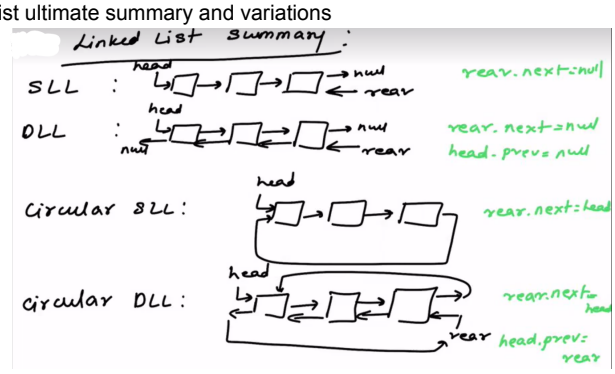
-What the Doubly Linked List?

-Unlike the normal singly linked list, there is a "next" and a "prev" reference

```
class DLL: //O(n) for accessing, search
    Node front = null
    Node rear = null
    int count = 0
        class Node: //inner Node class
        T elem
        Node next = null
        Node prev = null
    ///////////Operations///////////
    addFront(T theElem) //O(1)
        Node temp = new Node(theElem)
        if (isEmpty)
            front = rear = temp
        else:
            temp.next = front
            front.prev = temp
            front = temp
        count++
    addRear(T theElem) //O(1)
        Node temp = new Node(theElem)
        if (isEmpty)
```

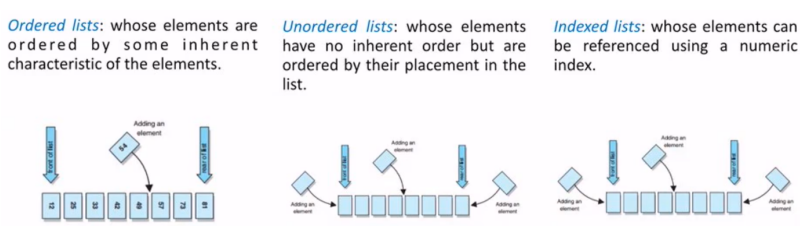
```
                front = rear = temp
        else:
                rear.next = temp
                temp.prev = rear
                rear = temp
        count++
insert(T theElem, int theIndex) //O(n)
        if (isEmpty)
                addFront(theElem) //or rear
        else:
                Node temp = new Node(theElem)
                //go to index
                Node curr = front //You can iterate from rear to save some time if in latter half
                for i → 0 to theIndex
                        curr= curr.next
                //after curr
                curr.next.prev = temp
                temp.next = curr.next
                //curr
                curr.next = temp
                temp.prev = curr
        count++
delete(T theKey) //O(n)
        //Check underflow (isEmpty then throw exception)
        if (front.elem == theKey)
                front = front.next
                front.prev = null
                return
        if (rear.elem == theKey)
                rear = rear.prev
                rear.next = null
                return
        //somewhere in the middle
        Node curr = front
        while (curr.elem != theKey && curr != null)
                curr.next.prev = curr.prev
                curr.prev.next = curr.next
        Node curr = front.next //start at 2nd node
        while (curr.elem != theKey && curr != rear) //iterate to Node before rear
                curr = curr.next
        //deletion steps
        curr.next.prev = curr.prev
        curr.prev.next = curr.next
        count++
```

-Linked List ultimate summary and variations



List ADT

-3 types



-Iterator

An iterator is an object that provides the means to iterate over a collection.

The two primary abstract methods defined in the Iterator interface are:

`hasNext`, which returns true if there are more elements in the iteration

`next`, which returns the next element in the iteration

-Common operations

-add, addFront, addRear, insert, delete, isEmpty

-Java API: https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

-Fat Time Complexity of Array & the 2 types of Linked Lists

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)
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)

-Array is faster to access & (if sorted) search, LinkedList is faster to insert & delete

Stack (On a)

-Items are removed from the stack in the reverse order from the way they were inserted.

-Last In First Out (LIFO) aka First In Last Out (FILO)

-Stacks are implemented by

-Arrays, LinkedLists

-Arrays

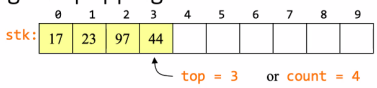
- To implement a stack, items are inserted and removed at the same end (called the top)
- Efficient array implementation requires that the top of the stack be towards the center of the array, not fixed at one end
- To use an array to implement a stack, you need both the array itself and an integer
  - The integer tells you either:
    - Which location is currently the top of the stack, or
    - How many elements are in the stack

-Having the top of the stack at the middle of the array avoids shifting when pop and push.

Stack-Array:

final static int SIZE = 10; //Size of array and subsequently Stack. Constructor Parameter ez.
T[] arr = new T[SIZE]
int top = -1 //Optional: int count = 0
///////Under&Overflow//////
boolean isEmpty() //underflow
return top == -1
boolean isFull() //overflow
return top == SIZE-1
////////////Operations////////////

Pushing and popping

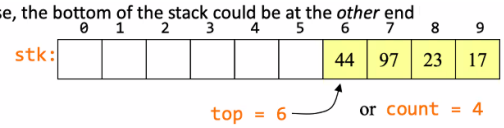


- If the bottom of the stack is at location 0, then an empty stack is represented by top = -1 or count = 0
- To add (push) an element, either:
  - Increment top and store the element in stk[top], or
  - Store the element in stk[count] and increment count
- To remove (pop) an element, either:
  - Get the element from stk[top] and decrement top, or
  - Decrement count and get the element in stk[count]

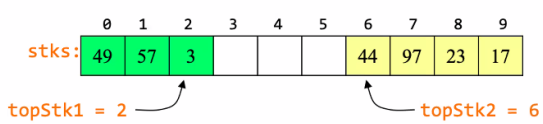
void push(T theElem) //O(1)
//check overflow & maybe resize
arr[++top] = theElem
T pop() //O(1)
//check underflow & maybe throw exception
T temp = arr[top]
arr[top] = null //useless on primitives, so use with objects
top--
return temp
T *peek() //O(1)*
//check underflow & maybe throw exception
return arr[top]

-2 Stacks, 1 Array

- Of course, the bottom of the stack could be at the *other* end



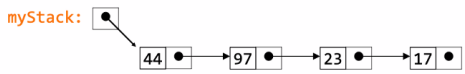
- Sometimes this is done to allow two stacks to share the *same storage area*



- There are two stack errors that can occur:
  - Underflow: trying to pop (or peek at) an empty stack
  - Overflow: trying to push onto an already full stack
- For underflow, you should throw an exception
  - If you don't catch it yourself, Java will throw an ArrayIndexOutOfBounds exception
  - You could create your own, more informative exception
- For overflow, you could do the same things
  - Or, you could check for the problem, and copy everything into a new, larger array

Stack-SLL:

- Since all the action happens at the top of a stack, a singly-linked list (SLL) is a fine way to implement it
- The header of the list points to the top of the stack



- Pushing is inserting an element at the front of the list
- Popping is removing an element from the front of the list

Node top = null //front
int count = 0
///////Undeflow///////
boolean isEmpty()

- With a linked-list representation, overflow will not happen (unless you exhaust memory, which is another kind of problem)
- Underflow can happen, and should be handled the same way as for an array implementation
- When a node is popped from a list, and the node references an object, the reference (the pointer in the node) does *not* need to be set to null
  - Unlike an array implementation, it really *is* removed--you can no longer get to it from the linked list
  - Hence, garbage collection can occur as appropriate

```
            if count == 0
    /////////Operations/////////
    void push(T theElem) //O(1)
            Node temp = new Node(theElem)
            if (isEmpty())
                    top = temp
            else:
                    temp.next = top
                    top = temp
            count++
    T pop() //O(1)
            //Check underflow & throw exception
            T result = top.elem
            top = top.next
            count--
            return result
    T peek() //O(1)
            //Check underflow & throw exception
            return top.elem
```

- Stack Applications
  - Tower of Hanoi
    - Ew
  - Evaluating Expression (Stack of operator)
    - Convert from infix input to postfix, then evaluated, both can use stacks

user compiler

Infix	Postfix
2+3*4	234*+
a*b+5	ab*5+
(1+2)*7	12+7*
a*b/c	ab*c/
(a/(b-c+d))*(e-a)*c	abc-d+/ea-*c*
a/b-c+d*e-a*c	ab/c-de*+ac*-

Postfix: no parentheses, no precedence

- Code
  - From left to right.
  - Write all numbers to output.
  - Stack of operators
  - If operator
    - If first operator case, push() it.
    - If new operator have less precedence than top (peek()).
      - pop() to the output until it is less or equal then push()
    - If new operator have equal precedence than top (peek()).
      - pop() and push() once.
    - If new operator have higher precedence than top (peek()).
      - push()
  - Brackets
    - push() the opening bracket
      - It has the lowest precedence, so keep push() new operator
    - When you reach closing bracket
      - pop() until opening
  - When input is empty, pop() and append operators to input.
- Ex:

$$A \cdot B - (C + D) + E$$

in	stack	out
$A \cdot B - (C + D) + E$		
$\cdot B - (C + D) + E$		A
$B - (C + D) + E$	$\cdot$	A
$-(C + D) + E$	$\cdot$	AB
$(C + D) + E$	$-$	$AB\cdot$
$(C + D) + E$	$-$	$AB\cdot$
$C + D) + E$	$-, ($	$AB\cdot$
$+D) + E$	$-, ($	$AB \cdot C$
$D) + E$	$-, (, +$	$AB \cdot C$
$) + E$	$-, (, +$	$AB \cdot CD$
$+E$	$+$	$AB \cdot CD+$
E	$+$	$AB \cdot CD + -$
	$+$	$AB \cdot CD + -E$
		$B \cdot CD + -E+$

- Evaluating postfix (Stack of numbers)
  - Read input Left to Right
  - Operands
    - push() into stack
  - Operator
    - pop() twice, perform calculation (2nd pop() operator 1st pop())
    - push() the result
  - Stack should have one left when String is empty, ending on doing the last operation.
  - Ex:

$$53 + 82 - \cdot$$

in	stack
$23 \cdot +$	
$234 \cdot +$	
$34 \cdot +$	2
$4 \cdot +$	$2, 3$
$\cdot +$	$2, 3, 4$
$+$	$2, 3 \cdot 4 = 12$
	$2 + 12 = 14$
return	14

in	stack
$53 + 82 - \cdot$	
$3 + 82 - \cdot$	5
$+82 - \cdot$	$5, 3$
$82 - \cdot$	$5 + 3 = 8$
$2 - \cdot$	$8, 8$
$- \cdot$	$8, 8, 2$
$\cdot$	$8, 8 - 2 = 6$
	$8 \cdot 6 = 48$
return	48

-There is also prefix. 2+3 → +23

-Evaluating Prefix

-Just the reversed of postfix, so reverse it and do postfix

-Not LeetCode Stacks

-Prints the reverse of the stack

```
void reverse(Stack s):
        Stack temp = new Stack() //to store s.pop()
                //pop() s
        while(!s.isEmpty()) //n
                temp.push(s.pop())
        end
        //print all of temp
        while (!temp.isEmpty())
                Character cTemp = temp.pop()
                System.out.print(cTemp)
                s.push(cTemp)
        end
end
void reverse(Stack s) //call on a STACK😂😐 recursion
if (s.size() == 1)
                System.out.print(s.dequeue())
        else:
                T temp = s.pop()
                reverse(s)
                System.out.print(temp)
                s.push()
        end
end
```

Queue

-Queues are implemented by

-Arrays (Circular Queue), LinkedLists, Priority Queue

-FIFO (aka LILO)

-First In First Out (line cheese)

-Applications

-Threads, Job Scheduling, AI task scheduling

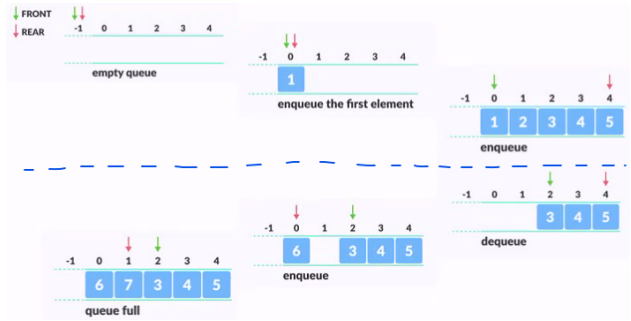
-Array Implementation

-One way is adding from the rear and removing from the front. So you need to keep track of both indexes.

```
QueueArray:
        int static final SIZE = 10
        T[] queue = new T[SIZE]
        int rear = -1 //-1 means empty
        int front = -1
        int count = 0

        boolean isFull() //Overflow check
                //(Is this right?)
                return rear == SIZE
        boolean isEmpty() //Underflow check
                return (count == 0)

        //Operations
        void enqueue(T theElem) //O(1)
                //Overflow check -> expand capacity, throw except, etc.
                rear++
                queue[rear] = theElem
                count++
        T dequeue() //O(1)
                //Underflow check -> throw exception
                front++
                T result = queue[front] //Save for return
                queue[front] = null //Garbage collection set up
                count--
                return result
```

-Circular Queue with Array

-Instead of running out of space when we reach the SIZE and needing to shift elements, have the index loop.




```
CircularQueueArray extends QueueArray:
        @Override
        boolean isFull() //Overflow check
                return count == Size
        @Override
        void enqueue(T theElem) //O(1)
                //Overflow check -> expand capacity, throw except, etc.
                rear = (rear+1) % SIZE //Mod loops the index
                queue[rear] = theElem
                count++
        @Override
        T dequeue() //O(1)
                //Underflow check -> throw exception
                front = (front+1) % SIZE //Mod loops the index
                T result = queue[front] //Save for return
                queue[front] = null //Garbage collection set up
                count--
                return result
```

-LinkedList implementation

-Dynamic size

-Dequeue front, enqueue back. Else, dequeueing back needs to access the 2nd to last Node to link, making it O(n)

```
QueueSLL:
        Node front = null
        Node rear = null
        int count = 0

        boolean isEmpty() //Underflow check
                return count == 0

        void enqueue(T theElem) //O(1)
                Node temp = new Node(theElem)
                //Case: Empty
                if (isEmpty)
                        front = rear = temp
                //Case: Normal not empty
                else :
                        rear.next = temp
                        rear = temp //aka rear = rear.next
                count++
        T dequeue() //O(1)
                //Check underflow - > throw exception
                T result = front.elem
                front = front.next
                if (count == 1) //Case: count == 1, rear is pointing to deleted Node for without this.
                        rear = null
                count--
                return result
```

-Circular Queue with SLL

-If we have a Circular SLL

-enqueue() & dequeue() would need rear.next update

-enqueue()

```
...
rear.next = temp
rear = temp
temp.next = front //wow append
...
```

-dequeue()

```
...
front = front.next
rear.next = front //wow append
…
```

-Priority Queue

-Like a queue, but enqueue() adds to the order of priority

-So with SLL, a Node with the highest priority will be added to the `rear` (“front of the queue”) of the queue.

-A Node with the lowest priority is added to the `front` (“last in the queue”)

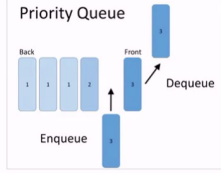
-The Node can have an `int priority`.

-This means O(n) because we need to iterate through and insert.

Priority Queues

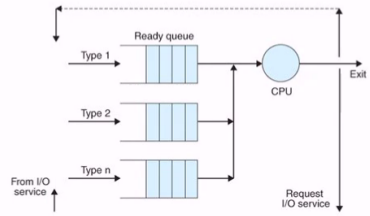
T elem
int priority

- A **priority queue** is a collection in which items in priority queue are **ordered** by **some key:**
  - Item with the lowest key / highest key is always at the front from where they are removed.
  - Items with the same priority are ordered in accordance with the first in, first out principle.
- It supports two major operations:
  - `insert()` and
  - `removeMin()/removeMax()`



Many, many applications.

