# Data Structures & Algorithms Notes

## Contents

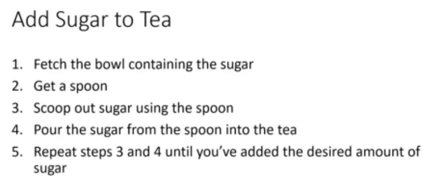
**Data Structure**

- Organises and stores data
- Each has its strengths and weaknesses

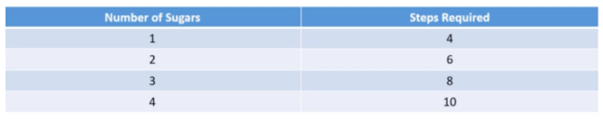
**Algorithms**

- Describes the steps you must perform to complete a task
- Make assumptions (e.g. the data your using etc)
- Is not an implementation, instead it describes the steps you must perform.
- **There can be many algorithms that accomplish the same task**
- **There can be many implementations of the same algorithm**

## Big-O Notation

- Time complexity
    - Number of steps taken to run an algorithm
    - Look at the worst case most of the time
    - 

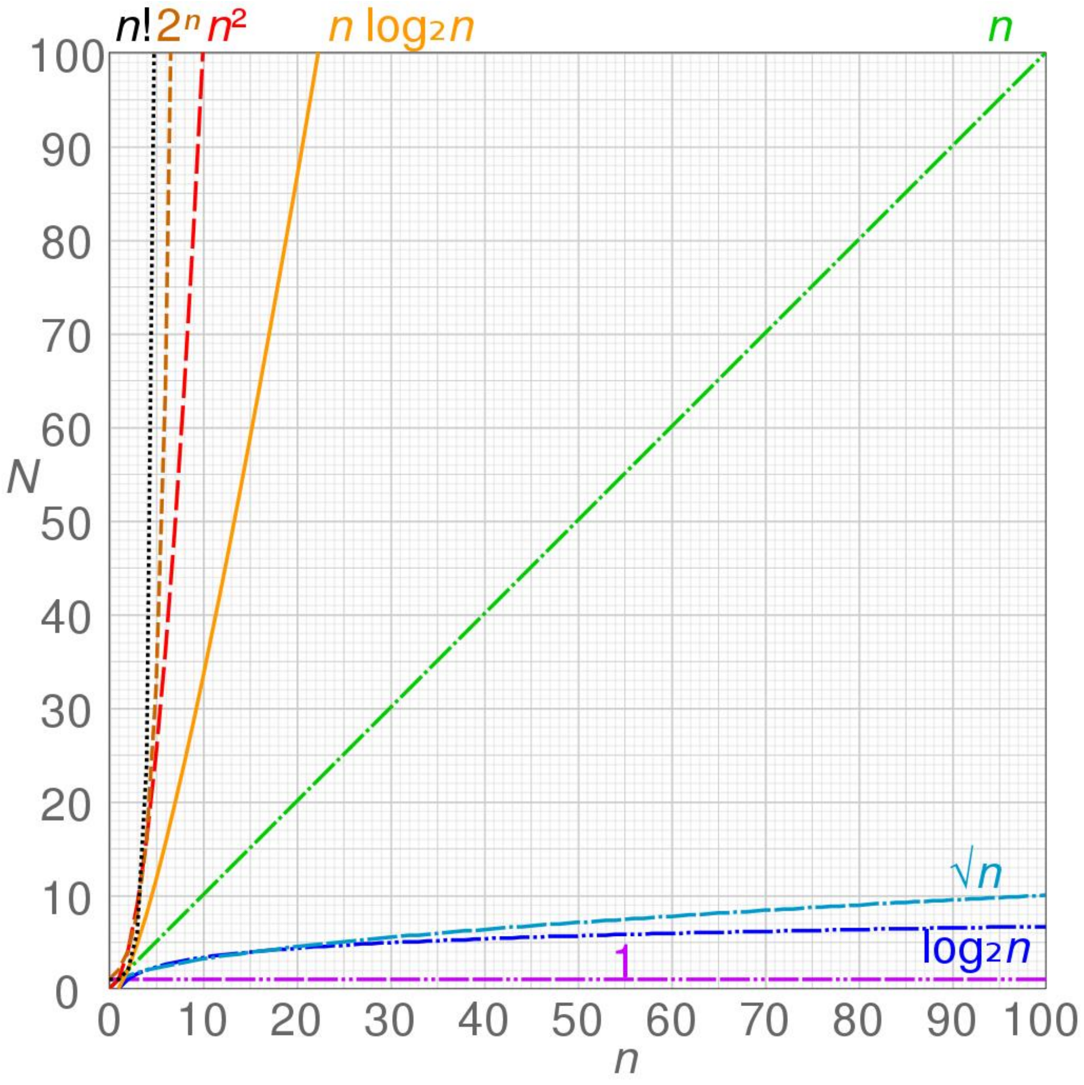
      Add Sugar to Tea

      1. Fetch the bowl containing the sugar
      2. Get a spoon
      3. Scoop out sugar using the spoon
      4. Pour the sugar from the spoon into the tea
      5. Repeat steps 3 and 4 until you've added the desired amount of sugar
    - 

      | Number of Sugars | Steps Required |
      |---|---|
      | 1 | 4 |
      | 2 | 6 |
      | 3 | 8 |
      | 4 | 10 |
    - Number of sugars = n
        - Total number of steps = 2n + 2
        - Time complexity = O(n)
        - Linear time complexity

| Big-O | |
|---|---|
| O(1) | Constant |
| O(logn) | Logarithmic |
| O(n) | Linear |
| O(nlogn) | n log-star n |
| $O(n^2)$ | Quadratic |

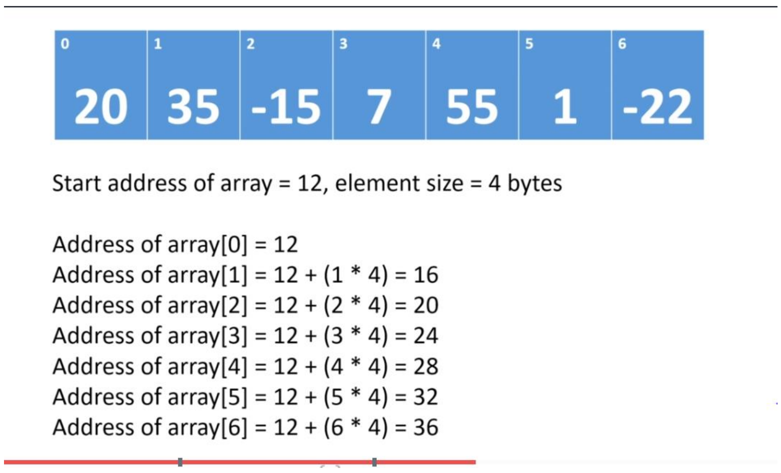
- Log to the base 2 -> log2N
- Order of table is best to worst.



- 
- O(1) obviously the best case
- Memory complexity
    - Not such an issue as memory is so cheap

## Arrays & Big-O

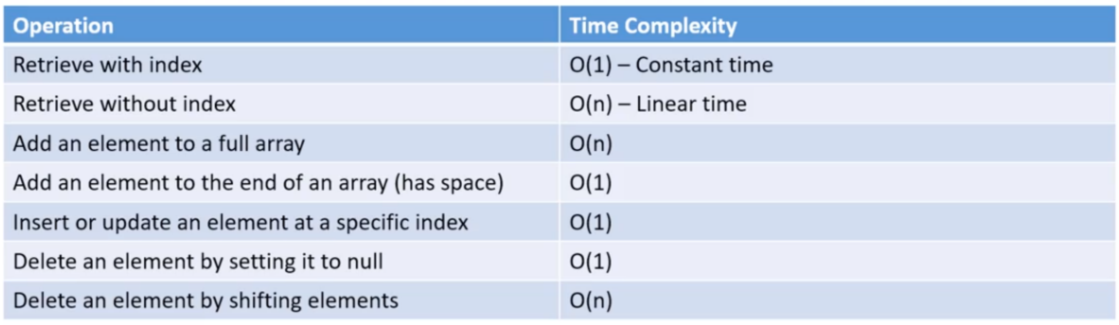
- Stored as one contiguous block in memory, not scattered throughout memory
- Every element occupies the same amount of space in memory
- If an array starts at memory address x, and the size of each element in the array is y, we can calculate the memory address of the ith element by using the following expression: x+i*y
- If we know the index of an element, the time to retrieve the element will be so the same, no matter where it is in the array



Start address of array = 12, element size = 4 bytes

Address of array[0] = 12
Address of array[1] = 12 + (1 * 4) = 16
Address of array[2] = 12 + (2 * 4) = 20
Address of array[3] = 12 + (3 * 4) = 24
Address of array[4] = 12 + (4 * 4) = 28
Address of array[5] = 12 + (5 * 4) = 32
Address of array[6] = 12 + (6 * 4) = 36

- Retrieve an element from an array
    - Multiply the size (bytes) of the element by its index
    - Get the start address of the array
    - Add the start address to the result of step 1
- Always 3 Steps
- O(1) time complexity in this case (Retrieving element)
- Easy way if it's a loop its linear, otherwise its constant



| Operation | Time Complexity |
|---|---|
| Retrieve with index | O(1) – Constant time |
| Retrieve without index | O(n) – Linear time |
| Add an element to a full array | O(n) |
| Add an element to the end of an array (has space) | O(1) |
| Insert or update an element at a specific index | O(1) |
| Delete an element by setting it to null | O(1) |
| Delete an element by shifting elements | O(n) |

## Sort Algorithms

### Bubble Sort

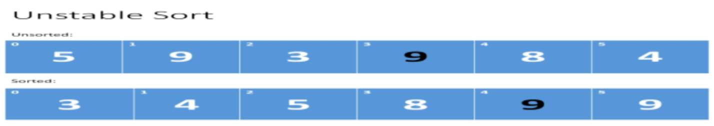
Sorts it from smallest to largest (larger values are bubbling up to the top)

- I = 0 -> index used to traverse the array from left to right
- unsortedPartition index = Length -1 (start)
- After the first traversal this index becomes (Length -2) etc…
- In-place algorithm
- O(n^2) time complexity – Quadratic (WORST CASE)
- It will take 100 steps to sort 10 items etc
- Algorithm degrades quickly

When it comes to determine the time complexity of an algorithm, we're not doing math, we want a general idea

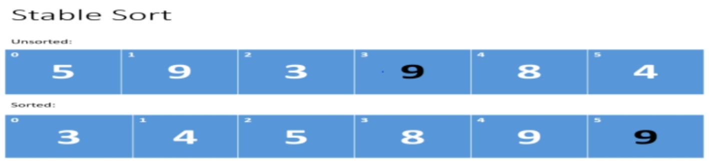
## Unstable Sort

- Relative ordering of equal integers is not preserved, which means its considered unstable.



## Stable Sort

- Preferred to an unstable sort.

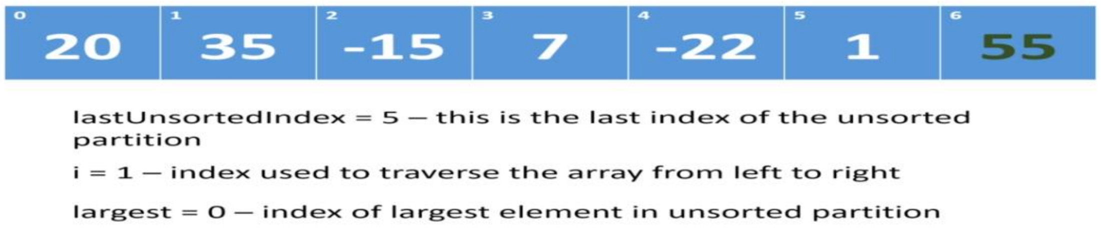


Might be an issue with sorting objects, For example if you want to do a sort by name then age it might affect the data. You don't want a sort to change the duplicating order of items.

- Bubble sort is stable! Only swap if (i > i+1)

## Selection Sort

- I = 1 – index used to traverse the array from left to right
- unsortedPartition index = Length -1 (start)
- After the first traversal this index becomes (Length -2) etc…
- Largest = 0 – index of largest element

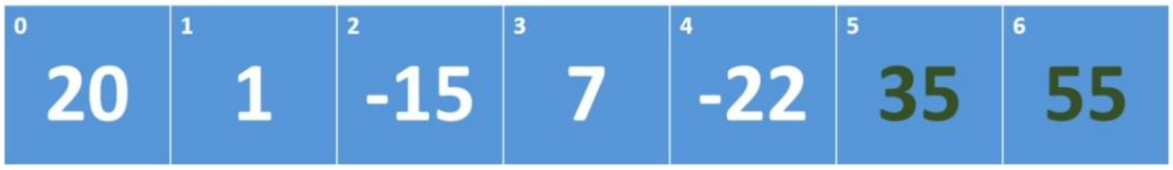


lastUnsortedIndex = 5 – this is the last index of the unsorted partition

i = 1 – index used to traverse the array from left to right

largest = 0 – index of largest element in unsorted partition

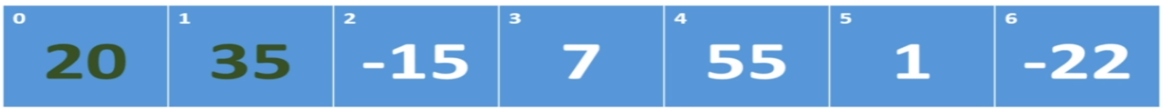
- 
- In this iteration largest would equal = 1
- And 35 will swap with 1



- 
- O(n^2) time complexity – Quadratic (WORST CASE)
- Doesn't requires as much swapping as bubblesort but is unstable.
- In the average case it will perform better than bubblesort

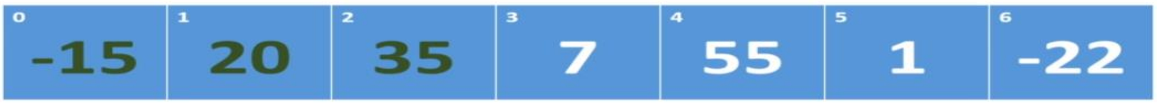
## Insertion Sort

- Grows the partition from the front of the array
- firstUnsortedIndex = 1;
- I = 0 – used to traverse from left to right
- newElement = would equal 35 which was want to insert into the sorted partion.
- First Iteration



| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 20 | 35 | -15 | 7 | 55 | 1 | -22 |

-
- Second iteration

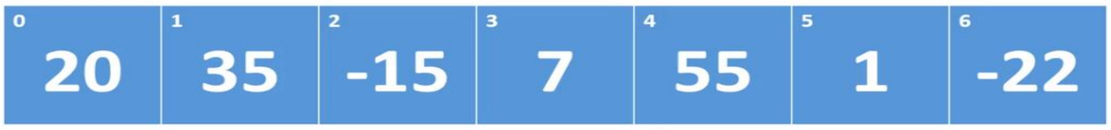


| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| -15 | 20 | 35 | 7 | 55 | 1 | -22 |

-
- 35 > -15 so we swap them, don't worry if its overridden as its stored in new element.
- 20> -15 so we swap them, we know its now at index 0 so we replace it there.
- O(n^2) time complexity – Quadratic (WORST CASE)
- Can be improved with Shell Sort

## Shell Sort

- Variation of insertion sort
- Insertion sort chooses which element to insert using a gap of 1
- Shell Sort starts out using a large gap value
- As the algorithm runs, the gap is reduced.
- **GOAL IS TO RECUDE Amount of shifting required**
  - The last gap is always 1, last iteration of the gap value is an insertion sort
  - By the time we get to insertion sore, the array has been partially sorted so there's left shifting required
-
- For our gap we'll base it on the array length /2
- Gap value gets divided by 2 on each iteration

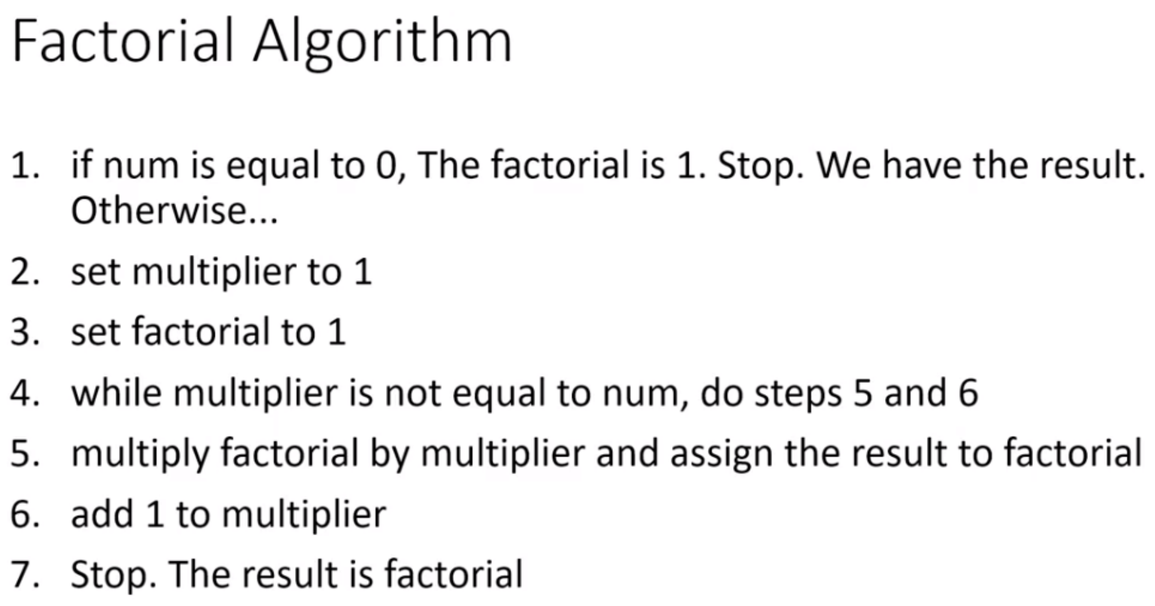


| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 20 | 35 | -15 | 7 | 55 | 1 | -22 |

-
- I = gap = 3
- J = I - gap
- New element = intArray[i]
- Compare intArray[j-gap] with newElement  = 7 < 20 – swap.
- When we hit the end of the array gap = gap/2
- Difficult to nail down time complexity
  - Worst Case = O(n^2)
  - Usually performs better than insertion sort but is unstable

## Recursion

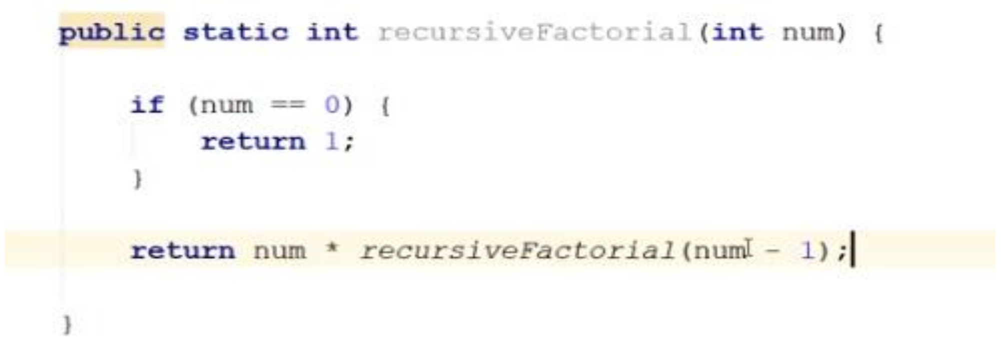
- A method is recursive when it calls itself when a result is met.