- Scheduling queues for a processor, print queues, transmit queues, backlogs, etc.....




```
PQueue-SLL:
        Node:
                T elem
                int priority
                Node next = null

                public Node (T theElem, int thePriority)
                        //blah blah

                Node front = rear = null
                int count = 0

        //The only new thing (Overrides enqueue)
        void insert(T theElem, int thePriority)
                temp = new Node (theElem, thePriority)
                if (isEmpty()) //empty case
                        front = rear = temp
                else if (temp.priority >= rear.priority): //1 Node case
                        rear.next = temp
                        rear = temp
                else: //iterate through the list case
                        Node curr = front
                        if (temp.priority < front.priority) //less priority in front
                                temp.next = front
                                front = temp
                        else:
                                while (curr.next.priority <= temp.priority) //wont reach rear, because we checked already
                                        curr = curr.next
                                temp.next = curr.next
                                curr.next = temp
                count++
```

-Fat time complexity of Stack and Queue

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
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)

-Exercise

-Implement queue with 2 stacks

-Always add to Stack A,

-if B is empty, pop() all until B.

-else pop() from B

```
UglyQueue: //Space: O(n) with additional space for both stacks compared to SLL
        Stack A,B
        int count = 0
        void enqueue(T theElem) //O(1)
                A.push(theElem)
                count++
        T dequeue() //O(n)
                if (A.isEmpty() && B.isEmpty())
                        throw Exception
                if (B.isEmpty())
                        while (!A.isEmpty())
                                B.push(A.pop())
                count--
                return B.pop()
```

-Implement Queue with 1 Stack

-Use recursion to hold elems

-Implement Stack with 2 Queues

-pop() always enqueue A

-dequeue() enqueue all of A to B

```
        UglyStack:
                Queue A,B
                int count = 0
        void push(T theElem) //O(1)
                A.eneque(theElem)
                count++
        T pop() //O(n)
                if (A.isEmpty() && B.isEmpty())
                        throw Exception
                //transfer from A to B except last
                if (B.isEmpty()
```

```
                while(A.size()! = 1)
                        B.enqueue(A.dequeue)
        T result = A dequeue()
        //Switch A&B
        Queue temp = A
        A = B
        B = temp
        count--
        return result
```

-Not Leetcode

-Print reverse order of queue

```
void reverse(Queue q):
        Queue temp = new Queue()
        while (q.size > 0 || temp.size > 0)
                //pop() s, fillup temp until last
                while(q.size > 1)
                        temp.enqueue(q.dequeue())
                end
                //print last of q
                System.out.print(q.dequeue())
                //switch s & temp
                Queue switch = temp
                temp = q
                q = switch
        end
end
void reverse(Queue q) //O_t(n^2) & O_s(1)
        for i -> 0 to q.size() - 1
                count = q.size()
                while(count != 1)
                        q.enqueue(q.dequeue())
                        count--
                print(q.peek())
```

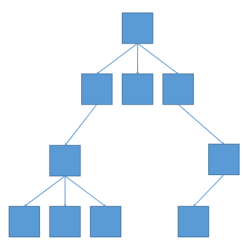
Trees

-Wat the tree?

-Nonlinear, stored into a hierarchy tree cheese

A *tree* is a nonlinear structure in which elements are organized into a hierarchy.

- A tree is composed of a set of *nodes* in which elements are stored and *edges* that connect one node to another.
- Each *node* is at a particular *level* in the tree hierarchy.
- The *root* of the tree is the only *node* at the top level of the tree.
- The nodes at lower levels of the tree are the *children* of nodes at the previous level.
- Nodes that have the same parent are called *siblings*.
- *A node that does not have any children is called a leaf.*
- *A node that is not the root and has at least one child is called an internal node.*



- Our trees *"grow"* from the root at the top of the tree to the leaves toward the bottom of the tree. The root is the entry point into a tree structure.
- We can follow a *path* through the tree from parent to child. This *path length* is determined by counting the number of edges that must be followed to get from the root to the child node.
- The *depth* of a node is the length of the path from the root to the node.
  - This is measured as number of edges (0 based) or number of nodes (1 based) – some disagreement.
  - The *depth of the tree* is defined as the maximum depth of any node (or any leaf).
- The *height* of a node is the length of the path from the node to the deepest leaf in the subtree.
  - This is measured as number of edges (0 based) or number of nodes (1 based) – some disagreement.
  - The *height of the tree* is defined as the maximum height of any node (or any leaf).

-Root, branches to Internal Nodes, ends with Leaves Nodes

-A path takes you to the desired node. Path Length: edges needed to travel to desired node, Depth: lowest leave node, Height: Max height of any node.

-No convention to start at 0 or 1 index, and count Nodes or edges. Just be consistent.

-Height (# of edges) determines time complexity

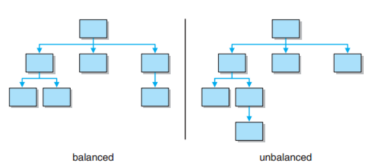
-Classification

-Order of the tree

-General: No limit to children

-N-ary: N children max

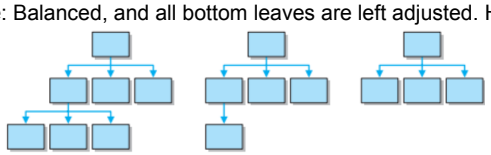
-Balance



-Balanced: the difference between the depth of leave nodes is 0 or within 1 level of each other (O(logn))

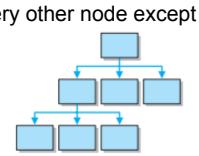
-Unbalanced: leaves not on same level (not O(logn), harder to sort and stuff)

-Complete: Balanced, and all bottom leaves are left adjusted. Has 2^k (same amount of) nodes at every level except last



(All trees are a Complete Balance tree)

-Full: Every other node except leaves have n (same amount) of children



Binary Tree

-At most two children, left and right of the parent

-2 common types

-Heap: maintain an ordering between parents and children

-Application: Heap sort, priority queue

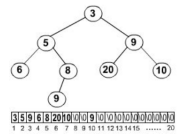
-Binary Search Tree (BST)

-Maintains ordering between all nodes for fast search

-Array implementation

-Patter of indexes denote root, left, and right

- One way of implementing a Binary Tree using Arrays is by indexing the positions of the nodes within the tree:
  - Root node will always be at index location = 0
  - left child of node p will be stored at index location = 2 * index(p) + 1
  - right child of node p will be stored at index location = 2 * index(p) + 2



a

-It's easy to find the parent, but it wastes 2 indexes when you have a leaf node

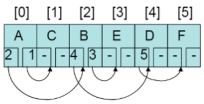
-We ignore 0, unused because calculating

-Tree to Array representation is traversing level order but includes null.

-Simulated Array Links implementation

-Each element is an object with 2 Nodes that store the data and int index that points to its child.

Simulated Link Strategy in an Array (added in the order A, C, B, E, D, F)

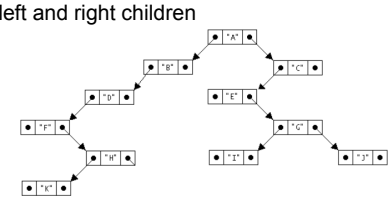


- In a binary tree, each element of the array is an object with a reference to a data element and an `int` index for each of its two children.
- A new child is always added to the end of the contiguous storage area in the array to avoid wasting space.
- However, there is an increased overhead to remove element from the array (to shift the remaining elements and alter the index values as required)

-Doesn't waste space like using Arrays, but it takes longer to delete elements because we shift.

-Linked List implementation

-2 links, left and right children



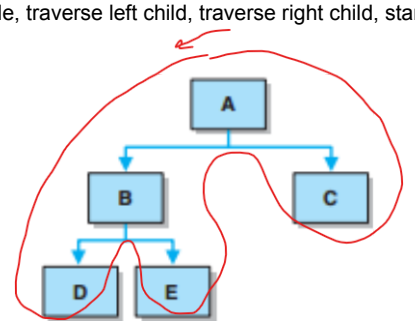
-Ez, no waste of space.

```
class Node:
        T elem
        Node left = null
        Node right = null
class BinaryTree:
        Node front = null
        int count = 0
        boolean isEmpty():
                return root == null
```

-Tree Traversal

-Preorder Traversal

-Visit node, traverse left child, traverse right child, staring at root



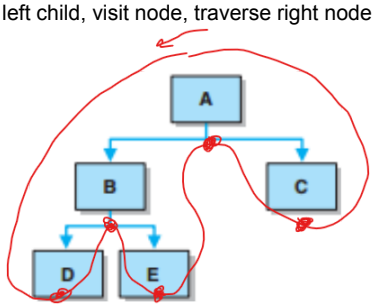
-A, B, D, E, C (Boat visits the nodes, prints when it reaches one or on left of node)

-Root always first

```
preorder(Node curr) //O(n)
        if (curr != null)
                print(curr.elem)
                preorder(curr.left)
                preorder(curr.right)
```

-Inorder Traversal

-Traverse left child, visit node, traverse right node



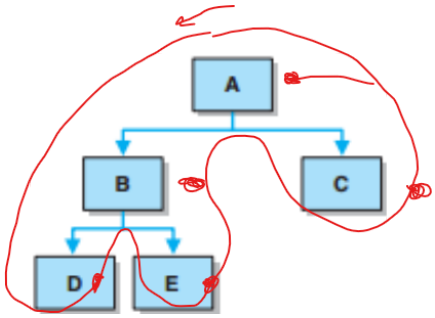
-D, B, E, A, C (Boat prints when under node)

```
inorder(Node curr) //O(n)
        if (curr != null)
                inorder(curr.left)
                print(curr.elem)
                inorder(curr.right)
```

-Binary Search Tree (smaller elem always on left child) uses inorder traversal to print elements in sorted order

-Postorder Traversal

-Traverse left child, Traverse right child, Visit node



-D, E, B, C, A (Boat prints when right of node)

-Root always last

```
postorder(Node curr) //O(n)
        if (curr != null)
                postorder(curr.left)
                postorder(curr.right)
                print(curr.elem)
```

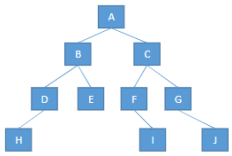
-All of the above will work for Array implementation too, just pass indexes

-Level Order Traversal

-Left to right, from the top. Actually reading it

```
levelorder(Node curr):
        Queue q //O(n)
        q.enqueue(curr) //Add A
        while(!q.isEmpty())
                curr = q.dequeue()
                print(q.dequeue()) //Prints A
                q.enqueue(curr.left) //Add B
                q.enqueue(curr.right) //Add C. Then repeat for B, then again for C, then D & E.
```

-Myth Prac Exercise: 3 traversals



-Preorder (boat immediate/left): ABDHECFIGJ

-Inorder (boat under): HDBEAFICGJ

-Postorder (boat right): HDEBIFJGCA

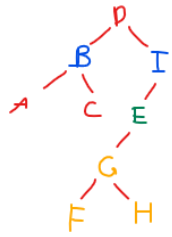
-Myth Prac Exercise: recreate tree from traversals

Consider the following traversals on a (same) tree:

preorder: D, B, A, C, I, E, G, F, H

inorder: A, B, C, D, F, G, H, E, I

What will be the postorder traversal of the above tree?



-Preorder & Inorder tree reconstruction [Reconstructing a Tree From Its Depth-First Traversals | Baeldung on Computer Science]

-Left to right preorder

-1st is the root

-This means you can split inorder at the root node, whatever is to the left is left child, same for right.

-Keep going left to right for next preorder.

-Split it again by that next node in preorder.

-Left or Right of that, extend the child accordingly

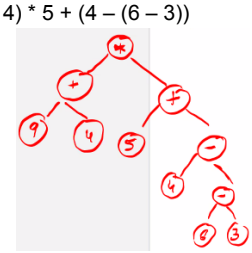
-If there is available opening for a child at a higher depth, add there instead.

-Pre/In/Postfix Equations

-Operators: Root/Internal Node & Operands: Leafs

-Traversal: Infix = Inorder, Prefix = Preorder, Postfix = Postorder (meaning tree is always the same)

-Ex: (9 + 4) * 5 + (4 – (6 – 3))



(start from bottom, then work up)

Binary Heap

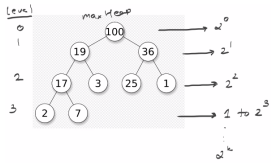
-A type of a Binary Tree

-A Complete tree (so Balanced)

-When you add/remove, you must ensure it's Complete

-That means every level except the last is filled with maximum children

-With 0 as root, each level has 2^(level) Nodes. So the level = log_2(amount of Nodes)



$2^k = n$ where $n = \#$ of nodes

ht: $n = 2^k$

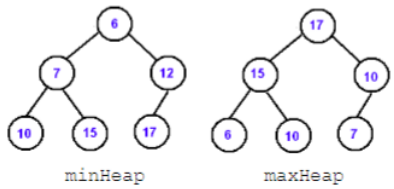
$\therefore k = \log_2 n$

k is max depth/level

-2 types of Heaps

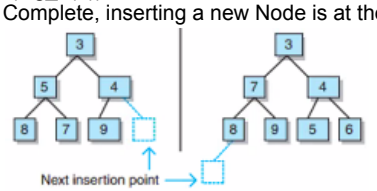
-minHeap: For each Node, the parent Node is <= to both left and right child (min elem at root)

-maxHeap: The parent Node is always > than it's children. (max elem at root)



-Operation: insert() O(log_2(n))

-Since it's Complete, inserting a new Node is at the height (highest level)



Because a heap is a complete tree, there is only one correct location for the insertion of a new node:

- Either the next open position from the left at level h.

OR

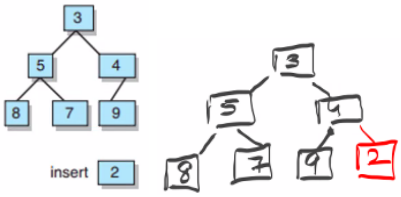
- The first position on the left at level h + 1 if level h is full.

-Then, you have to reorder (heapify) the element to keep it min/max

Reordering: compare the new value to its parent value and swap the values if the new node is less than its parent. Continue this process up the tree until the new value either is greater than its parent or is in the root of the heap.

-Ex: (minHeap)

-Insert



-Heapify/Swap to correct ordering

-Since the height indicates the max swap, that's the worst case.

-Operation: removeMin() O(log_2(n))

-For minHeap, removes the root.

Removes the minimum element from the `minHeap` and returns it:

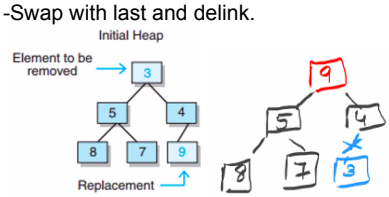
- Return the element stored at the root and replace it with another element in the heap.
- To maintain the completeness of the tree, there is only one valid element to replace the root, and that is the element stored in the last leaf in the tree (last leaf will be the rightmost leaf at level h of the tree).
- Once the element stored in the last leaf has been moved to the root, the heap will then have to be reordered to maintain the heap's ordering property.

-The root is replaced with the most far-right Node.

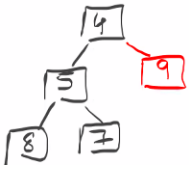
-Then heapify a cuh.

-Ex:

-Swap with last and delink.



-Then heapify



-Linked List Heap Implementation

-Add and Remove is slow because of traversal, reordering, needing references to last node, cancer.

- Using Linked Lists
  - `addElement()`: This method must accomplish three tasks: add the new node at the appropriate location, reorder the heap to maintain the ordering property, and then reset the `lastNode` pointer to point to the new last node.
    - The first step is adding an element(in the worst case, this involves traversing from the bottom-right node of the heap up to the root and then down to the bottom-left node of the heap) = 2*log n.
    - The next step is to insert the new node (involves only simple assignment statements)= (O(1)).
    - The last step is to reorder the path from the inserted leaf to the root, if necessary (heapify) = log n comparisons, because that is the length of the path.

  Thus the `addElement()` operation for the linked implementation has time complexity 2*log n + 1 + log n or O(log n).

  - `removeMin()`: This method must accomplish three tasks: replace the element stored in the root with the element stored in the last node, reorder the heap if necessary, and return the original root element.
    - The first process must remove the root element and replace it with the element from the last node (accomplished with simple assignment statements) = O(1)
    - The next step must reorder the heap, if necessary, from the root down to a leaf (heapify) = log n because the maximum path length from the root to a leaf is log n.
    - Finally, we must determine the new last node (the worst case is that we must traverse from a leaf through the root and down to another leaf) = 2*log n.

  Thus the `removeMin()` operation for the linked implementation has time complexity 2*log n + 1 + log n or O(log n).

-Array Heap Implementation (the better option)

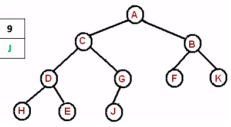
-Super easy to get elements by indices. No waste of space because a Heap is a Complete BT

- Using Arrays:

A complete binary tree can be uniquely represented by storing its level order traversal in an array.

index	0	1	2	3	4	5	6	7	8	9
	A	C	B	D	G	F	K	H	E	J

The left child of parent i is located at (2*i + 1) index
The right child of parent i is located at (2*i + 2) index
The parent of child i is located at (i-1/2) index



-The index can start at 1 so it's easier to calculate indices

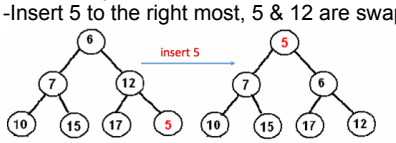
If Root = 1
left → 2*i
Right → 2*i +1
parent: i/2
i → index of child

-Add and Remove also O(logn), but it doesn't have overhead like Linked List to traverse for references.

- `addElement()`: This method must accomplish three tasks: add the new node at the appropriate location, reorder the heap to maintain the ordering property, and increment the count by one. Of course, as with all of our array implementations, the method must first check for available space and expand the capacity of the array if necessary.
  - Unlike the linked implementation, the array implementation does not require the first step of determining the parent of the new node. However, both of the other steps are the same as those for the linked implementation. Thus the time complexity for the `addElement()` operation for the array implementation is 1 + log n or O(log n).

-Ex: add() in minHeap

-Insert 5 to the right most, 5 & 12 are swapped, 5 & 6 are swapped, boom.

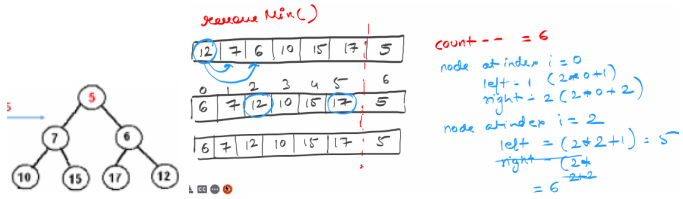


index	0	1	2	3	4	5	6	
	6	7	12	10	15	17	5	parent index of 5 = 6-1/2 = 2
	6	7	5	10	15	17	12	parent index of 5 = 2-1/2 = 0
	5	7	6	10	15	17	12	

- `removeMin()`: This method must accomplish three tasks: replace the element stored in the root with the element stored in the last element, reorder the heap if necessary, and return the original root element.
  - The array implementation of the `removeMin()` operation looks just like the linked implementation, except that it does not have to determine the new last node. Thus the resulting time complexity is log n + 1 or O(log n).

-Ex: removeMin() in minHeap (same Heap as above)

-Swap 5 (min) with 12 (last), heapify by comparing 12 with subsequent left and right children, return 5 (set null if Object).



-Heap visualization

[Heap Visualization (usfca.edu)]

-Fat Heap time complexity

Build-Heap	FindMax	ExtractMax	Increase Key	Insert	Delete	Merge
O(n)	O(1)	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(m+n)

-Build-heap is not heapify. Build-heap is checking everything, reordering everything.

-Build-heap O(n)

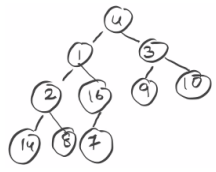
-O(n) is because you already know the all the elements beforehand.

-Building a heap normally by inserting one by one wastes time heapifying every insert.

-Ex: minHeap from a given array of unordered ints

1. represent array as heap (level order traversal)

[4,1,3,2,16,9,10,14,8,7]

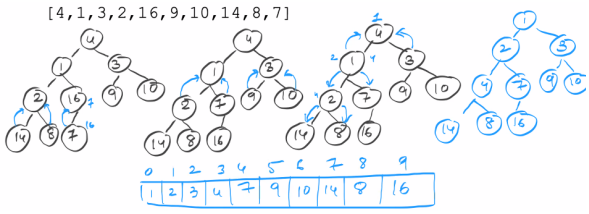


2. We need to heapify EVERYTHING. Both left and right child is compared to parent for swap.

-Start at bottommost level, compare each child to parent, then go to next level.

-You must also compare downwards when you are not at bottommost level

[4,1,3,2,16,9,10,14,8,7]

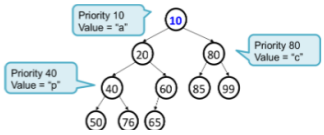


-We didn't sort the array btw, just put min on top, and larger is downwards.

-Priority Queue using Heap

-Instead of inserting based on element comparison, it's based on a priority value. minHeap or maxHeap, it depends.

-Ex: minHeap



-Since inserting to a heap is O(logn), it's faster than O(n) of normal queue, with the downside of elements not being 100% ordered.

-Because of heapify, elements with the same priority are not guaranteed FIFO.

	insert	removeMin	findMin
Ordered array	O(n)	O(1)	O(1)
Ordered list	O(n)	O(1)	O(1)
Binary heap	O(log n)	O(log n)	O(1)

```
class Node:
        T elem = null
        int priority
class ArrayHeap:
        int final size = 10
        T[] heap = new T[SIZE]
        int count = 0
        void addElement(T obj) //O(logn)
                //check capacity
                heap[count] = obj
                count+++
                //heapify
        T removeMin() //O(logn)
                //check underflow
                //swap root with count-1
                count--
                //heapify
                //return
class PQ extends ArrayHeap:
        add(T obj, int prior)
                Node temp = new Node(elem, prior)
                super.addElement(temp)
        T remove()
                return super.removeMin()
```

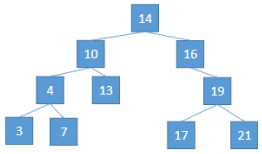
-Heap Summary

- **heap** A binary tree that is complete and is either a minheap or a maxheap.
- **minheap** A binary tree with two added properties: It is a complete tree, and for each node, the node is less than or equal to both the left child and the right child.
- **maxheap** A binary tree with two added properties: It is a complete tree, and for each node, the node is greater than or equal to both the left child and the right child.
- **complete binary tree** A balanced binary tree in which all of the leaves at level **h** (the lowest level of the tree) are on the left side of the tree.
- **priority queue** A collection that follows two ordering rules: Items with higher priority go first, and items with the same priority are ordered in accordance with the first in, first out principle.

Binary Search Tree (BST)

-The difference between this and a normal tree is because of the invariant

-Elements on the left child is smaller in value than its parent. Right, and it's bigger



-Best implementation is LL because Array wastes space since BST isn't complete

-Benefits

-Inorder traversal will result in displaying the elements lowest to greatest

-Searching BST is like binary search. The cost is equal to the depth/height.

-Since it's not a heap, you need to back track when reaching dead ends.

-Adding elements

-Move left and right depending on comparison with parent and argument, adding to next available spot

-Removing elements

-Find the item

-Removing

-If a leaf, we can just remove it

-If it has a child, remove it and promote child to the new empty position

-If it has 2 children

-Remove and Swap one of the children to its place.

-Replacement must return inorder successor (because we treat = to the right subtree) of the node to be removed (because equal elements are placed to the right).

-There are multiple solutions (like using inorder predecessor), the better one keeps the height as low as possible.

-Fat SLL vs BST Time Complexity

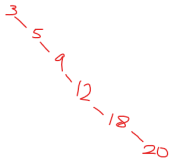
removeFirst	O(1)	O(log n)
removeLast	O(n)	O(log n)
remove	O(n)	O(log n)*
first	O(1)	O(log n)
last	O(n)	O(log n)
contains	O(n)	O(log n)
isEmpty	O(1)	O(1)
size	O(1)	O(1)
add	O(n)	O(log n)*

*both the add and remove operations may cause the tree to become unbalanced

-Ex:

Let's assume that we read the following list of integers from a file and added them to a binary search tree:

3 5 9 12 18 20



-Height = 6 = n, therefore O(n). This is worst case scenario for BST. We need to balance it.

-Depth of Binary Tree

- A binary tree is balanced if the left and right sub-trees of every node differ in depth by at most 1.
  - In balanced trees the leafs tend to be on about the same level.
  - In balanced binary trees the depth of the tree is $O(\log n)$ in the worst case.
  - In unbalanced binary trees the depth is $O(n)$ in the worst case.
- Thus our operations are only efficient if the we can ensure the binary search tree is balanced.

-If not balanced, it can be worse than linear structures.

-Maintaining Balance

-Imbalance happens when you add & remove

-So, after adding/removing, we need to rebalance if needed

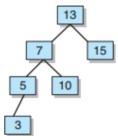
-The way to balance is "rotation". Left and Right Rotation

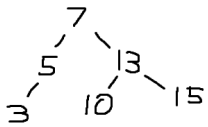
-When you detect unbalance, you should apply the rotation.

-The should method takes a root and will perform the rotation with all of the children.

-So you can pass a subset of a BST

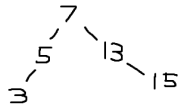
-Right Rotation





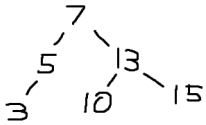
- Make the left child element of the root the new root element.

- Make the former root element the right child element of the new root.

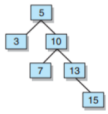


- Make the right child of what was the left child of the former root the new left child of the former root.

(This doesn't work all the time, mistake? You need to normal insert the one that got cut off.)

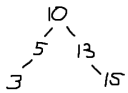


-Left Rotation

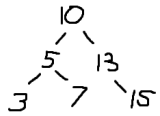


- Make the right child element of the root the new root element.

- Make the former root element the left child element of the new root.



- Make the left child of what was the right child of the former root the new right child of the former root.

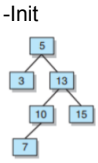


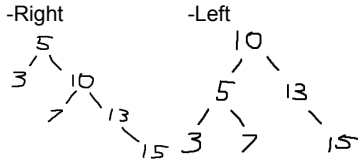
-Right-Left Rotation

- **Right-Left Rotation**

  first perform a right rotation of the left child of the right child of the root around the right child of the root, and then perform a left rotation of the resulting right child of the root around the root.

-Init -Right -Left



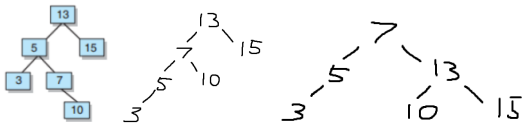


-Left-Right Rotation

- **Left-Right Rotation**

  first perform a left rotation of the right child of the left child of the root around the left child of the root, and then perform a right rotation of the resulting left child of the root around the root.

-Init -Left -Right



-Fat BST Time Complexity

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
BST	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

-The worst case comes from BST not rotating.

-You can't know to rotate if you can't detect imbalance.

AVL Tree (Self-Balancing BST)

-Self-Balancing BST by Adel'son-Vel'skii and Landis.

-Keep track of the height of left and right subtrees

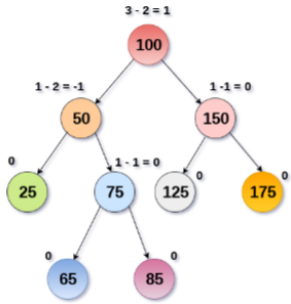
-Balance Factor (value indicating if tree is balanced) = height of left subtree - right (or other way)

-Each Node has a Balance Factor.

-If > 1 (imbalance on left) or < -1 (imbalance on right), the Node and its children need to be balanced.

-During add, remove, & rotation, we can recursively recalculate the balance factor of each.

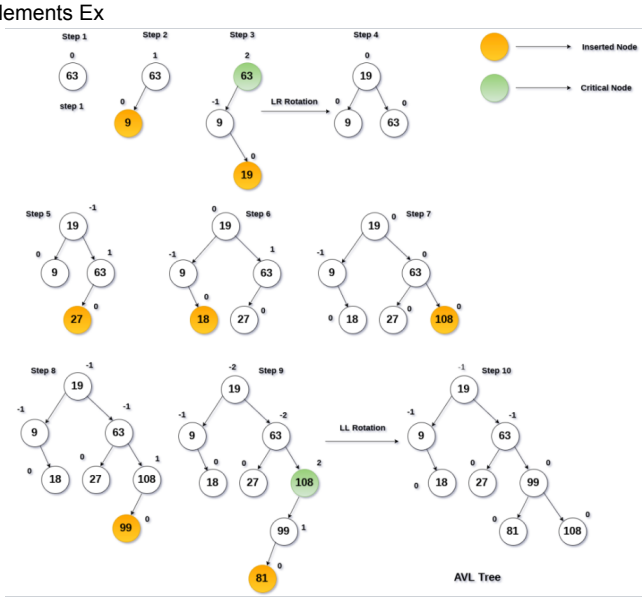
-You have to do multiple rotations if child node have the opposite side of imbalance to the parent.

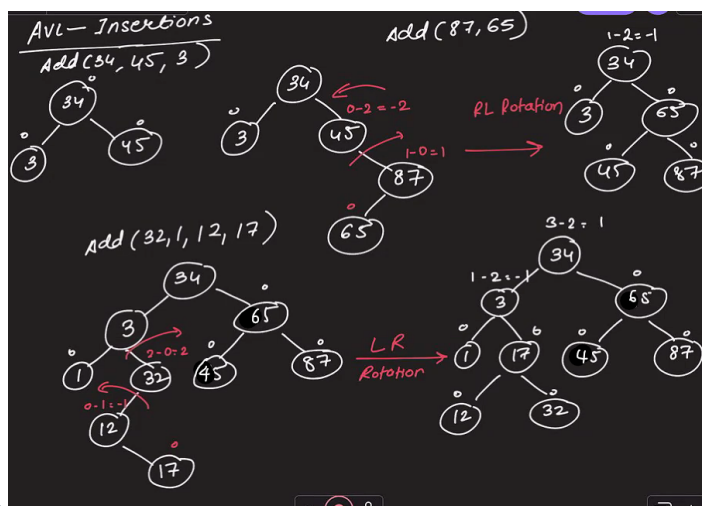


- AVL trees are mostly used for in-memory sorts of sets and dictionaries.
- AVL trees are also used extensively in database applications in which insertions and deletions are fewer but there are frequent lookups for data required.
- It is used in applications that require improved searching apart from the database applications.

```
class TreeNode:
        T elem
        int blanceFactor = 0
        TreeNode left = null
        TreeNode right null
```

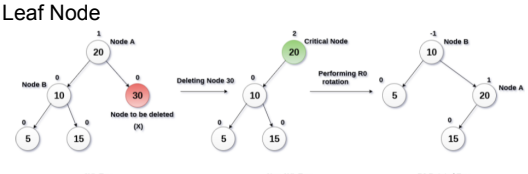
-Adding Elements Ex





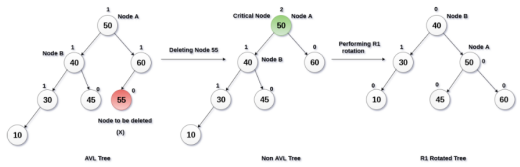
-Deleting Element Ex

-Deleting Leaf Node

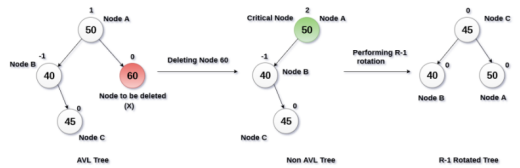


- In this case, the node B has balance factor 0, therefore the tree will be rotated by using Right rotation as shown in the following image. The node B(10) becomes the root, while the node A is moved to its right. The right child of node B will now become the left child of node A

-Deleting Leaf Node (Double Rotation)

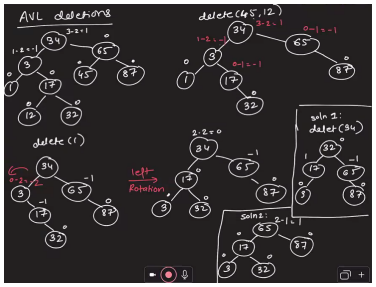


- Deleting 55 from the AVL Tree disturbs the balance factor of the node 50 i.e. **node A** which becomes the critical node. This is the condition of Right rotation in which, the node A will be moved to its right (shown in the image below). The right of B is now become the left of A (i.e. 45).