## Factorial Algorithm

1. if num is equal to 0, The factorial is 1. Stop. We have the result. Otherwise...
2. set multiplier to 1
3. set factorial to 1
4. while multiplier is not equal to num, do steps 5 and 6
5. multiply factorial by multiplier and assign the result to factorial
6. add 1 to multiplier
7. Stop. The result is factorial

-



```java
public static int recursiveFactorial(int num) {

    if (num == 0) {
        return 1;
    }

    return num * recursiveFactorial(num - 1);

}
```

Creates a "call Stack" e.g. if num was equal to 3

1st waits for recursive(2) which waits for recursive(1) which waits for recursive(0)

1. 1*1 = 1
2. 2*1 = 2
3. 3*2 = 6

Three-line method could result it hundreds of calls, if you didn't add a return condition it would never end.

Iterative solution is normally more efficient, developers normally still use recursion as its more elegant

Call stack = recursion stack

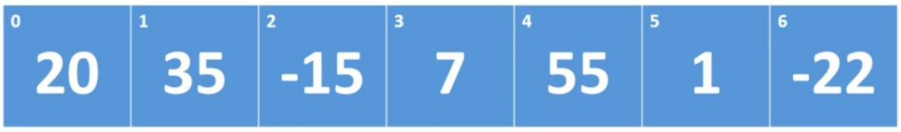
Stack overflow might not get hit if you call the recursive method invoke itself lots of times.

Can get over this issue using tail recursion ( Java compiler does not use this)

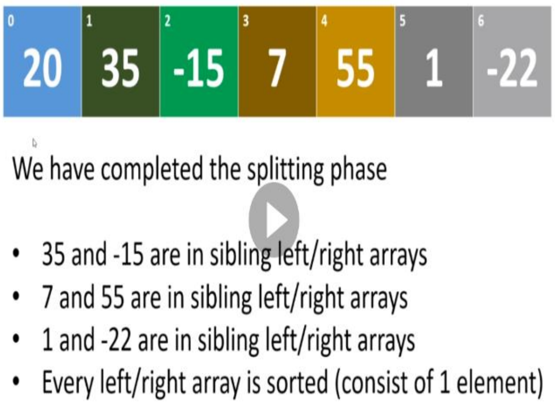
## Merge Sort

- Divide and conquer algorithm
- Recursive algorithm
- Two Phases: Splitting and Merging
- Splitting phase leads to faster sorting during the Merging phase
- Splitting is logical: We don't create new arrays
    - Start with an unsorted array
    - Divide array into two arrays
    - Split left and right array, keep splitting until each array only has one element
    - (these arrays are now sorted by default)
- Merging phase
    - Merge every left/right pair of sibling array into a sorted array
    - After the first merge we'll have a bunch of sorted 2-element arrays
    - Then merge those left right arrays (4-element arrays)
    - Repeat until you have a single sorted array.
    - Merging phase does not happen in place, uses temporary arrays

**How it works SPLITTING**

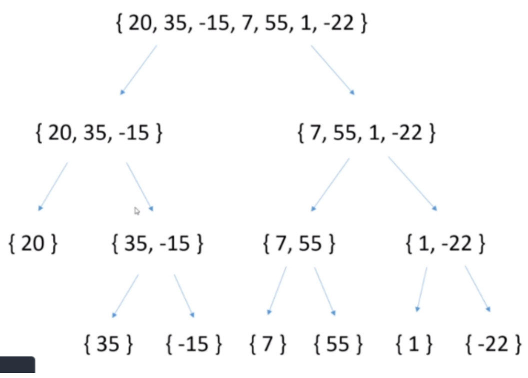


- Start = 0, end = 7 : Midpoint = 0+7/2
- Elements 0 to 2 go to the left array, rest to the right etc..



We have completed the splitting phase

- 35 and -15 are in sibling left/right arrays
- 7 and 55 are in sibling left/right arrays
- 1 and -22 are in sibling left/right arrays
- Every left/right array is sorted (consist of 1 element)



**How it works MERGING**

- We merge sibling left and right arrays
- We create a temporary array large enough to hold all the elements in the arrays we're merging. We set I to the first index of the left array and j to the first index of the right array
- We compare left[i] to right[j]. If left is smaller, we copy it to the temp array and increment I by 1. Same applies if J is bigger. Repeat until done.

- We repeat this process until all elements in the two arrays have been processed
- At this point, the temporary array contains the merged values in sorted order
- We then copy this temporary array back to the original input array, at the correct positions
- If the left array is at positions x to y and the right array is at positions y+1 to z, then after copy positions x to z will be sorted in the orginal array.

**Example**

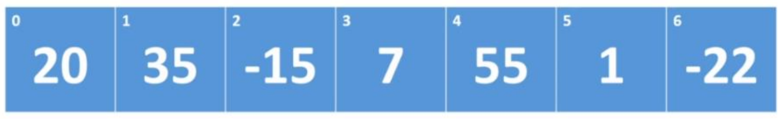
1. Start by merging the two siblings on the left (35 and -15)
2. -15 is smaller than 35 so it gets copied to the temp array
3. I will be initialised to 1, j to 2
4. Temp array = (-15,35)
5. We now copy these back into the original array
6. We've gone from 3 arrays on the left to two and both are sorted.
7. Next, we merge these two as they are no siblings
8. Final step is to merge the last two final arrays

## Divide and Conquering!!

- **Not an in place algorithm**
- **Time complexity of O(nlogn) base 2.**
- **Stable algorithm**

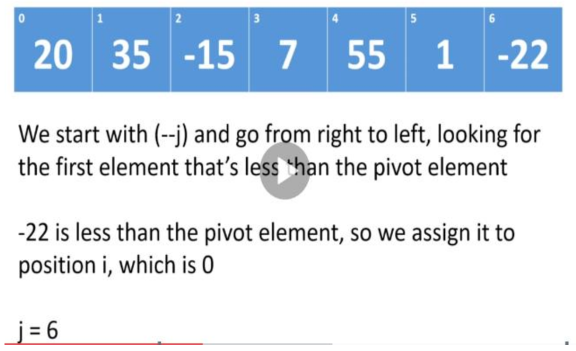
QuickSort

- Divide and conquer algorithm
- Recursive algorithm
- Uses a pivot element to partition the array into two parts
- Elements < pivot to the left, elements > to the right
- Pivot will then be in its correct sorted position
- Process is now repeated for left and right
- Eventually every element has been a pivot element
- Eventually it ends up as a 1 element array.
- **Does this in place**



Pivot will be the first element in the array (or subarray)

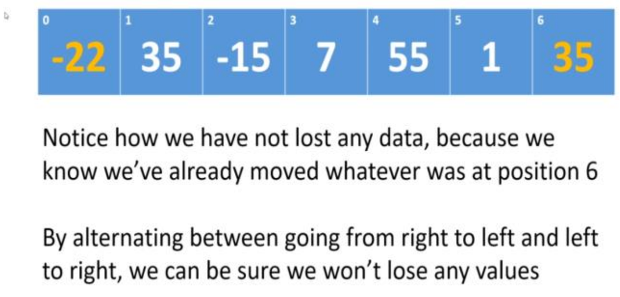
start = 0, i = start
end = 7, j = end
pivot = 20  (array[start])



We start with (--j) and go from right to left, looking for the first element that's less than the pivot element

-22 is less than the pivot element, so we assign it to position i, which is 0

j = 6



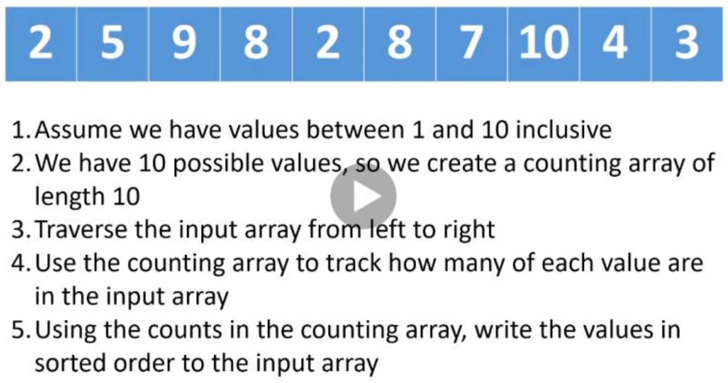
Notice how we have not lost any data, because we know we've already moved whatever was at position 6

By alternating between going from right to left and left to right, we can be sure we won't lose any values

- **In-place algorithm**
- **O(nlogn) base – 2**
- **Worst case is o(n^2)**
- **Most of the time it performs better than merge sort**
- **Unstable algorithm**

## Counting Sort

- Makes assumptions about the data
- Doesn’t use comparisons
- Counts the number of occurrences of each value
- Only works with non-negative discrete values
- Values must be within a specific range

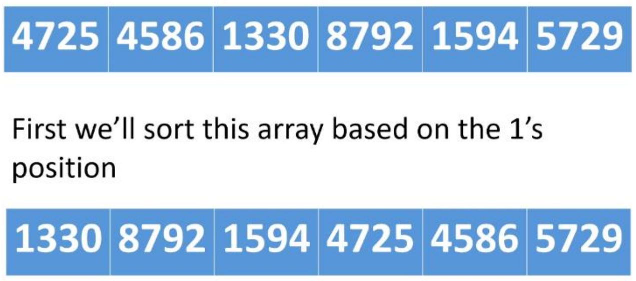


| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 5 | 9 | 8 | 2 | 8 | 7 | 10 | 4 | 3 |

1. Assume we have values between 1 and 10 inclusive
2. We have 10 possible values, so we create a counting array of length 10
3. Traverse the input array from left to right
4. Use the counting array to track how many of each value are in the input array
5. Using the counts in the counting array, write the values in sorted order to the input array

- Not an in-place algorithm
- O(n) – can achieve this because we’re making assumptions about the data
- If we want the sort to be stable, we have to perform some extra steps

## Radix Sort

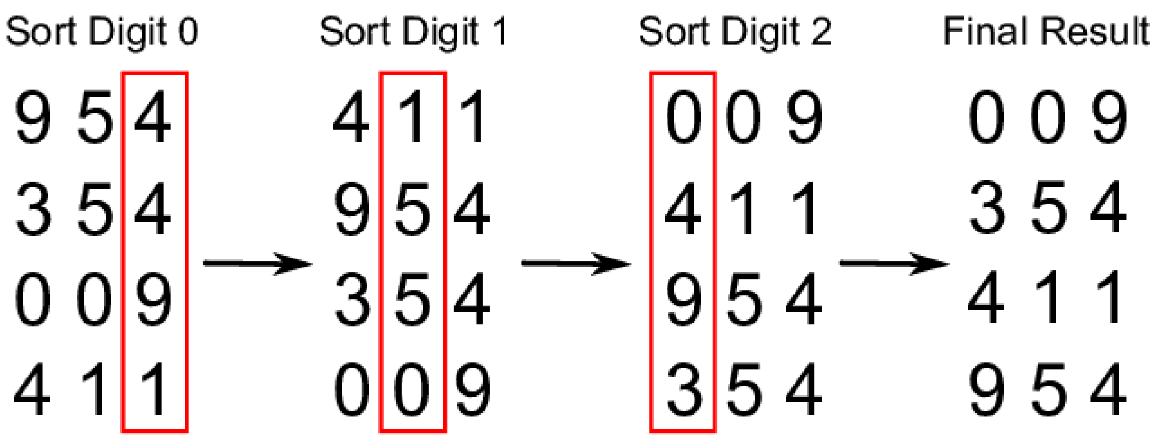
- Makes assumptions about the data
- Must have the same radix and width
- the radix or base is the number of unique digits, including the digit zero, used to represent numbers. For example, for the decimal/denary system the radix is ten, because it uses the ten digits from 0 through 9
- The data must be integers or strings
- Sort based on each individual digit or letter position
- Must use a stable sort algorithm at each stage
- Sort on the right most digit
- 

| | | | | | |
|---|---|---|---|---|---|
| 4725 | 4586 | 1330 | 8792 | 1594 | 5729 |

First we'll sort this array based on the 1's position

| | | | | | |
|---|---|---|---|---|---|
| 1330 | 8792 | 1594 | 4725 | 4586 | 5729 |

- Decrement this and move along
- Counting sort is often used as the sort algorithm for radix sort
- O(n) – can achieve this because we're making assumptions about the data
- Even so often runs slower than O(nLogn)
- In-place depends on what sort algorithm you use



-

## Abstract Data Type

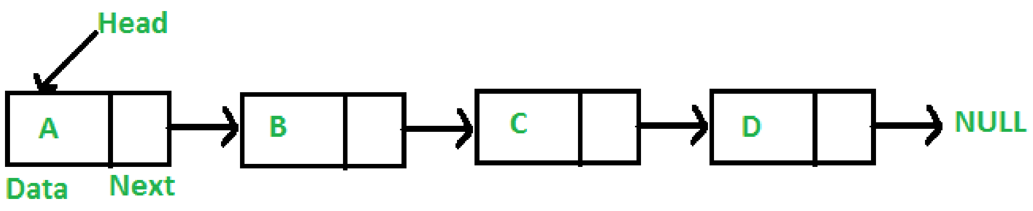
- Doesn't dictate how the data is organised
- Dictates the operations you can perform
- Concrete data structure is usually a concrete class
- Abstract data type is usually an interface

## Vectors

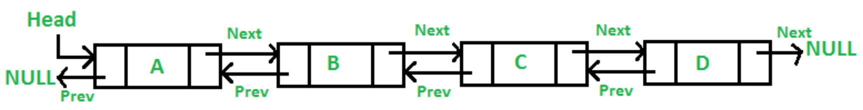
- Vectors a thread safe (Synchronised)
- If you use array lists with multiple threads you could run into running conflicts

## Singly Linked Lists

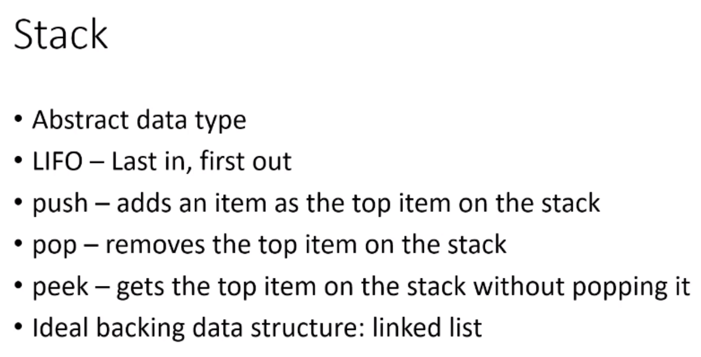
- In computer science, a linked list is a linear collection of data elements whose order is not given by their physical placement in memory. Instead, each element points to the next. It is a data structure consisting of a collection of nodes which together represent a sequence.



## Doubly linked lists

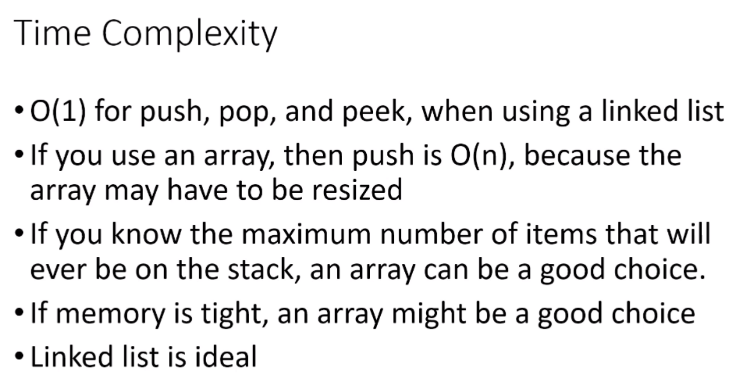


## Stacks



### Stack

- Abstract data type
- LIFO – Last in, first out
- push – adds an item as the top item on the stack
- pop – removes the top item on the stack
- peek – gets the top item on the stack without popping it
- Ideal backing data structure: linked list



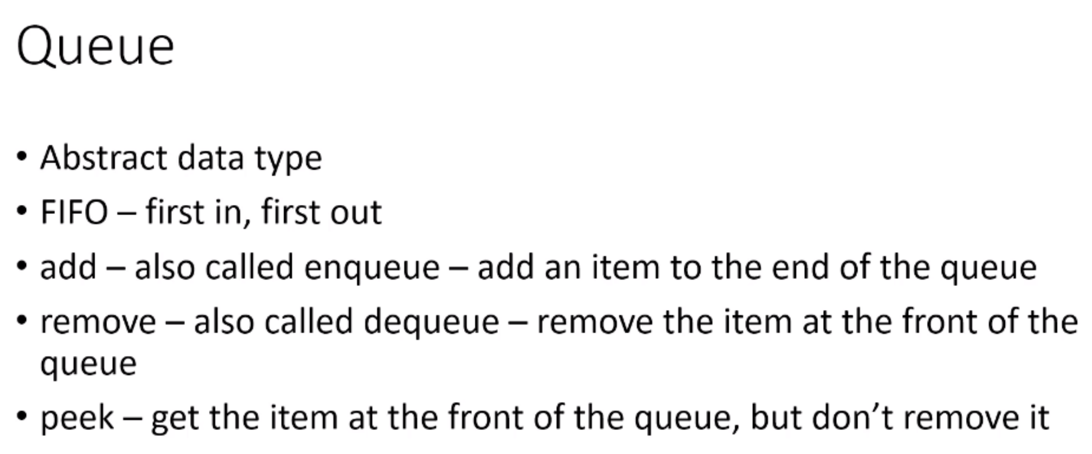
### Time Complexity

- O(1) for push, pop, and peek, when using a linked list
- If you use an array, then push is O(n), because the array may have to be resized
- If you know the maximum number of items that will ever be on the stack, an array can be a good choice.
- If memory is tight, an array might be a good choice
- Linked list is ideal

**Read In values from stdin**

- Scanner scanner = new Scanner(System.in);
- String myString = "";
- myString += scanner.nextLine();
- .nextInt();

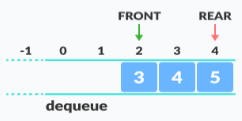
## Queues



### Queue

- Abstract data type
- FIFO – first in, first out
- add – also called enqueue – add an item to the end of the queue
- remove – also called dequeue – remove the item at the front of the queue
- peek – get the item at the front of the queue, but don’t remove it

Circular queue avoids the wastage of space in a regular queue implementation using arrays.



Limitation of the regular Queue

As you can see in the above image, after a bit of enqueuing and dequeuing, the size of the queue has been reduced.

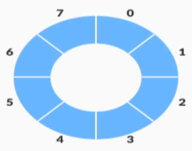
The indexes 0 and 1 can only be used after the queue is reset when all the elements have been dequeued.

## How Circular Queue Works

Circular Queue works by the process of circular increment i.e. when we try to increment the pointer and we reach the end of the queue, we start from the beginning of the queue.