- In this case, node B has balance factor -1. Deleting the node 60, disturbs the balance factor of the node 50 therefore, it needs to be L-R rotated. The node C i.e. 45 becomes the root of the tree with the node B(40) and A(50) as its left and right child.

-Ex Another



-Time Complexity of Rebalancing

- Recall that rebalancing is only necessary along the path of nodes from the altered node to the root.
  - This means at most the depth of the tree's worth of nodes need to be checked for imbalance.
- Rotate-left and rotate-right require a constant number of node assignments.
  - These changes only affect the local structure of the tree.
- When we encounter a node that needs rebalancing we have to do at most two rotations.
- The total cost of an add or remove remains at the depth of the node accessed.

Red Black Trees (Self-Balancing BST)

-Leonidas J Guibas and Robert Sedgewick in 1978 paper described a self balancing tree where each Node has a color (boolean).

-1. The root is black.
-2. All children of red Node are black.
-3. Every path from the root to a leaf contains the same number of black Nodes. (Most important, acts like AVL Balance Factor)
-4. null Nodes count as black!

- If a Red Black Tree is complete, with all Black nodes except for Red leaves at the lowest level the height will be minimal, ~log N
- To get the max height for N elements there should be as many Red nodes as possible down one path and all other nodes are Black
  - This means the max height would be < 2 * log N

    -More efficient to find places to rotate than AVL

- New Node always a leaf
  - can't be black or we will violate rule
  - therefore the new leaf must be red
  - If parent is black, done (trivial case) – Case 0
  - if parent red, things get interesting because a red leaf with a red parent violates rule

    -If the parent is red, you first try recoloring. If that doesn't work, you need rotation and recoloring.

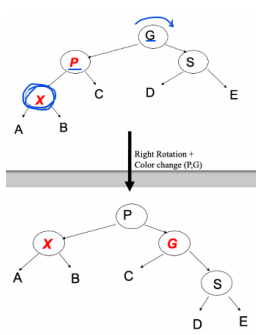
- Real-world uses of red-black trees include *TreeSet*, *TreeMap*, and *Hashmap* in the Java Collections Library.
- Also, the Completely Fair Scheduler in the Linux kernel uses this data structure.
  - Linux also uses red-black trees in the *mmap* and *munmap* operations for file/memory mapping.
- Furthermore, red-black trees are used for geometric range searches, k-means clustering, and text-mining.

Note: Red-black trees are mainly used under-the-hood, and we, as developers, don't come into contact with them that often, even though we use them on a daily basis.

-Insertion Ex (2 Cases)

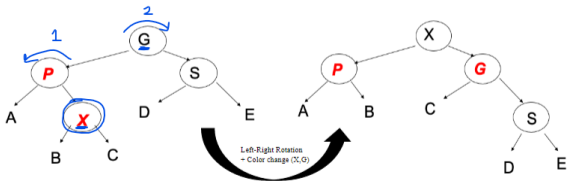
Case 1: Suppose sibling of (Red) parent node is a Black node.

- Note: by convention null nodes are black
- Action: Rotation + Color Change
- Let,
  - X be the new leaf Node,
  - P be its Red Parent node,
  - S the **Black** sibling node and
  - G is P's and S's parent and X's grandparent
- Example 1: Single Rotation

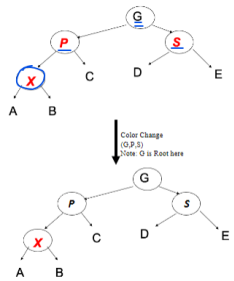


(Letter without circles are null)

- Example 2: double rotation



Case 2: Suppose sibling of (Red) parent node is a Red node.



- Action: Color Change
- Let,
  - X be the new leaf Node,
  - P be its Red Parent node,
  - S the **Black** sibling node and
  - G is P's and S's parent and X's grandparent

-Insertion (1 to 11)

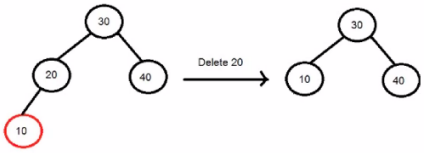
[6: Binary Search Trees (BST): TCSS 342 A Wi 22: Data Structures (uw.edu)](https://uw.edu)

-Deletion (5 Cases)

-Do BST Delete, then deal with cancer black red crap

-Red node deletion is easy, but deleting black is uh oh stinky

- Step 1: Perform [standard BST delete]
- Must maintain rules of Red Black Tree.
  - If node deleted is Red node, done (trivial case) – Case 0



  - If node deleted is a **black** node (violation)
    - # of black nodes across each path need to be the same

Case 1: Black leaf node deleted with Black / null node as sibling (with no children)

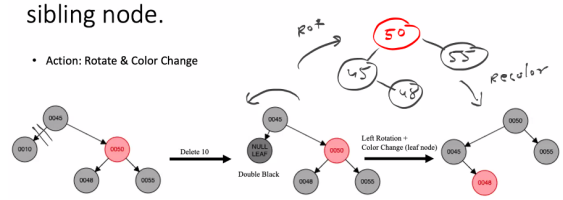
- Note: by convention null nodes are black
- Action: Color Change

Case 2: Black leaf node deleted with Black node as sibling with Red child node.

- Action: Rotate & Color Change

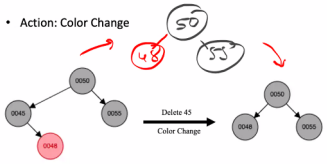
Case 3: Black leaf node deleted with Red sibling node.

- Action: Rotate & Color Change



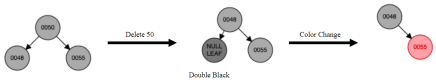
Case 4: Black internal node deleted with Red child node.

- Action: Color Change



Case 5: **Black** internal node deleted with **Black** child nodes

- Action: Color Change



-Fat Time&Space Complexity of AVL and Red Black Trees

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
AVL Trees	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Trees	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

-AVL does more rotations than Red Black Trees, making Red Black more preferred.

-There is another self-balacing BST not covered for Splay Tree.

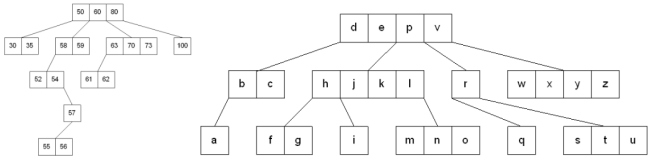
-The new element becomes the root everytime you add.

-Used for frequent lookup to. The children of the root is the most similar/near in value (and the root is always most recent element added).

-Used by internet router to sort IP and whatnot

Multiway

-Can have more than 2 children, and can store than 1 element. Like BST, left < root, and right > root.



-More branches reduce the height, it lowers time to access and write, useful when data requires disk access (ie virtual memory page cheese).

-O(logn_m(n)) where m is the max degree of the tree.

-Rules for max efficiency

-A node can have *degree* children

-A node can have *degree - 1* elements
-Leaf Nodes are on the same level
-Children Nodes and Elems in a Node must be order from left to right as lowest to highest.

-Insertion (quick guide)
-Add to a Leaf Node. If there are too many children, borrow & swap elements from the parent and neighbor in order to add new in place.

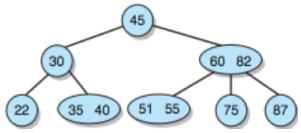
-Deletion (quick guide)
-Just remove it. If it causes issues with the rules, borrow & swap elements.

-"2-3" Tree (3rd Degree/Order)
-A node can have 0, 2, 3 children.
-It can have 2 elements, 1 is designated as the smaller elem, while the other is the bigger one.
-If a Node have 3 children, the leftmost is the Node with the least elem & the right is the biggest.
-Leaves are on the same level

A 2-3 tree is a multi-way search tree in which each node has two children (referred to as a 2-node) or three children (referred to as a 3-node):

- A 2-node contains one element and, like a binary search tree, the left subtree contains elements that are less than that element and the right subtree contains elements that are greater than or equal to that element.
  Note: a 2-node can have either no children or two children—it cannot have just one child.
- A 3-node contains two elements, one designated as the smaller element and one designated as the larger element.
  Note: a 3-node has either no children or three children.
- If a 3-node has children, the left subtree contains elements that are less than the smaller element and the right subtree contains elements that are greater than or equal to the larger element.
- The middle subtree contains elements that are greater than or equal to the smaller element and less than the larger element.

Note: All of the leaves of a 2-3 tree are on the same level.



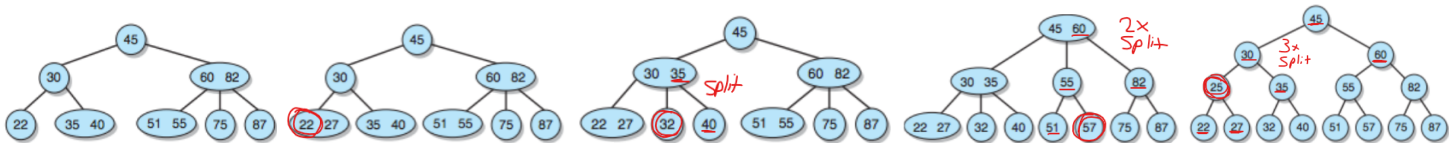
-Inserting

All insertions into a 2-3 tree occur at the leaves of the tree:

- Case 1: The tree is empty, a new node is created containing the new element, and this node is designated as the root of the tree. Duh
- Case 2: To insert a new element at a leaf that is a 2-node. That is, we traverse the tree to the appropriate leaf (which may also be the root) and find that the leaf is a 2-node (containing only one element). In this case, the new element is added to the 2-node, making it a 3-node.
  Note: The new element may be less than or greater than the existing element.
- Case 3: To insert a new element at a leaf that is a 3-node (containing two elements). In this case, because the 3-node cannot hold any more elements, it is split, and the middle element is moved up a level in the tree.

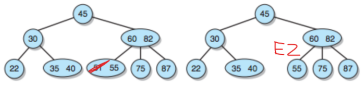
-Case 3 most important! The middle elem moves up and the 2 side elems split to be its children

-Ex: Insert 27, 32, 57, 25.



-Deleting

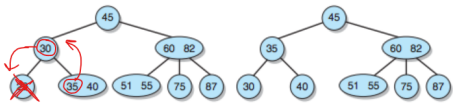
- **Case 1:** The element to be removed is in a leaf that is a 3-node, we simply remove the element from the node.

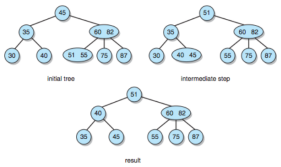


(Remove 51)

- **Case 2:** The element to be removed is in a leaf that is a 2-node. This condition is called *underflow* and creates a situation in which we must rotate the tree and/or reduce the tree's height in order to maintain the properties of the 2-3 tree:
  i. The parent node has a right/left child that is a 3-node, we can maintain the properties of a 2-3 tree by rotating the smaller/larger element of the 3-node around the parent.

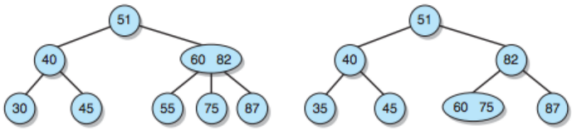
ii. Remove 22





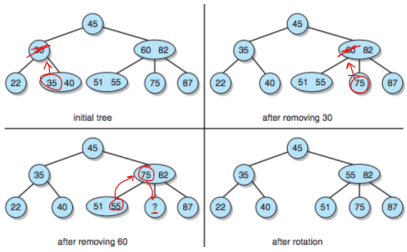
(Remove 30)

-Rotate is more like Borrowing



(Remove 50)

- **Case 3:** The element to be removed is in an internal node, simply replace the element to be removed with its in-order successor.



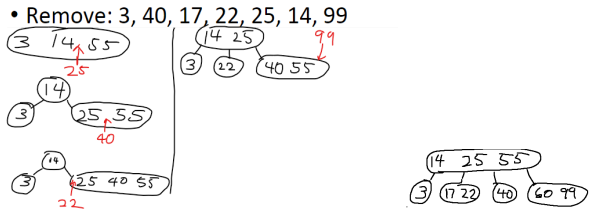
-"2-4" Tree (4th Degree/Order)
-Nodes have 1 to 3 elems. (because 4th degree - 1 = 3)
-Left to right → lowest to higest
-A node can have up to 4 children (because 4th degree)
-Children orders from left to right -> least to greatest
-Leaves are on the same level like 2-3

A 2-4 tree is similar to a 2-3 tree, a 4-node contains three elements and has either no children or four children. The same ordering property applies:

- The left child will be less than the leftmost element of a node, which will be less than or equal to the second child of the node, which will be less than the second element of the node, which will be less than or equal to the third child of the node, which will be less than the third element of the node, which will be less than or equal to the fourth child of the node.
- Insertion and Removal operations are same as the 2-3 Trees.

-Ex: (Unfinised)

- Insert: 3, 55, 14, 25, 40, 22, 99, 60, 17
- Remove: 3, 40, 17, 22, 25, 14, 99



-Multiway Tree Visualization

[B-Tree Visualization (usfca.edu)]

-Multiway with higher degree

-Let max degree = m.

-The max elems in a node is m - 1.

-When you insert and need to split because it is at the max elems, the new node that you split should have (m - 1) / 2 elems, and one gets move to the parent.

B-Tree

-Multiway Tree of order/max degree *M* and max data in leafs *L*

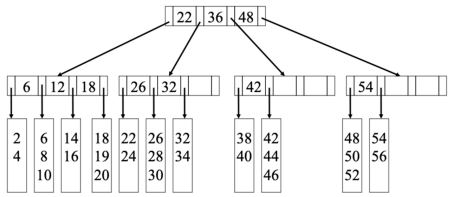
B-trees of order m have the following properties:

- The root has at least two subtrees unless it is a leaf.
- Each non-root internal node n have between [m/2] and m subtrees (i.e. each interior node is at least half-full)
- All leaves are on the same level.
- A leaf must store between [L/2] and L data elements, where L is a fixed constant >= 1 (i.e. each leaf is at least "half full", except when the tree has fewer than L/2 elements)

- When designing a B-Tree (choosing the values of **M** and **L**), we need to consider the size of the data stored in the leaves, the size of the keys and pointers stored in the interior nodes, and the size of a disk block.

-To use it practically, you have to calculate the amount of bytes is needed for the B-Tree to store in memory.

-Ex: m=4, l=3



- The following figure (next slide) shows a B-Tree with M = 4 and L = 3
- The root node can have between 2 and M = 4 subtrees
- Each other interior node can have between [M / 2] = [4 / 2]= 2 and M = 4 subtrees and up to M – 1 = 3 keys.
- Each leaf can hold between [L / 2] = [3 / 2] = 2 and L = 3 data elements.

-Ex: Student Record Storing Finding M & L

- Suppose our B-Tree stores student records which contain name, address, etc. and other data totaling **1024 bytes**.
- Further assume that the key to each student record (student ID??) is **8 bytes** long.
- Assume also that a pointer (really a disk block number, not a memory address) requires **4 bytes**
- And finally, assume that our disk block is **4096 bytes**

- L is the number of data records that can be stored in each leaf. Since we want to do just one disk access per leaf, this is the same as the number of data records per disk block.
- Since a disk block is 4096 and a data record is 1024, we choose

L = [4096 / 1024] = 4 data records per leaf.

Each interior node contains M pointers and M-1 keys. To maximize M (and therefore keep the tree flat and wide) and yet do just one disk access per interior node, we have the following relationship

4M + 8 (M – 1) <= 4096

12M <= 4104

M <= 342

So choose the largest possible M (making tree as shallow as possible) of 342.