Here, the circular increment is performed by modulo division with the queue size. That is,

```
if REAR + 1 == 5 (overflow!), REAR = (REAR + 1)%5 = 0 (start of queue)
```



Circular queue representation

Circular Queue Operations

The circular queue work as follows:

- two pointers FRONT and REAR
- FRONT track the first element of the queue
- REAR track the last elements of the queue
- initially, set value of FRONT and REAR to -1

1. Enqueue Operation

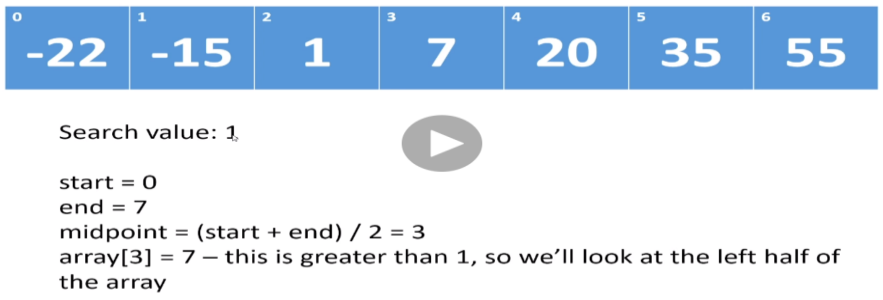
- check if the queue is full
- for the first element, set value of FRONT to 0
- circularly increase the REAR index by 1 (i.e. if the rear reaches the end, next it would be at the start of the queue)
- add the new element in the position pointed to by REAR

2. Dequeue Operation

- check if the queue is empty
- return the value pointed by FRONT
- circularly increase the FRONT index by 1
- for the last element, reset the values of FRONT and REAR to -1

## Binary Search

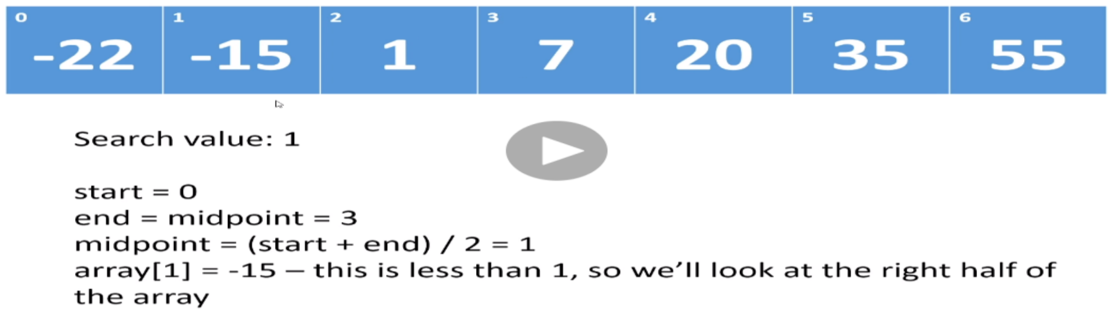
- **Data Must be sorted!!**
- **Chooses the element in the middle of the array and compares it to the search value**
- **If Element is equal to the value we're done**
- **If element is greater than the value, search the left half of the array**
- **If element is less than the value, search the right half of the array**



| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| -22 | -15 | 1 | 7 | 20 | 35 | 55 |

Search value: 1

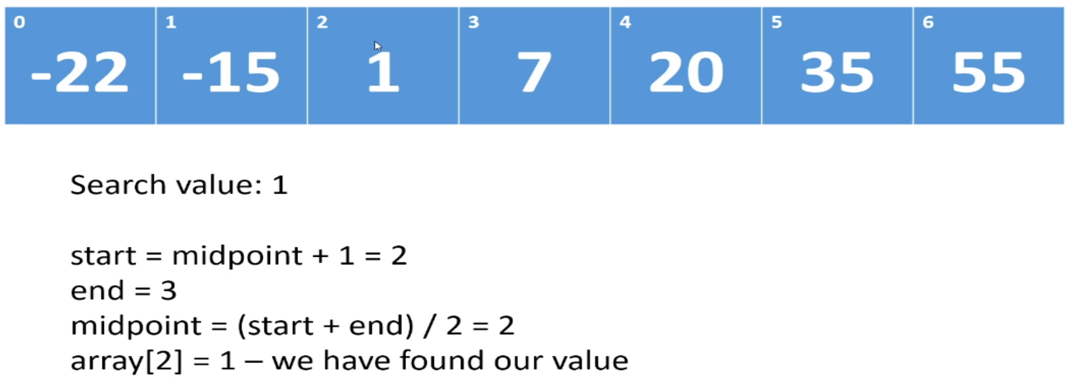
start = 0
end = 7
midpoint = (start + end) / 2 = 3
array[3] = 7 – this is greater than 1, so we'll look at the left half of the array



| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| -22 | -15 | 1 | 7 | 20 | 35 | 55 |

Search value: 1

start = 0
end = midpoint = 3
midpoint = (start + end) / 2 = 1
array[1] = -15 – this is less than 1, so we'll look at the right half of the array



| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| -22 | -15 | 1 | 7 | 20 | 35 | 55 |

Search value: 1

start = midpoint + 1 = 2
end = 3
midpoint = (start + end) / 2 = 2
array[2] = 1 – we have found our value

- At some point, there will be only one element in the partition you're checking, but it doesn't have to get to that point
- Can be implemented recursively
- O(logn)
-
```
public static int iterativeBinarySearch(int[] input, int value){
    int start = 0;
    int end = input.length;

    while(start < end){
        int midpoint = (start + end)/2;
        if(input[midpoint] == value){
            return midpoint;
            }
            else if(input[midpoint] < value){
                start = midpoint + 1;
            }
            else {
                end = midpoint;
            }
        }
        return -1;
    }
```

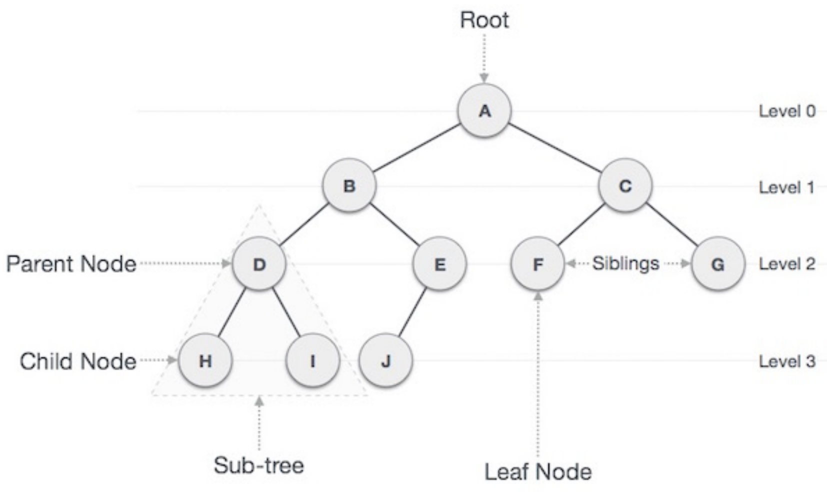
```
public static int recursiveBinarySearch(int[] input, int value){
    return recursiveBinarySearch(input,0,input.length,value);
}

public static int recursiveBinarySearch(int[] input, int start, int end, int value){
    if(start >= end){
        return -1;
    }
    int midpoint = (start + end)/2;
    if(input[midpoint] == value){
        return midpoint;
    }
    else if(input[midpoint] < value){
        return recursiveBinarySearch(input,midpoint+1,end,value);
    }
    else {
        return recursiveBinarySearch(input,start,midpoint,value);
    }
}
```

Recursive vs iteratively

# TREES

Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically.

Binary Tree is a special datastructure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.



**Important Terms**

- **Root –** Refers to the sequence of nodes along the edges of a tree
- **Path** – The node at the top of the tree. There is only one root per tree and one path from the root to any node
- **Parent –** Any node except the root node has one edge upwards to a parent node
- **Child –** The node below any given node is known as the child node
- **Leaf –** The node which does not have any child nodes
- **Subtree** – subtree represents the descendants of a node
- **Visiting** – Visiting refers to checking the value of anode when control is on the node
- **Traversing –** Traversing means passing through nodes in a specific order
- **Levels –** Level of a node represents a generation
- **Keys -** Key represents a value of a node based on which search operation is to be carried out for a node

## BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following –

- **Insert** – Inserts an element in a tree/create a tree.
- **Search** – Searches an element in a tree.
- **Preorder Traversal** – Traverses a tree in a pre-order manner.
- **Inorder Traversal** – Traverses a tree in an in-order manner.
- **Postorder Traversal** – Traverses a tree in a post-order manner.