- With M = 342 the height of our tree for N students will be $\lceil \log_{342}(N/L) \rceil$
- For example, with N = 100,000 (student population) the height of the tree with M = 342 would be no more than 2, because:

$$\log_{342}(100{,}000 / 4) = \log_{342}(25000) = 2$$

- So any student record can be found in 3 disk accesses. If the root of the B-Tree is stored in memory, then only 2 disk accesses are needed

-1 record = 1024 bytes

-key = 8 bytes

-pointer = 4 bytes

-size = 4096 bytes

-L = disk block size / record = 4096 / 1024 = maximum of 4 data records per leaf

-M = (pointer size + key size <= disk block size) = (4M + 8(M-1) <= 4096)

12M <= 4104

M <= 342 bytes

-Inserting

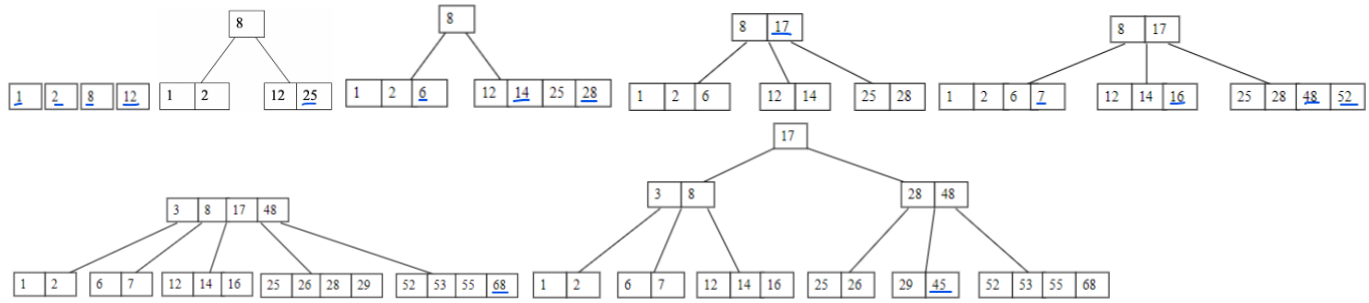
- Attempt to insert the new key into a leaf
- If this would result in that leaf becoming too big, split the leaf into two, promoting the middle key to the leaf's parent
- If this would result in the parent becoming too big, split the parent into two, promoting the middle key
- This strategy might have to be repeated all the way to the top
- If necessary, the root is split in two and the middle key is promoted to a new root, making the tree one level higher

-Same as Multiway

-Insert at leaf Node in order.

-Split and swap elements if too many.

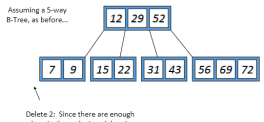
-Ex



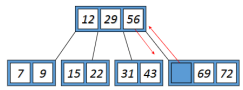
-Delete

- During insertion, the key always goes into a leaf. For deletion we wish to remove from a leaf. There are three possible ways we can do this:
- **Case 1** - If the key is already in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted.
- **Case 2** - If the key is *not* in a leaf then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf -- in this case we delete the key and promote the predecessor or successor key to the non-leaf deleted key's position.

Deletion: Case 1

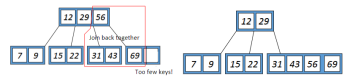


Deletion: Case 2

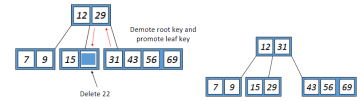


- If (1) or (2) lead to a leaf node containing less than the minimum number of keys then we have to look at the siblings immediately adjacent to the leaf in question:
  - **Case 3**: if one of them has more than the min number of keys then we can promote one of its keys to the parent and take the parent key into our lacking leaf
  - **Case 4**: if neither of them has more than the min' number of keys then the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key) and the new leaf will have the correct number of keys; if this step leaves the parent with too few keys then we repeat the process up to the root itself, if required

Deletion: Case 4



Deletion: Case 3



-Analysis of B-Trees

- The maximum number of items in a B-tree of order m and height h:

root	$m - 1$
level 1	$m(m - 1)$
level 2	$m^2(m - 1)$
. . .	
level h	$m^h(m - 1)$

BST
root = 1 $(2-1)$
level 1 = 2 $2^1(2-1)$
level 2 = 4 $2^2(2-1)$
level 3 = 8 $2^3(2-1)$

- So, the total number of items is

$$(1 + m + m^2 + m^3 + \ldots + m^h)(m - 1) =$$
$$[(m^{h+1} - 1)/ (m - 1)] (m - 1) = \mathbf{m^{h+1} - 1}$$

- When $m = 5$ and $h = 2$ this gives $5^3 - 1 = 124$

- Store data only at leaves; all leaves at same level
  - interior and exterior nodes have different structure
  - interior nodes store one key and two subtree pointers
  - all search paths have same length: [log n] (assuming one element per leaf)
  - can store multiple data elements in a leaf

-Why use B-Trees?

- When searching tables held on disc, the cost of each disc transfer is high but doesn't depend much on the amount of data transferred, especially if consecutive items are transferred
  - If we use a B-tree of order 101, say, we can transfer each node in one disc read operation
  - A B-tree of order 101 and height 3 can hold $101^4 - 1$ items (approximately 100 million) and any item can be accessed with 3 disc reads (assuming we hold the root in memory)
- If we take $m = 3$, we get a **2-3 tree**, in which non-leaf nodes have two or three children (i.e., one or two keys)
  - B-Trees are always balanced (since the leaves are all at the same level), so 2-3 trees make a good type of balanced tree

-Get as much data out of a disk read as possible before processing.

-Time Complexity of B-Tree

Time Complexity								Space Complexity
Average				Worst				Worst
Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

-Another Ex of Finding M & L

- Using a B-Tree, we want to store and process information concerning the driving records for citizens in the Washington state. We assume the following :
  - we have 300,000,000 items,
  - each key is 8 bytes (ID number), (M-1) keys
  - a record is 256 bytes, 1 record in leaf = 256 bytes
  - one disk block holds 8,192 bytes,
  - and a pointer to each block is of size 4 bytes. M ptrs

-1. L = disk block size / record = (8192 / 256) bytes = **32 bytes**
-2. M = (pointer size + key size <= disk block size) = (4M + 8(M-1) <= 8192) bytes = 683.33 bytes → **684 bytes**
-3. Height = log_M(amount of items / L) = log_684(300000000 / 32) = 2.45 → **3**
-Extra: If we use BST, then Height = log_2(300000000 / 32) = 28, which requires way more disk access.

Concept of Hashing

-Previously on Data Structures

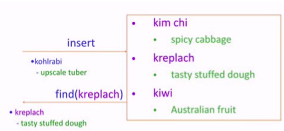
	unsorted list	sorted array	Trees BST – average AVL – worst case splay – amortized	Array of size n where keys are 0,…,n-1
insert	find+θ(1)	θ(n)	θ(log n)	O(1)
find	θ(n)	θ(log n)	θ(log n)	O(1)
delete	find+θ(1)	θ(n)	θ(log n)	O(1) How?

-We can sacrifice some space (a big enough has table to avoid collision) for this.
-Handling collision add some time. If we keep it less than O(logn), ez win.

-Dictionary / Map

Dictionary

- Operations
  - create
  - destroy
  - insert
  - find
  - delete



- Dictionary: Stores *values* associated with user-specified *keys*
  - keys may be any (homogenous) comparable type
  - values may be any (homogenous) type
  - implementation: data field is a struct with two parts
- Search ADT: keys = values

-We can implement one to store our hashes

-Hashing with Array

- Use a key (arbitrary string or number) to index directly into an array (hash table) – O(1) time to access records
  - A["kreplach"] = "tasty stuffed dough"
  - Need a *hash function* to convert the key to an integer

	Key	Data
0	kim chi	spicy cabbage
1	kreplach	tasty stuffed dough
2	kiwi	Australian fruit

-Use a key that encodes the index.
-Here it's a String that gets decoded by a hash function into an Integer index.
-Time complexity of hashing (using the hash function) should be O(1).
-However, we waste some space for a big enough hash table / array in order to avoid collision.
-So then, what should the hashing function be?
-Does it depend on other keys? Collision with other keys? Array resize, how to stay in range and rehash?

- What problems can we face while hashing ?
  1. The situation where two elements or keys map to the same location in the table is called a *collision*.
     - A hashing function that maps each element to a unique position in the table is said to be a *perfect hashing function*.
  2. How large the table should be.
     - If the data set is of known size and a perfect hashing function can be used, then we simply make the table the same size as the data set. If a perfect hashing function is not available or practical but the size of the data set is known, a good rule of thumb is to make the table 150 percent the size of the data set.
  3. What if we do not know the size of the data set?
     - In this case, we depend on *dynamic resizing*. Dynamic resizing of a hash table involves creating a new hash table that is larger than, perhaps even twice as large as, the original, inserting all of the elements of the original table into the new table, and then discarding the original table.

-Hashing Applications

- When log(n) is just too big…
  - Symbol tables in interpreters
  - Real-time databases (in core or on disk)
    - air traffic control
    - packet routing
- When associative memory is needed…
  - Dynamic programming
    - cache results of previous computation

      f(x) →if (Find(x)) then Find(x) else f(x)
    - Chess endgames
  - Many text processing applications – e.g. Web

    $Status{$LastURL} = "visited";

-Fat Hashing Time Complexity

Time Complexity								Space Complexity
Average				Worst				Worst
Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
	O(1)	O(1)	O(1)		O(n)	O(n)	O(n)	O(n)

Hash Functions

-Extraction Method

-Use only 1 part of the element's value to compute the index it will be stored at.

- Simple Approach: *Extraction*

Extraction involves using only a part of the element's value or key to compute the location at which to store the element.

Example:

A simple example where we create an array that will hold 26 elements. Wishing to store names in our array, we create a hashing function that equates each name to the position in the array associated with the first letter of the name.

-Division Method

-Take the key and modulo it with something (ie the hash table size) which will keep it in range.

-Modulo with a prime number will provie better distribution.

Creating a hashing function by division simply means we will use the remainder of the key divided by some positive integer p as the index for the given element.

```
Hashcode(key) = Math.abs(key) % p
```

This function will yield a result in the range from 0 to p–1.

Example:

if our key value is 79 and our table size is 43, the division method would result in an index value of 36.

Note:

1. Using a prime number p as the table size and the divisor helps provide a better distribution of keys to locations in the table.
2. The division method is very effective when dealing with an unknown set of key values.

-Ex: hash table size = array size = 11

- Draw the hash table that results from adding the following integers (34 102 87 31 13 17) to a hash table of size 11 using the division method

key	% 11
34	% 11 = 1
102	% 11 = 3
87	% 11 = 10
31	% 11 = 9
13	% 11 = 2
17	% 11 = 6

0	
1	34
2	13
3	102
4	
5	
6	17
7	
8	
9	31
10	

(Here, the key is an int, so ez modulo)

-Folding Method

-Use multiple equal parts of the key to creatively combine them into a int for index.

In the *folding method*, the key is divided into parts that are then combined or folded together to create an index into the table.

This is done by first dividing the key into parts where each of the parts of the key will be the same length as the desired index, except possibly the last one.

1. *Shift – Folding method:* these parts are then added together to create the index.
2. *Boundary-Folding method:* they involve reversing some of the parts of the key before adding.

Note:

Folding may also be a useful method for building a hashing function for a key that is a string.

One approach to this is to divide the string into substrings the same length (in bytes) as the desired index and then combine these strings using an exclusive-or function. This is also a useful way to convert a string to a number so that other methods, such as division, may be applied to strings.

-Ex: The 4 digits are broken into ½ & added together here. We also use Division Method here to keep it in range of the array.

Draw the hash table that results from adding the following integers (1983, 2312, 6543, 2134, 3498, 7654, 1234) to a hash table using shift folding of the first two digits with the last two digits. Use a table size of 13.

key	shift fold	% 13
1983	19 + 83 = 102	% 13 = 11
2312	23 + 12 = 35	% 13 = 9
6543	65 + 43 = 108	% 13 = 4
2134	21 + 34 = 55	% 13 = 3
3498	34 + 98 = 132	% 13 = 2
7654	76 + 54 = 130	% 13 = 0
1234	12 + 34 = 46	% 13 = 7

0	7654
1	3498
2	2134
3	6543
4	
5	
6	1234
7	
8	2312
9	
10	1983
11	
12	

-String: ASCII Method

-Add up ASCII value of all chars

- Add up the ASCII values of all characters of the key.

```
int hash(const string &key, int tableSize)
{
      int hasVal = 0;

      for (int i = 0; i < key.length(); i++)
            hashVal += key[i];
      return hashVal % tableSize;
}
```

- Simple to implement and fast.
- However, if the table size is large, the function does not distribute the keys well.
  - e.g. Table size =10000, key length <= 8, the hash function can assume values only between 0 and 1016

(Anagrams = bruh moment)

-String: Cyclic Shift Code Method

-Add the ASCII values of char, but after every add, the hash is bit shifted.

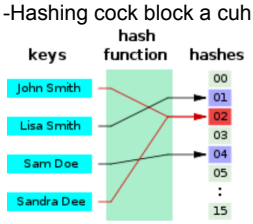
```
int hash(String s) {
      h = 0;
      for (i = s.length() - 1; i >= 0; i--) {
            h = (s.keyAt(i) + h<<5) % tableSize;
      }
      return h;
}
```

- minimum number of multiplications (handled by shifts!)
- avoids overflow, because is doing mods during computation

Hash Collision Handling

-Wat the Collision?

-Hashing cock block a cuh. Trying to insert into an index that's already occupied.



-If John Smith is added first, Sandra Dee is the *primary collision*.

-Seperate Chaining (Closed Addressing)

-Treat each element/index as a another data structure/list.

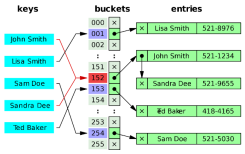
-Overflow Area (Array): A smaller subarea in an array that corresponds to the collision entry of the index.

-Linked List: Each index is a root.

-New Collection: Whatever list type beat like BST (meaning the worst case is O(logn), but hopefully most operation is O(1)).

The *chaining* method for handling collisions simply treats the hash table conceptually as a table of collections rather than as a table of individual cells.

- Overflow Area
- Linked List
- New Collection



-Ex (Overflow Area): Insert 10, 22, 107, 12, 42 in a Hash Table of size = 13, using division method on an available table size of 10.

-So we leave 10, 11, 12 for collision entries.

-22 collides with 12 (primary), so we add 12 to the 1st index of the overflow area (10).

-Then, 22 & 12 collides with 42 so we add it to the 2nd index of the overflow area (11).

-If we remove 22 afterwards, we should use some flag @ index 2 to indicate that 12 & 42 are in overflow area. Don't shift during deletion.

-Extra: Alt is to use Buckets Array, where you allocate multiple indexes for one actual index (ie allocate 3 spots for each index, overflow on 4th collision)

-Ex (LL): Insert 10, 22, 107, 12, 42 in a Hash Table of size = 10, using division method.

-So when 22 collides with 12, we add to the front / before 22 to be O(1).

-In the end, it'll be 42, 12, 22.

-If you have like ½ of the keys mapping to the some index, tweak the hashing function.

-Open Addressing

-Where you look for another position in the same hash table (overflow area doesn't count).

-Kinda like Bucket Array but the buckets are dynamic and the indexes is found by the probing algorithm.

-Useful when you can't do Closed Addressing because things like memory constraint.

-Linear Probing

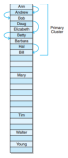
-If collision, retry at index = ((hash(key) + i) % size) where i is the retry counter = 1,2,...Size-1.

-Like to create clusters, so bad distribution and therefore slower search & insert.

- In *linear probing*, if an element hashes to position p and position p is already occupied, we simply try position (p+1) % s, where s is the size of the table.
- If position (p+1) % s is already occupied, we try position (p+2)%s, and so on until either we find an open position or we find ourselves back at the original position.

Problem:

- The problem with linear probing is that it tends to create clusters of filled positions within the table, and these clusters then affect the performance of insertions and searches.



-Quadratic Probing

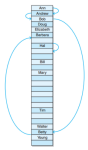
-If collision, retry at index = ((hash(key) + i^2) % size) where i is the retry counter = 1,2,...Size-1.

-With i^2, you can fail inserting because you get a loop of the same index when probing.

$$newhashcode(x) = (hashcode(x) + (i)^2)\% s$$

for i in the range 1 to s–1 where s is the table size

- Followed by the division method to keep the hash code in range of the table size.
- The benefit of the quadratic probing method is that it does not have as strong a tendency toward clustering as does linear probing.



-Double Hashing

-Use a secondary hash function and an offset using the prev hash code/index to find a different index when the normal hash function collides.

-Let p be the first hash code and p' the 2nd, p'' the 3rd…

-p = hash(key) % size

-p' = ((p * 1) + hash2(key)) % size

-p'' = ((p * 2) + hash2(key)) % size

-p_i = ((p * i) + hash2(key)) % size where i is the retry counter = 1,2,...Size-1.

-Alternative:

-p = hash(key) % size

-p' = (hash(key) + hash2(key)) % size (but hash2 must ultilize a prime number)

-Ex: p' = prevIndexOffset(key, p) + reverse(key) *(here, the (p * i) is the prevIndexOffset())*

-reverse(key) is the 2nd hash function, that reverses the digits of an int key.

-prevIndexOffset(key, p) = key+p, takes in the int index that previously had a collision to offset it with the key.

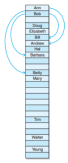
- Using the *double hashing* method, we will resolve collisions by providing a secondary hashing function to be used when the primary hashing function results in a collision.
- For example, if a key x hashes to a position p that is already occupied, then the next position p that we will try will be

```
p' = (p + secondaryhashcode(x)) % s
```

- If this new position is also occupied, then we look to position

```
p" = (p + 2 * secondaryhashcode(x)) % s
where s = table size
```

- This method, while somewhat more costly because of the introduction of an additional function, tends to further reduce clustering beyond the improvement gained by quadratic probing.



-Cuckoo Hashing (Extra)

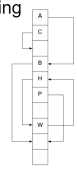
-Like the bird, kicks out the original elem at the index

Open Addressing: Cuckoo Hashing

- The basic idea is to use two hash functions instead of only one. This provides two possible locations in the hash table for each key.
- When a new key is inserted, a greedy algorithm is used: The new key is inserted in one of its two possible locations, "kicking out", that is, displacing, any key that might already reside in this location. This displaced key is then inserted in its alternative location, again kicking out any key that might reside there, until a vacant position is found, or the procedure would enter an infinite loop. In the latter case, the hash table is rebuilt in-place using new hash functions:

  hashcode(k) = k % table size

  hashcode'(k) = [k / table size] % table size



Hashing Operations and Load Factor

-Deleting Elements

Chaining with Overflow Area

- **Case 1:** The element we are attempting to remove is the only one mapped to the particular location in the table.
- **Case 2:** The element we are attempting to remove is stored in the table (not in the overflow area) but has an index into the overflow area for the next element at the same position.
- **Case 3:** The element we are attempting to remove is at the end of the list of elements stored at that location in the table.
- **Case 4:** The element we are attempting to remove is in the middle of the list of elements stored at that location in the table.
- **Case 5:** The element we are attempting to remove is not in the list.

Open Addressing

Problem:

What happens if we now remove "Andrew"? If we then search for "Amy" we will not find that element because the search will find "Ann" and then follow the linear probing rule to look in the next position, find it null, and return an exception.

-Load Factor

-The ratio between the # of elements and the table size

-Low means efficient

-Adding more to the hash table raises the Load Factor.

-Load Factor max should be 1, because > 1 means there are too many data to add.

- The load factor α of a hash table with n elements is given by the following formula:

$$\alpha = \frac{n}{Table\ Size}$$

  - Where n = number of elements to be hashed

-Separate Chaining Notes

-Worst case is that everything is a collision, O(n)

-There are other trade-offs, like Array VS LinkedList implementation, Bucket Array?

-Space is traded off for Time in all cases with hashing

- Worst-case time for **find**: linear
  - But only with really -bad luck or bad hash function
  - So not worth avoiding (e.g., with balanced trees at each bucket/index)
- Beyond asymptotic complexity, some "data-structure engineering" may be warranted
  - Linked list vs. array (lists should be short!)
  - Move-to-front
  - Better idea: Leave room for 1 element (or 2?) in the table itself, to optimize constant factors for the common case
- A time-space trade-off…

-Load Factor, Successful vs Unsuccessful Searches. (Unsuccessful means O(n) search)

- **Definition:** The load factor, **α**, of a hash table is:

α = n / Table Size

- Under chaining, the average number of elements per bucket is **α**
- So if some inserts are followed by *random* finds, then on average:
  - Each unsuccessful **find** compares against **α** items
  - Each successful **find** compares against **α / 2** items
  - If is **α** low, **find** and **insert** likely to be O(1)

successful = $\frac{\alpha}{2}$ = 1 $\alpha = \frac{40}{20} = 2$

unsucc = α = 2

-Open Addressing and Load Factor

- This is *one example* of open addressing
- In general, open addressing means resolving collisions by trying a sequence of other positions in the table.
- Trying the next spot is called probing
  - Our i^{th} probe was **(h(key) + i) % Table Size**
    - This is called linear probing
  - In general have some probe function **g** and use

    h(key) + g(i) % Table Size
- Open addressing does poorly with high load factor **α**
  - So want larger tables
  - Too many probes means no more *O*(1)

-Linear & Quadratic Probing and Load Factor

- It turns out linear probing is a *bad idea*, even though the probe function is quick to compute (a good thing)
- Tends to produce *clusters*, which lead to long probing sequences called primary clustering

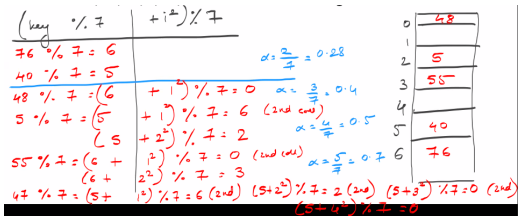
-Load Factor

- Trivial fact: For any $\alpha < 1$, linear probing will find an empty slot
  - It is "safe" in this sense: no infinite loop unless table is full
- Non-trivial facts we won't prove:
  Average # of probes given α (in the limit as Table Size → ∞)
  - Unsuccessful Search: $\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$
  - Successful Search: $\frac{1}{2}\left(1+\frac{1}{(1-\alpha)}\right)$
- This is pretty bad: need to leave sufficient empty space in the table to get decent performance

- Thus, $0 < \alpha < 1$ for linear probing. (α can be greater than 1 for other collision resolution methods)
- For linear probing, as α approaches 1, the number of collisions increases
  - Performance quickly degrades for $\alpha > 1/2$
- The probability of a collision increases as the load factor increases.

-Ex: Quadratic Probing & Load Factor

Insert 76, 40, 48, 5, 55, 47 in a table size = 7



-Then we started looping when Load Factor become high.

-We should keep Load Factor < ½, we don't have to detect cycles

-That means resize when Load Factor is ½

- The bad news is: After **Table Size** quadratic probes, we will just cycle through the same indices
- The good news:
  - If **Table Size** is *prime* and $\alpha < 1/2$, then quadratic probing will find an empty slot in at most **Table Size /2** probes.
  - So: If you keep $\alpha < 1/2$, no need to detect cycles
- Quadratic probing does not suffer from primary clustering:
  - no problem with keys initially hashing to the same neighborhood
- But it's no help if keys initially hash to the same index
  - called secondary clustering
- Can avoid secondary clustering with a probe function that depends on the key: double hashing...

-Double Hashing and Load Factor

- Intuition: Since each probe is "jumping" by **g(key)** each time, we "leave the neighborhood" *and* "go different places from other initial collisions"
- But we could still have a problem like in quadratic probing where we are not "safe" (infinite loop despite room in table)
- It is known that this cannot happen in at least one case:
  - **h(key) = key % p (p here is Table Size)**
  - **g(key) = q – (key % q)**
  - **2 < q < p**
  - **p** and **q** are prime

-Load Factor

- Average # of probes given α (in the limit as Table Size → ∞)
  - Unsuccessful Search: $\frac{1}{(1-\alpha)}$
  - Successful Search: $\frac{1}{\alpha}\log_e\left(\frac{1}{(1-\alpha)}\right)$
- Bottom line: unsuccessful bad (but not as bad as linear probing), but successful is not nearly as bad

-Rehashing

-Trigger it based on Load Factor (ie keeping it < 1)

-Double the size of the table and rehash. Takes a while O(n), so hopefully it's rare

- Like with array-based stacks/queues/lists, if table gets too full, create a bigger table and copy everything over
- Especially with chaining, we get to decide what "too full" means
  - Keep load factor reasonable (e.g., < 1)?
  - Consider average or max size of non-empty chains?
  - For open addressing, half-full is a good rule of thumb
    - Or, we can rehash only when insert fails
- New table size
  - Twice-as-big is a good idea, except, uhm, that won't be prime!
  - So go *about* twice-as-big
  - Can have a list of prime numbers in your code since you won't grow more than 20-30 times

- We double the size (rather than "add 1000") to get good amortized guarantees
- But one resize is an $O(n)$ operation, involving n calls to the hash function (1 for each insert in the new table)
- Space/time tradeoff: Could store **h(key)** with each data item, but since rehashing is rare, this is probably a poor use of space
  - And growing the table is still $O(n)$

Note: `HashMap` and `HashSet` use a default load factor of 0.75 and an initial capacity of 16.

-Summary

- Hashing is the best search method (constant running time) if we don't need to have the records sorted.
- The **choice of the hash function** remains the most difficult part of the task and depends very much on the nature of the keys.
- **Separate chaining or open addressing?**
  - **Open addressing** is the preferred method if there is enough memory to keep a table twice larger than the number of the records.
  - **Separate chaining** is used when we don't know in advance the number of the records to be stored. Though it requires additional time for list processing, it is simpler to implement.
- **Some application areas**

Dictionaries, on-line spell checkers, compiler symbol tables

- Chaining is easy
  - **insert**, **find**, **delete** proportion to load factor on average
- Open addressing uses probe functions, has clustering issues as table gets full
  - Why use it:
    - Less memory allocation?
    - Easier data representation?

-Ex1: Using Load Factor

-Using Linear Probing

$n = 60{,}000$

$\alpha = 0.75$

-1. With the # of elems and the load factor, we can find the table size

$\alpha = \frac{n}{\text{table size}}$ $\therefore$ table size $= \frac{n}{\alpha} = \frac{60000}{0.75} = 80{,}000$

-2. Avg # of Successful Search

$$= \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right)$$

$$= \frac{1}{2}\left(1 + \frac{1}{(1-0.75)}\right) = \frac{1}{2}\left(1 + \frac{1}{0.25}\right)$$

$$= 2.5 \text{ comparisons}$$

(meaning on avg, its takes 2.5 comparison to successfully search)

-Ex2: Using Load Factor

-Using Chaining

$n = 60{,}000$ coll → chaining

Avg # of comp(succ) for search = 2.5 (So what is our table size?)

-1. With n & avg # of comparison, we can find the load factor

Avg # of comp (chaining) = $1 + \frac{\alpha}{2}$

$2.5 = 1 + \frac{\alpha}{2}$

$\therefore 1.5 = \frac{\alpha}{2}$

$\therefore \boxed{\alpha = 3.0}$

-2. Now we can find table size

$\alpha = \frac{n}{\text{table size}}$

$\therefore$ table size $= \frac{60000}{3.0}$

$\boxed{\text{table size} = 20{,}000}$

Graph Theory

-Definition of Graph

-Useful to represent relationships. Vector (V) & Edge (E).

- A graph is a finite set of nodes with edges between nodes
- Formally, a graph G is a structure (V,E) consisting of
  - a finite set V called the set of nodes, and
  - a set E that is a subset of VxV. That is, E is a set of pairs of the form (x,y) where x and y are nodes in V

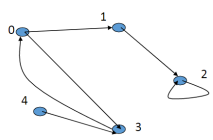
- The nodes represent entities (such as people, cities, computers, words, etc.)
- Edges (x,y) represent relationships between entities x and y, such as:
  - "x loves y"
  - "x hates y"
  - "x is a friend of y" (note that this not necessarily reciprocal)
  - "x considers y a friend"
  - "x is a child of y"
  - "x is a half-sibling of y"
  - "x is a full-sibling of y"
- In those examples, each relationship is a different graph

-Usually, a

-Ex: (and def of adjacent)

- V={0,1,2,3,4}
- E={(0,1), (1,2), (0,3), (3,0), (2,2), (4,3)}

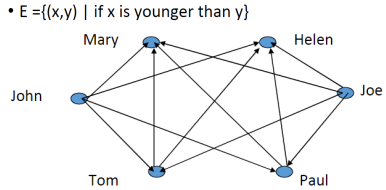
When (x,y) is an edge, we say that x is *adjacent to* y, and y is *adjacent from* x.



0 is adjacent to 1.
1 is not adjacent to 0.
2 is adjacent from 1.

-Ex:

- V=set of 6 people: John, Mary, Joe, Helen, Tom, and Paul, of ages 12, 15, 12, 15, 13, and 13, respectively.
- E ={(x,y) | if x is younger than y}



-History: Seven Bridges of Königsberg

-We need a spaning graph?

Problem: To devise a walk through the city that would cross each bridge once and only once, with the provision that: the islands could only be reached by the bridges and every bridge once accessed must be crossed to its other end. The starting and ending points of the walk need not be the same.

Solution: (Leonhard Euler) Euler Circuit or Euler Path

-Representing Graphs

-Adjacency Matrix Representation

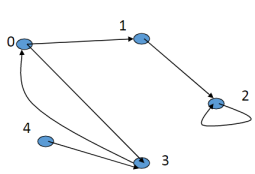
-Just a 2D array where the sides are vectors x vectors.

- In this representation, each graph of n nodes is represented by an n x n matrix A, that is, a two-dimensional array A
- The nodes are (re)-labeled 1,2,…,n
- A[i][j] = 1 if (i,j) is an edge
- A[i][j] = 0 if (i,j) is not an edge

- Pros:
  - Simple to implement
  - Easy and fast to tell if a pair (i,j) is an edge: simply check if A[i][j] is 1 or 0
- Cons:
  - No matter how few edges the graph has, the matrix takes $O(n^2)$ in memory

-Ex:

$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$



-You read it as the column to row. So if you want to add an edge from 3 to 0, you'll need to change A[3,0] = 1.

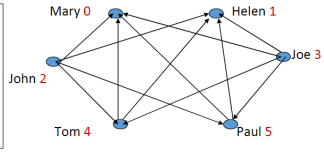
-Size is O(V^2).

-Adding an edge and query is O(1)

-Adding another vector, resize needed.

-Ex:

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$



-Adjacency Lists Representation

-LL for each Vertex which has elements denote edge to other.

- A graph of n nodes is represented by a one-dimensional array L of linked lists, where
  - L[i] is the linked list containing all the nodes adjacent from node i.
  - The nodes in the list L[i] are in no particular order

- Pros:
  - Saves on space (memory): the representation takes as many memory words as there are nodes and edge.
- Cons:
  - It can take up to O(n) time to determine if a pair of nodes (i,j) is an edge: one would have to search the linked list L[i], which takes time proportional to the length of L[i].

-Ex:

L[0]: empty

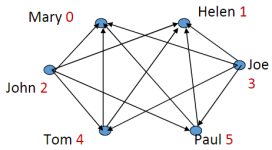
L[1]: empty

L[2]: 0, 1, 4, 5

L[3]: 0, 1, 4, 5

L[4]: 0, 1

L[5]: 0, 1



-Compared with Matrix Representation, this uses space only if there is adjacency. Size = O(V+E)

-Search is slower because LL, O(E)

-Adding a edge or vector is O(1)

-Removing edge is O(E), because LL iterate through to delete.

-Graph Operations

The basic operations provided by a graph data structure *G* usually include:

- adjacent(*G*, *x*, *y*): tests whether there is an edge from the vertices *x* to *y*;
- neighbors(*G*, *x*): lists all vertices *y* such that there is an edge from the vertices *x* to *y*;
- add_vertex(*G*, *x*): adds the vertex *x*, if it is not there;
- remove_vertex(*G*, *x*): removes the vertex *x*, if it is there;
- add_edge(*G*, *x*, *y*): adds the edge from the vertices *x* to *y*, if it is not there;
- remove_edge(*G*, *x*, *y*): removes the edge from the vertices *x* to *y*, if it is there;
- get_vertex_value(*G*, *x*): returns the value associated with the vertex *x*;
- set_vertex_value(*G*, *x*, *v*): sets the value associated with the vertex *x* to *v*.

Structures that associate values to the edges usually also provide:

- get_edge_value(*G*, *x*, *y*): returns the value associated with the edge (*x*, *y*);
- set_edge_value(*G*, *x*, *y*, *v*): sets the value associated with the edge (*x*, *y*) to *v*.