## Binary Tree

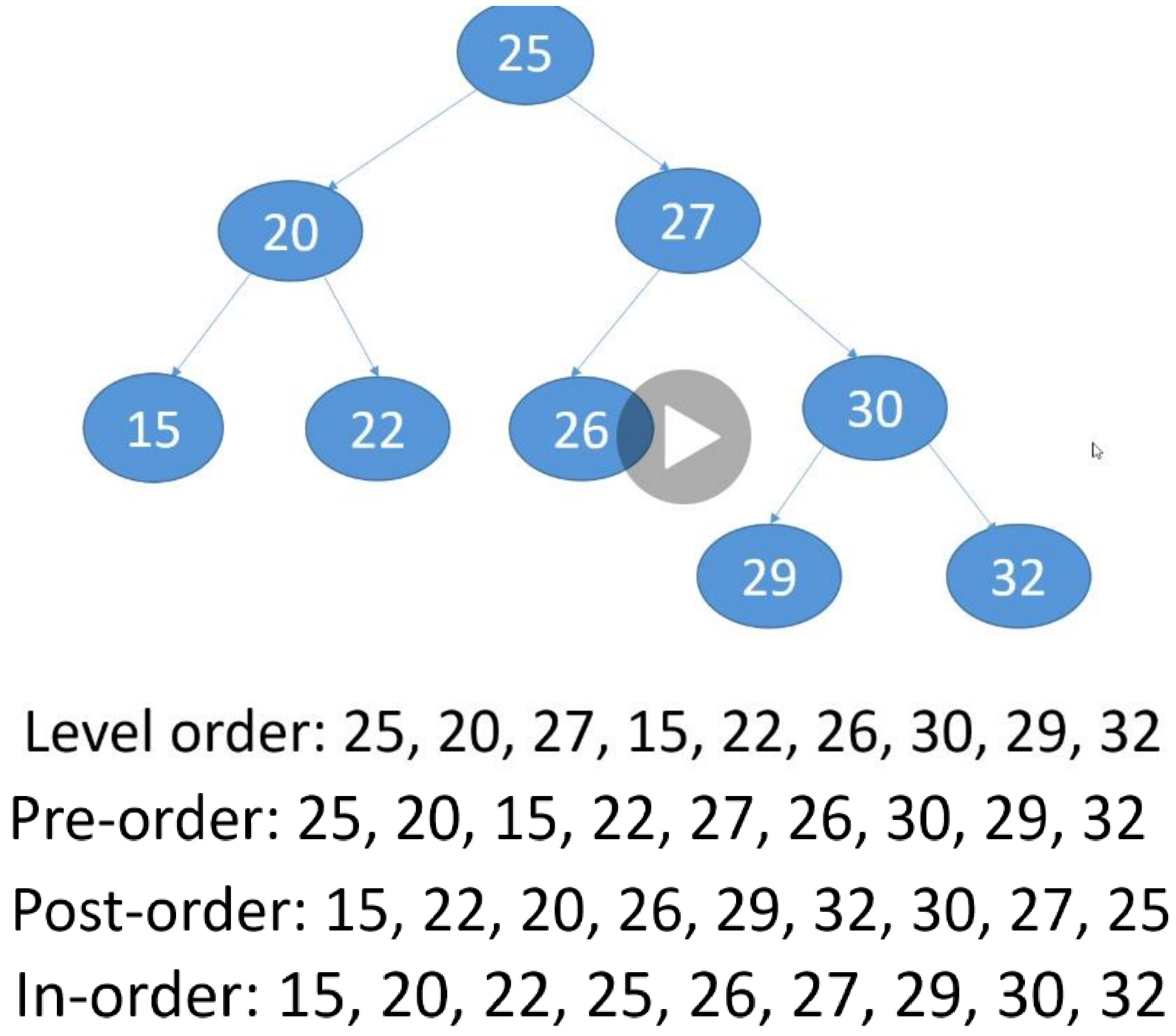
- **Every** node has 0,1 or 2 children
- Children are referred to as the left and right child
- In practice, we use binary search trees
- Complete tree
    - Every level except the last node must be filled
    - And must be added to the left side first
    - Full binary tree means every node must have 2 children if they are not a leaf node
- **IT is completely fine to have incomplete tree**

### Binary Search Trees

- Can perform insertions, deletions, and retrievals in O(logn) time
- The left child always has a smaller value than its parent
- The right child always has a larger value than its parent
- This means everything to the left to root is less than the value of the right, and everything on the right of the root is greater than the value of the root
- Because of that, we can do a binary search
- Not worth inserting a list that's already sorted
    - Would essentially create a linked list
    - Try to keep it as balanced as possible

#### Traversal

- Level – visit nodes on each level
- Pre-order – visit the root of every subtree first
- Post order – visit the root of every subtree last
- In-order- visit the left child, then root then right child.



Level order: 25, 20, 27, 15, 22, 26, 30, 29, 32

Pre-order: 25, 20, 15, 22, 27, 26, 30, 29, 32

Post-order: 15, 22, 20, 26, 29, 32, 30, 27, 25

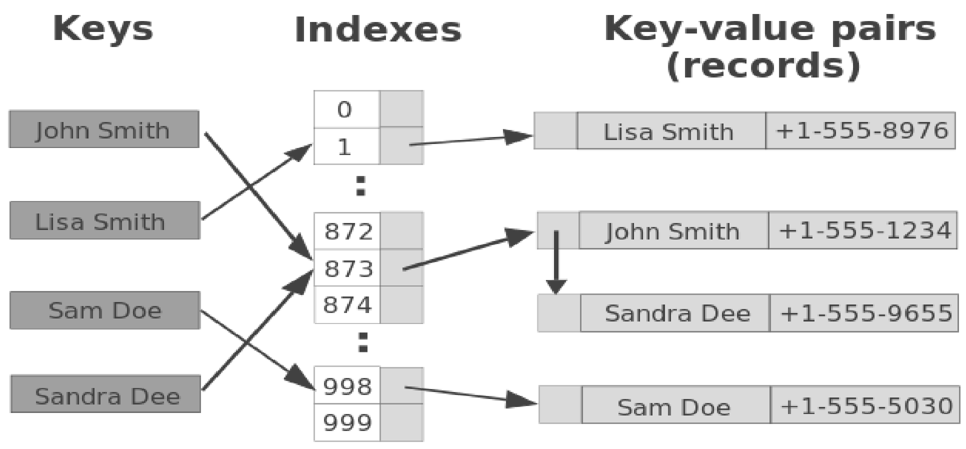
In-order: 15, 20, 22, 25, 26, 27, 29, 30, 32

Delete Has 3 Possibilities

- Node is a leaf
- Node has one child
- Node has two children
    - Need to figure out what the replacement node will be
    - Want minimal disruption to existing tree
    - Can take the replacement node from the deleted nodes left subtree or right subtree
    - If taking it from the left subtree, we have to take the largest value in the left subtree
    - If taking it from the right subtree, we have to take the smallest value in the right subtree
    - Choose one and stick to it
- Not many options in terms on JKD
    - TreeMap uses a red-black tree
    - Garuntees log(n) time – binary search tree

## Hash Tables

- A hash table is an unordered collection of key-value pairs, where each key is unique.
- Hash tables offer a combination of **lookup**, **insert** and **delete** operations. Neither arrays nor linked lists can achieve this:
  - a lookup in an unsorted array takes linear worst-case time;
  - in a sorted array, a lookup using binary search is very fast, but insertions become inefficient;
  - in a linked list an insertion can be efficient, but lookups take linear time.
- The most common hash table implementation uses chaining with linked lists to resolve collisions.
- This combines the best properties of arrays and linked lists.
- Hash table operations are performed in two steps:
  - A key is converted into an integer index by using a hash function.
  - This index decides the linked list where the key-value pair record belongs.



Inserting a new record (key, value) is a two-step procedure:

- we extract the three last digits of the key, hash = key % 1000,
- and then insert the key and its value into the list located at table[hash

The **average time complexity** of both the lookup and insert operations is *O*(1). Using the same technique, deletion can also be implemented in constant average time.

**<u>Realistic hash function example</u>**

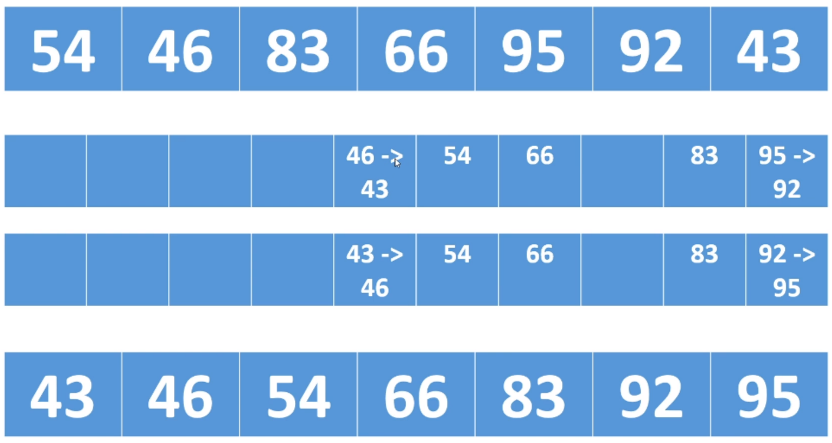
We want to generalize this basic idea to more complicated keys that aren't evenly distributed. The number of records in each list must remain small, and the records must be evenly distributed over the lists. To achieve this we just need to change the **hash function**, the function which selects the list where a key belongs.

You cannot add the same key for more than one value, it gets rewritten.

## Bucket Sort (Generalisation of counting sort)

- Uses hashing
- Makes assumptions about the data, like radix and counting sort
- Because it makes assumptions, can sort in O(n) time
- Performs best when hashed values of items being sorted are evenly distributed
- The values in bucket X must all be greater than the values in bucket x-1 and less than the values in bucket x+1

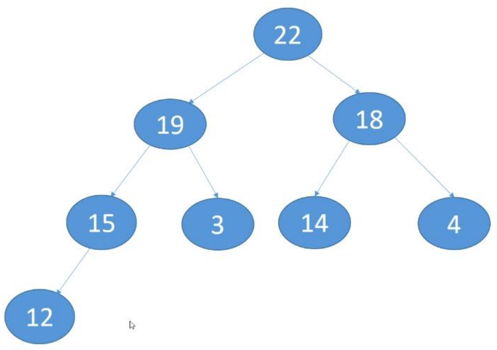
1. Distribute the items into buckets based on their hashed values
2. Sort the items in each bucket
3. Merge the buckers



- Example above hashing on the first int index, that is 46 and 43 will be in the same bucket etc.
- Then the bucket will organise in terms of smallest to largest (Sort), then join the buckets back together.
- Not in-place
- Stability will depend on sort algorithm used to sort the buckets, ideally, you want a stable sort
- To Achieve O(N), you must have only one item per bucket
- Insertion sort is often used to sort the buckets because its fast when the number of items is small

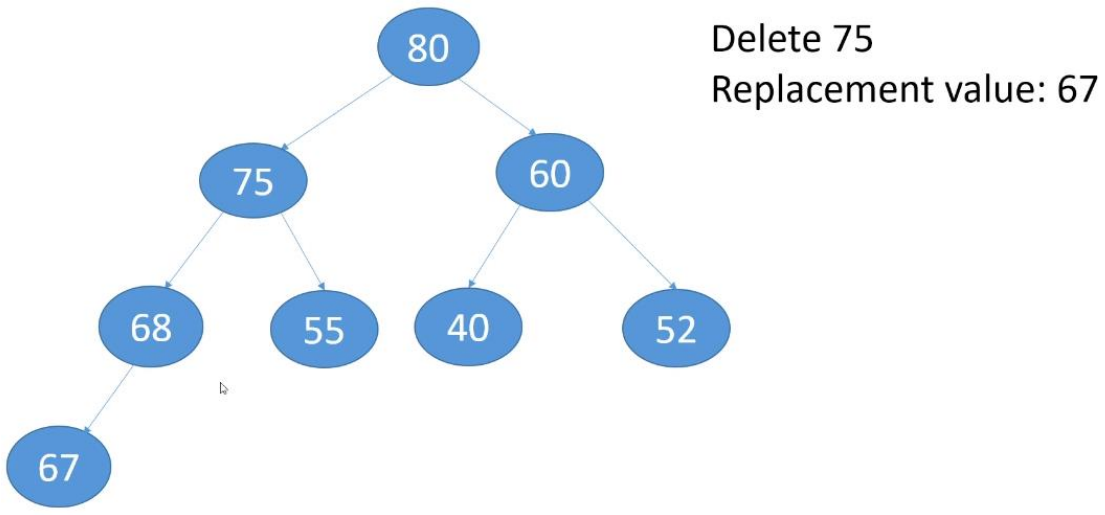
## Heaps

- A complete binary tree
- Must satisfy the heap property (Every level is full, except the last level potentially)
    - Existing leaves must be as far to the left as poss
- Max Heap: Every parent is greater than or equal to its children
- Min heap: Every parent is less than or equal to its children
- Usually implemented as arrays, children added from left to right



A heap!

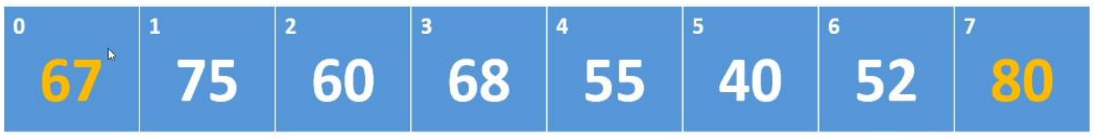
- Heaps as Arrays
    - We can store binary heaps as arrays
    - We put the root at array[o]
    - We then traverse each level from left to right, and so the left child of the root would go into array[1], its right into array[2] etc..
    - Above would be [22,19,18,15,3,14,4,12]
    - Left child = 2i +1
    - Right child = 2i+2
    - Parent = floor(i-1/2)
- Insert into heap
    - Always add new items to the end of the array
    - Then we must fix the heap(heapify)
    - We compare the new item against its parent, if the item is greater than its parent, we swap it (Rinse and repeat)
- Delete Theory
    - Must choose a replacement value
    - Will take the rightmost value, so that the tree remains complete
    - Then we must heapify the heap
    - When replacement value is greater than parent, fix heap above, otherwise fix heap below.
    - Fix heap above – same as insert, swap replacement value with parent
    - Fix heap below, swap replacement value with the larger of its two children
    - Rinse and repeat in both cases until in correct position
    - Only need to perform one of the two cases



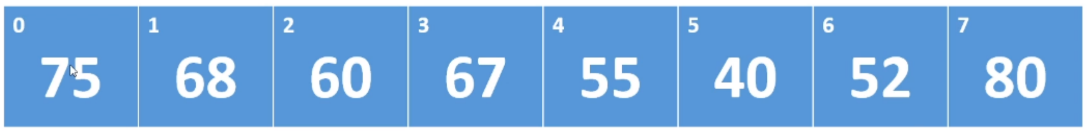
- Peek
    - We want to see what's at the root. Return index 0
- Priority Queues
    - When we take something of the queue, we always take the highest priority item next. Stored in the root item();

## HeapSort

- We know the root has the largest or smallest value
- Swap root with last element in the array
- Heapify the tree, but exclude the last node
    - After this the second largest element is at the root
    - Rinse and repeat



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 67 | 75 | 60 | 68 | 55 | 40 | 52 | 80 |



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 75 | 68 | 60 | 67 | 55 | 40 | 52 | 80 |

- Then 75 swaps with 52, repeat…

## Kruskal's Minimum Spanning Tree Algorithm

*What is Minimum Spanning Tree?*

Given a connected and undirected graph, a *spanning tree* of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A *minimum spanning tree (MST)* or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

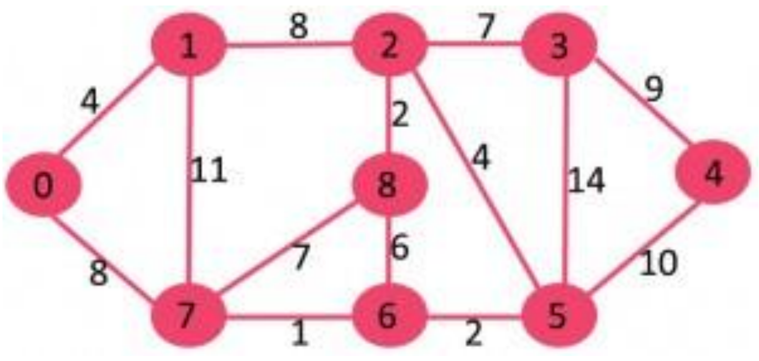
*How many edges does a minimum spanning tree has?*

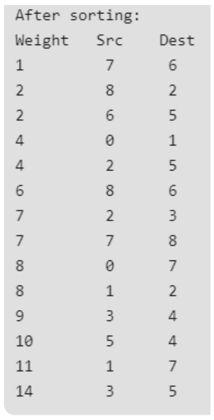
A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

*What are the applications of Minimum Spanning Tree?*

See this for applications of MST.

***1.*** *Sort all the edges in non-decreasing order of their weight.*
***2.*** *Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.*
***3.*** *Repeat step#2 until there are (V-1) edges in the spanning tree*



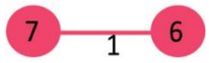

After sorting:

| Weight | Src | Dest |
|---|---|---|
| 1 | 7 | 6 |
| 2 | 8 | 2 |
| 2 | 6 | 5 |
| 4 | 0 | 1 |
| 4 | 2 | 5 |
| 6 | 8 | 6 |
| 7 | 2 | 3 |
| 7 | 7 | 8 |
| 8 | 0 | 7 |
| 8 | 1 | 2 |
| 9 | 3 | 4 |
| 10 | 5 | 4 |
| 11 | 1 | 7 |
| 14 | 3 | 5 |

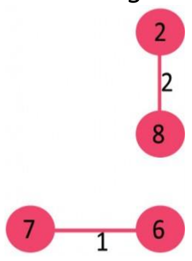
The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

Now pick all edges one by one from sorted list of edges

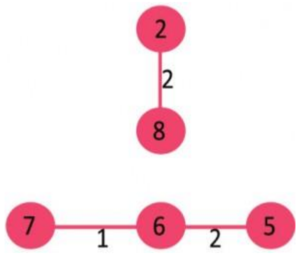
**1.** *Pick edge 7-6:* No cycle is formed, include it.



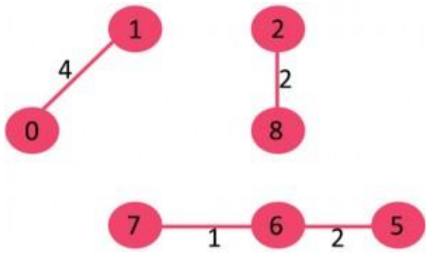
**2.** *Pick edge 8-2:* No cycle is formed, include it.



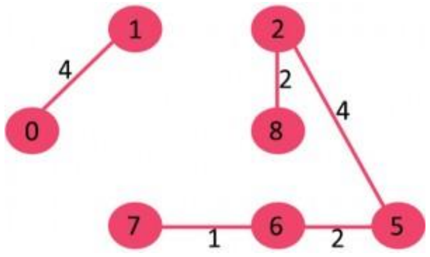
**3.** *Pick edge 6-5:* No cycle is formed, include it.



**4.** *Pick edge 0-1:* No cycle is formed, include it.

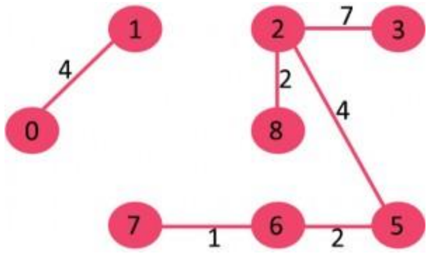


**5.** *Pick edge 2-5:* No cycle is formed, include it.



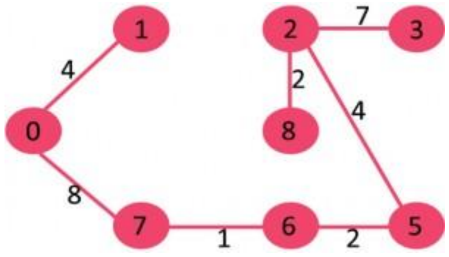
**6.** *Pick edge 8-6:* Since including this edge results in cycle, discard it.