-Time Complexity of the 2 Graphs Covered

Node/Edge Mgmt	Storage	Add Edge	Add Vertex	Remove Edge	Remove Vertex	Query
Adjacency List	O(|V|+|E|)	O(1)	O(1)	O(|E|)	O(|V|+|E|)	O(|V|)
Adjacency Matrix	O(|V|^2)	O(1)	O(|V|^2)	O(1)	O(|V|^2)	O(1)

-Properties of Graph

-Directed VS Undirected

-Are the edges only relating one way (Directed, usually denoted with arrow) or both way (Undirected)

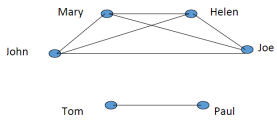
-Tree is a Directed graph because children only expand one way.

-Undirected graphs are symmetrical.

- If the directions of the edges matter, then we show the edge directions, and the graph is called a *directed graph* (or a *digraph*)
- The previous two examples are digraphs
- If the relationships represented by the edges are symmetric (such as (x,y) is edge if and only if x is a sibling of y), then we don't show the directions of the edges, and the graph is called an *undirected graph*.

-Ex: Undirected

- V=set of 6 people: John, Mary, Joe, Helen, Tom, and Paul, where the first 4 are siblings, and the last two are siblings
- E ={(x,y) | x and y are siblings}



if (x,y) is an edge: we say that x is *adjacent to* y, & y adjacent to x. We also say that x and y are *neighbors*

Linked Lists:

L[0]: 1, 2, 3

L[1]: 0, 2, 3

L[2]: 0, 1, 3

L[3]: 0, 1, 2

L[4]: 5

L[5]: 4

Adjacency Matrix:

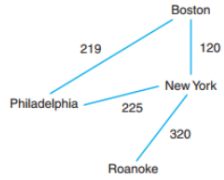
$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

- The same two representations for directed graphs can be used for undirected graphs
- Adjacency matrix A:
  - A[i][j]=1 if (i,j) is an edge; 0 otherwise
- Adjacency Lists:
  - L[i] is the linked list containing all the neighbors of i

-Networks

-A weighted graph. The edge stores some value of weight/cost

-Ex: GPS between cities. The numbers represent distance between cities.



$$\begin{array}{c} \\ B \\ P \\ N \\ R \end{array} \begin{array}{c} \begin{array}{cccc} B & P & N & R \end{array} \\ \begin{bmatrix} 0 & 219 & 120 & 0 \\ 219 & 0 & 225 & 0 \\ 120 & 225 & 0 & 300 \\ 0 & 0 & 300 & 0 \end{bmatrix} \end{array}$$

-Degree of a Vector

-How many incoming and outgoing edges are there of a vector?

- Let G be a directed graph
  - The *indegree* of a node x in G is the number of edges coming to x
  - The *outdegree* of x is the number of edges leaving x.
- Let G be an undirected graph
  - The *degree* of a node x is the number of edges that have x as one of their end nodes
  - The *neighbors* of x are the nodes adjacent to x

-Complete

-Every vector has edges to each other, except self.

- An undirected graph is considered *complete* if it has the maximum number of edges connecting vertices.

No of Vertices (V) = n

No of Edges (E) = $\frac{n(n-1)}{2}$

-Paths

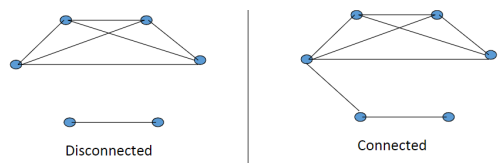
-A sequence of node that represent some path, hierarchy, connection, etc.

- A path in a graph G is a sequence of nodes $x_1, x_2, \ldots, x_k$, such that there is an edge from each node the next one in the sequence
- For example, in the first example graph, the sequence 3, 0, 1, 2 is a path, but the sequence 0, 3, 4 is not a path because (0,3) is not an edge
- In the "sibling-of" graph, the sequence **John, Mary, Joe, Helen** is a path, but the sequence **Helen, Tom, Paul** is not a path

-Connectivity

-When you can path to each vector from any vector.

- An undirected graph is said to be *connected* if there is a path between every pair of nodes. Otherwise, the graph is *disconnected*
- Informally, an undirected graph is connected if it hangs in one piece
- If an undirected graph is not connected, then each "piece" is called a connected component.
  - A piece in itself is connected, but if you bring any other node to it from the graph, it is no longer connected
- If the graph is connected, then the whole graph is one single connected component
- Of Interest: Given any undirected graph G,
  - Is G connected?
  - If not, find its connected components.



-Extra:

-There are actually 4 represpresentation

-Incident property?

Graph Traversal

-2 Types of Traversal

-Depth First Search (uses stacks/recurssion)

-Breadth First Search (uses queue)

-Each creates a tree because it need to start from a point.

-Each needs a way to flag that the node/vector is already visited (ie set)

-If you only want to find a path, you can stop the search after dequeuing/popping the one you are searching for.

- In both DFS and BFS, the nodes of the undirected graph are visited in a systematic manner so that every node is visited exactly one.
- Both BFS and DFS give rise to a tree:
  - When a node x is visited, it is labeled as visited, and it is added to the tree
  - If the traversal got to node x from node y, y is viewed as the parent of x, and x a child of y

-Depth First Search

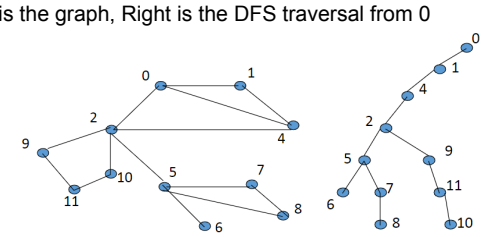
-Steps

-Commit to a path, add all V to on that path (a straight line on tree), find its dead-end then backtrack (new branch on tree).

1. Select an unvisited node x, visit it, and treat as the current node
2. Find an unvisited neighbor of the current node, visit it, and make it the new current node;
3. If the current node has no unvisited neighbors, backtrack to the its parent, and make that parent the new current node;
4. Repeat steps 3 and 4 until no more nodes can be visited.
5. If there are still unvisited nodes, repeat from step 1.

- Observations:
  - the last node visited is the first node from which to proceed.
  - Also, the backtracking proceeds on the basis of "last visited, first to backtrack too".
  - This suggests that a stack is the proper data structure to remember the current node and how to backtrack.

-Ex: Left is the graph, Right is the DFS traversal from 0



Graph G

DFS Tree

-Psuedo Code

```
DFS(input: Graph G) {
  Stack S; Integer x, t;
      while (G has an unvisited node x){
       visit(x); push(x,S);
       while (S is not empty){
            t := peek(S);
            if (t has an unvisited neighbor y){

                  visit(y);
                  push(y,S);
            }
            else
                  pop(S);
       }
      }
}
```

-Breadth First Search

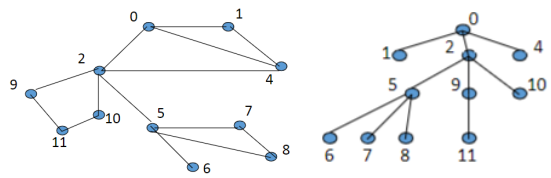
-Steps (Similar to Level Order)

-Look at all connecting V,

1. Select an unvisited node x, visit it, have it be the root in a BFS tree being formed. Its level is called the current level.
2. From each node z in the current level, in the order in which the level nodes were visited, visit all the unvisited neighbors of z. The newly visited nodes from this level form a new level that becomes the next current level.
3. Repeat step 2 until no more nodes can be visited.
4. If there are still unvisited nodes, repeat from Step 1.

- Observations:
  - the first node visited in each level is the first node from which to proceed to visit new nodes.
- This suggests that a queue is the proper data structure to remember the order of the steps.

-Ex:



Graph G

BFS Tree

(Less height compared to DFS)

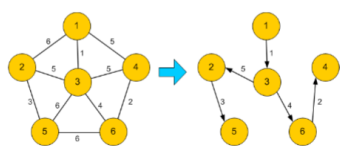
-Psuedocode

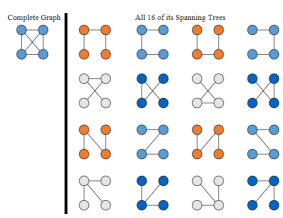
```
BFS(input: graph G) {
     Queue Q;    Integer x, z, y;
     while (G has an unvisited node x) {
      visit(x); Enqueue(x,Q);
      while (Q is not empty){
           z := Dequeue(Q);
           for all (unvisited neighbor y of z){
                visit(y);
                Enqueue(y,Q);
           }
      }
    }
}
```

Spanning Tree

-Wat the Spanning Tree?

- A *spanning tree* is a tree that includes all of the vertices of a graph and some, but possibly not all, of the edges.

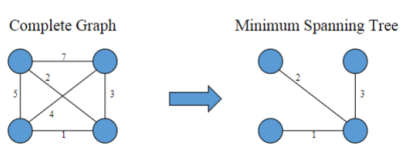




-Minimal Spanning Tree (for weighted)

-The least edges with the least weights.

The Minimum Spanning Tree for a given graph is the Spanning Tree of minimum cost for that graph.



-Applications

- **Network design.**
  – *telephone, electrical, hydraulic, TV cable, computer, road*
- **Cluster analysis**
  k clustering problem can be viewed as finding an MST and deleting the k-1 most expensive edges.
- **Approximation algorithms for NP-hard problems.**
  – *travelling salesman problem*

-Minimum Spanning Tree Algorithms

-Kruskal's Algorithm

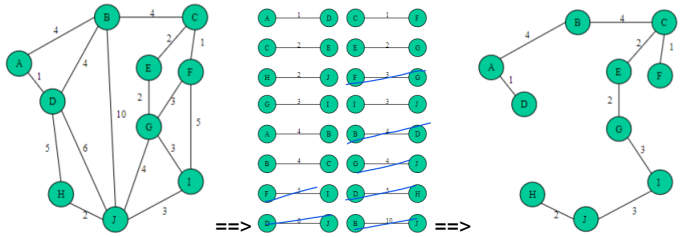
-Read all edges, sort edges by min weight (PQ), go through each of the sorted edges discard edges that make a cycle (keep Set of visited V)

1. The forest is constructed - with each node in a separate tree.
2. The edges are placed in a priority queue.
3. Until we've added n-1 edges,
   1. Extract the cheapest edge from the queue,
   2. If it forms a cycle, reject it,
   3. Else add it to the forest. Adding it to the forest will join two trees together.

Every step will have joined two trees in the forest together, so that at the end, there will only be one tree in T.

This algorithm creates a forest of trees. Initially the forest consists of n single node trees (and no edges). At each step, we add one edge (the cheapest one) so that it joins two trees together. If it were to form a cycle, it would simply link two nodes that were already part of a single connected tree, so that this edge would not be needed.

-Ex:



-Time Analysis O(Elog(V))

Running Time = O(m log n) (m = no of edges, n = no of vertices)

Testing if an edge creates a cycle can be slow unless a complicated data structure called a "union-find" structure is used.

It usually only has to check a small fraction of the edges, but in some cases (like if there was a vertex connected to the graph by only one edge and it was the longest edge) it would have to check all the edges.

This algorithm works best, of course, if the number of edges is kept to a minimum.

-Prim's Algorithm

-Start with a V, start branching out by selecting lowest cost edge form visited vertices, stop when Set of visited

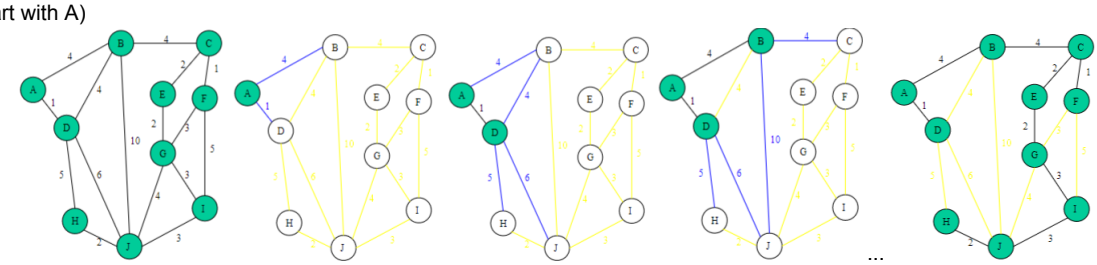
-Unlike Kruskal's, it will stop at a dead end instead of finding spanning for all disconnect clusters

1. The new graph is constructed - with one node from the old graph.
2. While new graph has fewer than n nodes,
   1. Find the node from the old graph with the smallest connecting edge to the new graph,
   2. Add it to the new graph

Every step will have joined one node, so that at the end we will have one graph with all the nodes and it will be a minimum spanning tree of the original graph.

This algorithm starts with one node. It then, one by one, adds a node that is unconnected to the new graph to the new graph, each time selecting the node whose connecting edge has the smallest weight out of the available nodes' connecting edges.

-Ex: (Start with A)



-Time Complexity O(E + Vlog(V)) with heap

Running Time = O(m + n log n) (m = no of edges, n = no of vertices)

If a heap is not used, the run time will be O(n^2) instead of O(m + n log n). However, using a heap complicates the code since you're complicating the data structure. A Fibonacci heap is the best kind of heap to use, but again, it complicates the code.

Unlike Kruskal's, it doesn't need to see all of the graph at once. It can deal with it one piece at a time. It also doesn't need to worry if adding an edge will create a cycle since this algorithm deals primarily with the nodes, and not the edges.

Searching & Sorting Algorithms

-Searching Array

-Linear Search O(n) look at every elem for target

- Linearly search through an unsorted array
- Return index of the element found, else -1
- Time Complexity: O(n)

-Binary Search O(logn) half cheese

- Need array elements to be in a sorted order
- Return index of the element found,
  - Else, -(insertion point)
- Time Complexity: ?

-Ex: Search 64

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	
11	18	19	22	24	35	37	64	68	Find middle
11	18	19	22	24	35	37	64	68	Find half
11	18	19	22	24	35	37	64	68	Find middle
11	18	19	22	24	35	37	64	68	Find half
11	18	19	22	24	35	37	64	68	Find middle
									Found

-Ex: Search 20 (returns -3)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	
11	18	19	22	24	35	37	64	68	Find middle
11	18	19	22	24	35	37	64	68	Find half
11	18	19	22	24	35	37	64	68	Find middle
11	18	19	22	24	35	37	64	68	Find half
11	18	19	22	24	35	37	64	68	Find middle
11	18	19	22	24	35	37	64	68	Find half
11	18	19	22	24	35	37	64	68	Find middle
...									Not found

```
int binarySearch(T[] arr, T key):
        int first = 0, last = arr.length - 1
        while (first <= last)
                mid = (first + last) / 2
                if (key < arr[mid])
                        last = mid - 1
                else if (key > arr[mid]) //Bad optimization, does compare 2x
                        first = mid + 1
                else
                        return mid
        return -mid //Indicate where 20 should be inserted
```

-Why O(logn)?

$$1st = \frac{n}{2} \cdots \frac{n}{2^1}$$

$$2nd = \frac{\frac{n}{2}}{2} = \frac{n}{4} \cdot \frac{n}{2^2}$$

$$3rd = \frac{\frac{n}{4}}{2} = \frac{n}{8} = \frac{n}{2^3}$$

$$\vdots \qquad \vdots$$

$$k^{th} = \frac{n}{2^k} = 1 \qquad \therefore 2^k = n \quad k = (\log_2 n)$$

-Other Algorithms

- Interpolation Search
  - more like what people really do
- Indexed Searching
- Binary Search Trees
- Hash Table Searching
- Grover's Algorithm (Waiting for quantum computers to be built)
- best-first
- A*

-Sorting Algorithms (Visualization 5 min be like)

-O(n^2) Algorithms

-Selection Sort

-1. Go through entire array, find smallest, swap it with index 0.
-2. Go throught entire array find next smallest, swap it with index 1.
-3. Repeat for other elem.
-Doesn't have a mechanic to know if the array is sorted, so always O(n^2)
-In place so O(1) space.

- Given an array of length n,
  - Search elements 0 through n-1 and select the smallest
    - Swap it with the element in location 0
  - Search elements 1 through n-1 and select the smallest
    - Swap it with the element in location 1
  - Search elements 2 through n-1 and select the smallest
    - Swap it with the element in location 2
  - Search elements 3 through n-1 and select the smallest
    - Swap it with the element in location 3
  - Continue in this fashion until there's nothing left to search

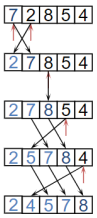
- The selection sort might swap an array element with itself--this is harmless, and not worth checking for
- Analysis:
  - The outer loop executes n-1 times
  - The inner loop executes about n/2 times on average (from n to 2 times)
  - Work done in the inner loop is constant (swap two array elements)
  - Time required is roughly (n-1)*(n/2)
  - You should recognize this as $O(n^2)$

```
void selectionSort(T[] arr):
        for i → 0 to arr.length-2 //end before the last elem
                int min = i //index of smallest elem
                for j → i+1 to arr.length-1 //search for smallest from i+1
                        if (arr[j] < arr[min])
                                min = j
                swap(arr, i , min) //Helper swap method
```

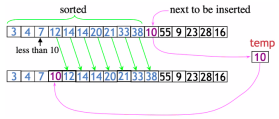
-Insertion Sort

- The outer loop of insertion sort is:
  for (outer = 1; outer < a.length; outer++) {...}
- The invariant is that all the elements to the left of outer are sorted with respect to one another
  - For all i < outer, j < outer, if i < j then a[i] <= a[j]
  - This does *not* mean they are all in their final correct place; the remaining array elements may need to be inserted
  - When we increase outer, a[outer-1] becomes to its left; we must keep the invariant true by inserting a[outer-1] into its proper place
  - This means:
    - Finding the element's proper place
    - Making room for the inserted element (by shifting over other elements)
    - Inserting the element

- We run once through the outer loop, inserting each of n elements; this is a factor of n
- On average, there are n/2 elements already sorted
  - The inner loop looks at (and moves) half of these
  - This gives a second factor of n/4
- Hence, the time required for an insertion sort of an array of n elements is proportional to $n^2/4$
- Discarding constants, we find that insertion sort is $O(n^2)$



-Splits aray into 2 parts, left is sorted, right is unsorted.
-Each step, take the leftmost elem of the right/unsorted part and start compare & swapping the prev index (in left/sorted) until in place.




```
void insertionSort(T[] arr):
        for i → 1 to arr.length-1
                j=i
                while (arr[j] < arr[j-1] && j > 0)
                        swap(arr, j, j-1)
                        j--
```

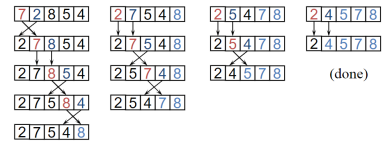
-Knowing that the left split is already sorted, Insertion knows when to break during each iteration.
-So best is O(n), on a sorted array, unlike Selection.

-Bubble Sort

-1. Sweep through from 0 to size - 2 (one before end), compare curr to next (adjacent elem), swap if out of place.
-This puts the largest to the end
-2. Repeat from 0 to size - 2 - 1, then size - 2 - 2,… then (size - 2 - (size - 3)) until you are done.

- Compare each element (except the last one) with its neighbor to the right
  - If they are out of order, swap them
  - This puts the largest element at the very end
  - The last element is now in the correct and final place
- Compare each element (except the last *two*) with its neighbor to the right
  - If they are out of order, swap them
  - This puts the second largest element next to last
  - The last two elements are now in their correct and final places
- Compare each element (except the last *three*) with its neighbor to the right
  - Continue as above until you have no unsorted elements on the left

- Let n = a.length = size of the array
- The outer loop is executed n-1 times (call it n, that's close enough)
- Each time the outer loop is executed, the inner loop is executed
  - Inner loop executes n-1 times at first, linearly dropping to just once
  - On average, inner loop executes about n/2 times for each execution of the outer loop
  - In the inner loop, the comparison is always done (constant time), the swap might be done (also constant time)
- Result is n * (n/2 + k), that is, $O(n^2/2 + k) = O(n^2)$




```
bubbleSort(T[] arr):
        for i → arr.size-2 to 0
                boolean hasSwapped = false //if never swapped, then it's sorted.
                for j → 0 to i-1
                        if (arr[j] > arr[j+1])
                                swap(arr, j, j+1)
                                hasSwapped = true
                        else if (hasSwapped = false)
                                return
```

-Also more efficient than selection sort because the inner loop doesn't always swap and the hasSwapped flag let it know when to stop.

-Divide and Conquer

-Recursively sort the list by split the list in some way.

- **Divide** the problem into a number of sub-problems
  - Similar sub-problems of smaller size
- **Conquer** the sub-problems
  - Solve the sub-problems recursively
  - Sub-problem size small enough ⇒ solve the problems in straightforward manner
- **Combine** the solutions of the sub-problems
  - Obtain the solution for the original problem

-Merge Sort

-1. Split array into halves.

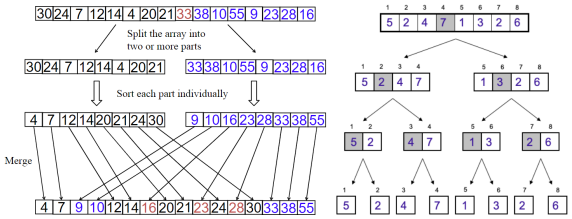
-Optional. Keep splitting into even smaller part.

-2. Merge back into a temp/result array by comparing the smallest left vs right and putting that into temp/result array.

-Good for extremely large data size that doesn't fit in memory because it divides and works on some sections at a time.

Don Knuth cites **John von Neumann** as the creator of this algorithm:

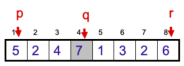
- Divide and Conquer
- Recursively:
  - Split the list till elements in list < 2
  - Take 2 sorted list and merge them together



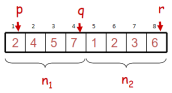
-Though these pic make it look like it runs in parallel, it actually works from left to right. The order of list creation is like preorder trav.

-Psuedocode (mergeSort(T arr[], p, r) and merge(T arr[], p, q, r))

```
MERGE-SORT(A, p, r)

  if p < r

  then q ← ⌊(p + r)/2⌋

      MERGE-SORT(A, p, q)

      MERGE-SORT(A, q + 1, r)

      MERGE(A, p, q, r)
```




```
MERGE(A, p, q, r)
Compute n1 and n2
Copy the first n1 elements into L[0..n1]
and the next n2 elements into R[0..n2]

i ← 0;   j ← 0
for k ← p to r
    do if L[ i ] ≤ R[ j ]
          then A[k] ← L[ i ]
               i ← i + 1
          else A[k] ← R[ j ]
               j ← j + 1
```



-q = split index, p&r = end points, A = array / temp array

-Time Complexity

-Time = O(nlogn), Space = O(n),

- Initialization (copying into temporary arrays):
  - $O(n_1 + n_2) = O(n)$
- Adding the elements to the final array:
  - n iterations, each taking constant time ⇒ O(n)
- Total time for Merge:
  - **O(n)**

- **Divide:**
  - compute q as the average of p and r: = O(1)
- **Conquer:**
  - recursively solve 2 subproblems, each of size n/2 ⇒ 2T (n/2)
- **Combine:**
  - MERGE on an n-element subarray takes O(n) time

O(1)	if n =1
T(n) = 2T(n/2) + (n)	if n > 1

Solving the Recurrence: **T(n) = O(n log n)**

-No mechanism to detect that the array is already sorted.

-Why use Merge Sort?

- Merge sort isn't an "in place" sort—it requires extra storage
- However, it doesn't require this storage "all at once"
- This means you can use merge sort to sort something that doesn't fit in memory—say, 300 million census records—then much of the data must be kept on backup media, such as a hard drive
- Merge sort is a good way to do this

- *Very roughly,* here's how to sort large amounts of data:
  - Repeat:
    - Read in as much data as fits in memory
    - Sort it, using a fast sorting algorithm (quicksort may be a good choice)
    - Write out the sorted data to a new file
  - After all the data has been written into smaller, individually sorted files:
    - Read in the initial portion of each sorted file into individual arrays
    - Start merging the arrays
    - Whenever an array becomes empty, read in more data from its file
    - Every so often, write the destination array to the (one) final output file
- When you are done, you will have one (large) sorted file

-Bottom-up Merge Sort

-Treat given array as individual lists of 1 elem, merge back to full ordered array.

5 2 4 7 1 3 2 6

2,5 4,7 1,3 2,6

2,4,5,7 1,2,3,6

1,2,2,3,4,5,6,7

-Quick Sort

-1. Choose a pivot, split the elems to 2 sublists (< pivot & >= pivot) (split doesn't make another data structure)

-Scroll down for 1a. & 1b. partitioning steps

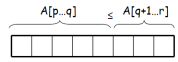
-2. Recursively quicksort (pivot and split) 1st list, then 2nd list.

Invented by **C.A.R. (Tony) Hoare**

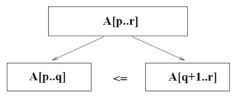
A divide and conquer approach that uses recursion:

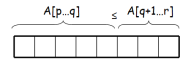
- If the list has 0 or 1 elements it is sorted
- otherwise, pick any element p in the list. This is called the pivot value
- Partition the list minus the pivot into two sub lists according to values less than or greater than the pivot (equal values go to either)
- return the quicksort of the first list followed by the quicksort of the second list

- Sort an array $A[p...r]$



- **Divide**
  - Partition the array A into 2 subarrays $A[p..q]$ and $A[q+1..r]$, such that each element of $A[p..q]$ is smaller than or equal to each element in $A[q+1..r]$
  - Need to find index q to partition the array





- **Conquer**
  - Recursively sort $A[p..q]$ and $A[q+1..r]$ using Quicksort
- **Combine**
  - Trivial: the arrays are sorted in place
  - No additional work is required to combine them
  - The entire array is now sorted

-Psuedcode

```
QUICKSORT(A, p, r)
  if p < r
    then q ← PARTITION(A, p, r)
         QUICKSORT (A, p, q)
         QUICKSORT (A, q+1, r)
```

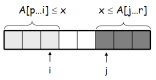
Initially: p=1, r=n

Recurrence:

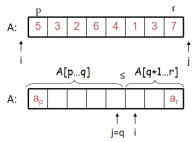
T(n) = T(q) + T(n – q) + f(n)

($f(n)$ depends on PARTITION())

- Choosing PARTITION()
  - There are different ways to do this
  - Each has its own advantages/disadvantages
- Hoare partition
  - Select a pivot element **x** around which to partition
  - Grows two regions
    $A[p...i] \le x$
    $x \le A[j...r]$




```
PARTITION (A, p, r)
x ← A[p]
i ← p - 1
j ← r + 1
while TRUE
    do repeat j ← j - 1
        until A[j] ≤ x
    do repeat i ← i + 1
        until A[i] ≥ x
    if i < j
        then exchange A[i] ↔ A[j]
    else return j
```

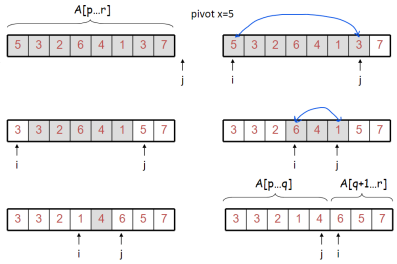


Each element is visited once!

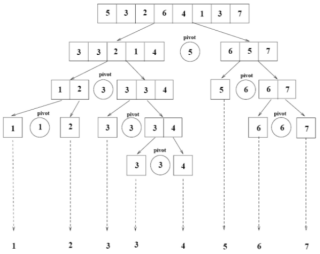
Running time: $\Theta(n)$

$n = r - p + 1$

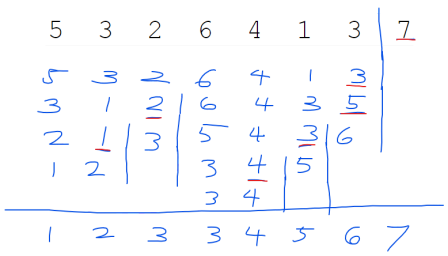
-Ex: Pivot Value = 5



-Ex: Full



-Ex: Rightmost Pivot



-1a. (In or De)crement (i or j) when the value is (< or >=) than the pivot, else it waits for the other to find the a value to switch.

-1b. The pivot must be consistently picked, like always the first elem, always the last elem, always the mid elem, etc.

-If you know that the elems are stored kind of close in order, it's better to pick pivot as midpoint elem.

-Quick3: Alternative, splits into 3 sublist, <, = , > the pivot.

-Time Analysis

-Worst Case: Each split has a sublist that is size 1, O(n^2). Merge sort is better for worst of O(nlogn).

-Best Case: Even half splits, O(nlogn)

-Avg: O(nlogn), it's not that ez to get worst case consistently

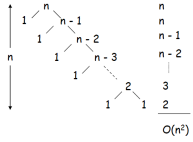
- Worst-case partitioning
  - One region has one element and the other has n – 1 elements
  - Maximally unbalanced
- Recurrence: q=1

  $T(n) = T(1) + T(n-1) + n,$

  Now $T(1) = O(1)$

  $T(n) = T(n-1) + n$

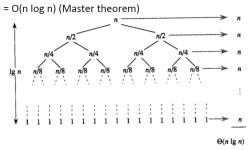
  $$= n + \left(\sum_{k=1}^{n} k\right) - 1$$
  $$= O(n) + O(n^2) = O(n^2)$$



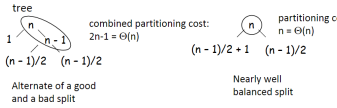
- Best-case partitioning
  - Partitioning produces two regions of size n/2
- Recurrence: q=n/2

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

  $T(n) = O(n \log n)$ (Master theorem)



- Average case
  - All permutations of the input numbers are equally likely
  - On a random input array, we will have a **mix** of well balanced and unbalanced splits
  - Good and bad splits are randomly distributed across throughout the tree



- Running time of Quicksort when levels alternate between good and bad splits is O(n log n)

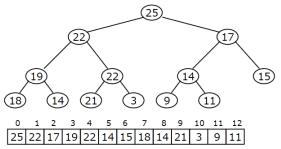
-Heap Sort

-Use Heap and spam remove. Each remove gives you the next max, so it's in ordered.

-Heap is usually an array, so removing is actually reordering, where you use the null spots as the new ordered list.

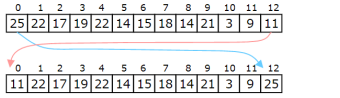
-Maxheap will sort the array be in ascending order (else you have to shift elems)

-Remember to build heap instead add() individually when create heap from a known data set.



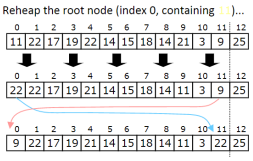
- Notice:
  - The left child of index i is at index 2*i+1
  - The right child of index i is at index 2*i+2
  - Example: the children of node 3 (19) are 7 (18) and 8 (14)

- The "root" is the first element in the array
- The "rightmost node at the deepest level" is the last element
- Swap them...

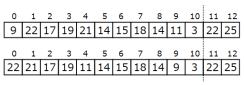


- ...And pretend that the last element in the array no longer exists—that is, the "last index" is 11 (9)

- Reheap the root node (index 0, containing 11)...



- ...And again, remove and replace the root node
- Remember, though, that the "last" array index is changed
- Repeat until the last becomes first, and the array is sorted!



-Time Analysis

- We perform the following steps *n-1* times:
  - Swap 1st and last element of the array: *constant time*
  - Reduce the size: *constant time*
  - heapify(): *log n*
- Note: We are performing `removeMax()` *n-1* times
- Running Time: **O(n log n)**

-Fat time complexity of sorting algorithm

Algorithm	Time Complexity			Space Complexity
	Best	**Average**	**Worst**	**Worst**
Quicksort	O(nlogn)	O(nlogn)	$O(n^2)$	O(logn)
Mergesort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Heapsort	O(nlogn)	O(nlogn)	O(nlogn)	O(1)
Bubblesort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Shell Sort	O(n)	$O(n(\log n)^2)$	$O(n(\log n)^2)$	O(1)
Bucket Sort	O(n+k)	O(n+k)	$O(n^2)$	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)

- Language libraries often have sorting algorithms in them
  - Java Arrays and Collections classes
  - C++ Standard Template Library
  - Python sort and sorted functions
- Hybrid sorts
  - when size of unsorted list or portion of array is small use insertion sort, otherwise use O(n log n) sort like Quicksort of Mergesort
- Many other sorting algorithms exist.

-Choosing the right sorting algorithm based on the randomness and elem variety will help reach the best-case time.