**7.** *Pick edge 2-3:* No cycle is formed, include it.



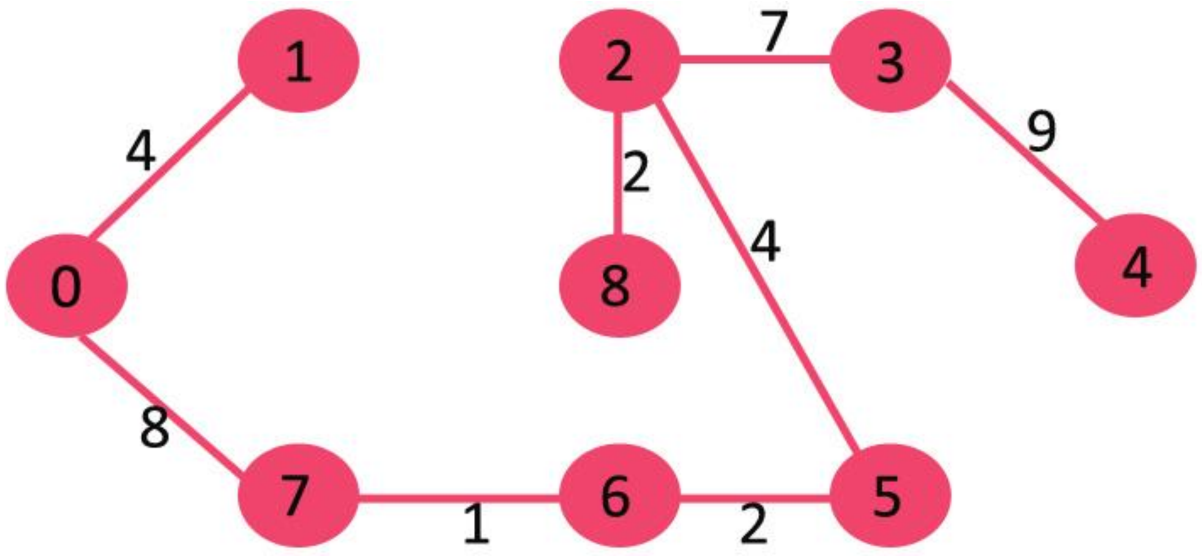
**8.** *Pick edge 7-8:* Since including this edge results in cycle, discard it.

**9.** *Pick edge 0-7:* No cycle is formed, include it.



**10.** *Pick edge 1-2:* Since including this edge results in cycle, discard it.

**11.** *Pick edge 3-4:* No cycle is formed, include it.



See GitHub for implementation in java

**Time Complexity:** O(ElogE) or O(ElogV).

Sorting of edges takes O(ELogE) time. After sorting, we iterate through all edges and apply find-union algorithm. The find and union operations can take atmost O(LogV) time. So overall complexity is O(ELogE + ELogV) time. The value of E can be atmost O($V^2$), so O(LogV) are O(LogE) same. Therefore, overall time complexity is O(ElogE) or O(ElogV)

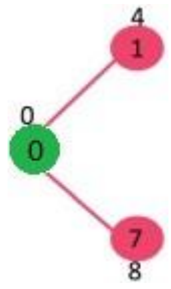
## Prim's Minimum Spanning Tree (MST)

Like Kruskal's algorithm, Prim's algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

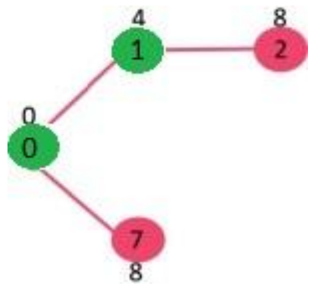
***Algorithm***
**1)** Create a set *mstSet* that keeps track of vertices already included in MST.
**2)** Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
**3)** While mstSet doesn't include all vertices
**a)** Pick a vertex *u* which is not there in *mstSet* and has minimum key value.
**b)** Include *u* to mstSet.
**c)** Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u*-*v* is less than the previous key value of *v*, update the key value as weight of *u*-*v*
The idea of using key values is to pick the minimum weight edge from cut. The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

## Using Same Example Tree

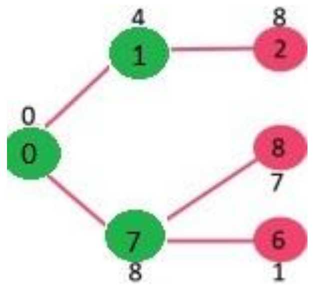
The set *mstSet* is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with the minimum key value. The vertex 0 is picked, include it in *mstSet*. So *mstSet* becomes {0}. After including to *mstSet*, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.

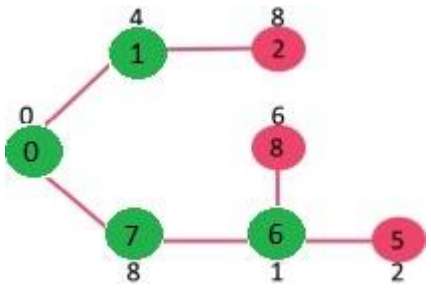


Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.



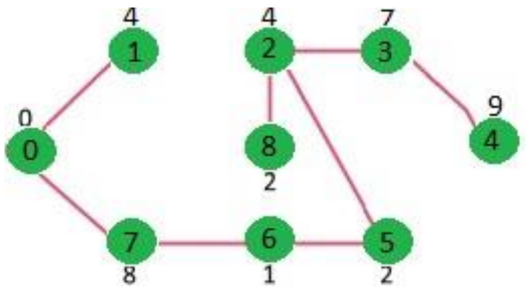
Pick the vertex with minimum key value and not already included in MST (not in mstSET). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So mstSet now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (1 and 7 respectively).





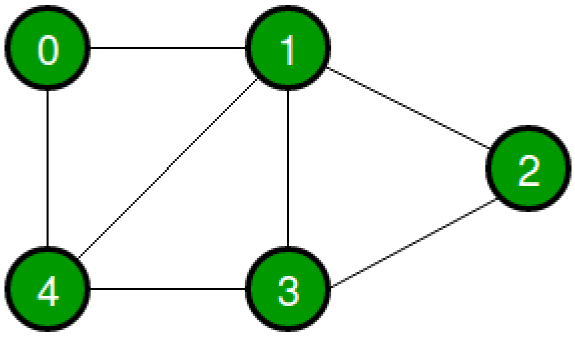
Pick the vertex with minimum key value and not already included in MST (not in mstSET). Vertex 6 is picked. So mstSet now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.

We repeat the above steps until *mstSet* includes all vertices of given graph. Finally, we get the following graph.



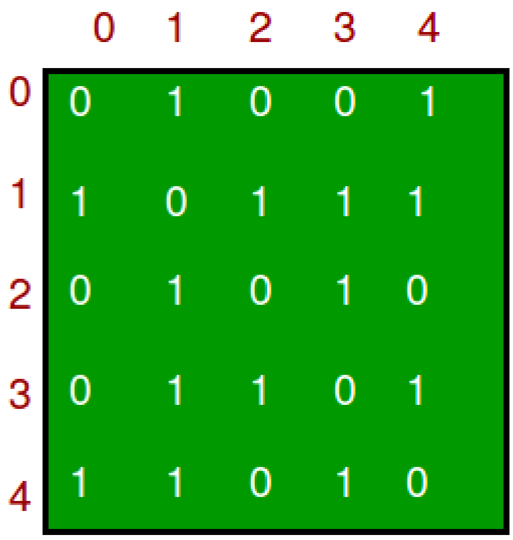
Time Complexity of the above program is O(V^2). If the input graph is represented using adjacency list, then the time complexity of Prim's algorithm can be reduced to O(E log V) with the help of binary heap.

## Graph and its representations



The following two are the most commonly used representations of a graph.

**1.** Adjacency Matrix

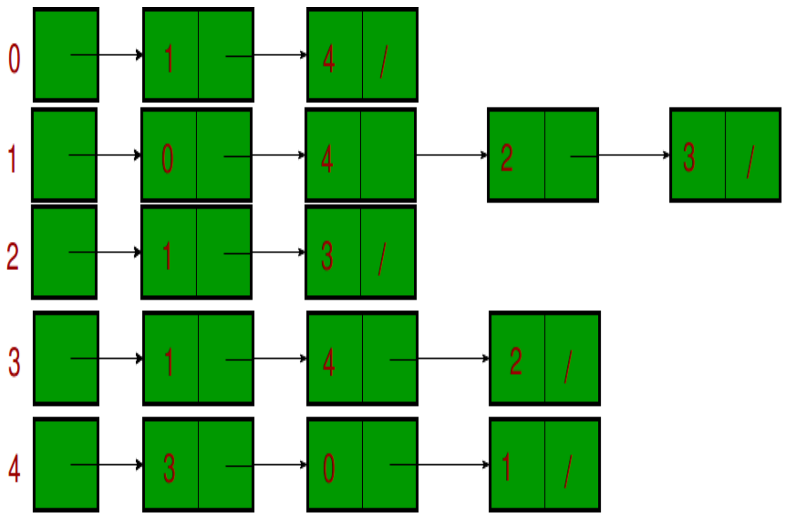


$$\begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 2 & 0 & 1 & 0 & 1 & 0 \\ 3 & 0 & 1 & 1 & 0 & 1 \\ 4 & 1 & 1 & 0 & 1 & 0 \end{array}$$

*Pros:* Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).

*Cons:* Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time. Please see this for a sample Python implementation of adjacency matrix.

**2.** Adjacency List



An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an array[]. An entry array[i] represents the list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above graph.

# Dijkstra’s shortest path algorithm

Dijkstra’s algorithm is very similar to Prim’s algorithm for minimum spanning tree. Like Prim’s MST, we generate a *SPT (shortest path tree)* with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

Algorithm

**1)** Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

**2)** Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

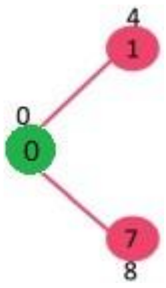
**3)** While *sptSet* doesn’t include all vertices

**a)** Pick a vertex u which is not there in *sptSet* and has minimum distance value.

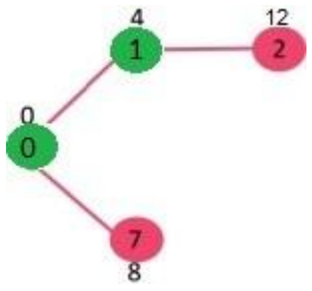
**b)** Include u to *sptSet*.

**c)** Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

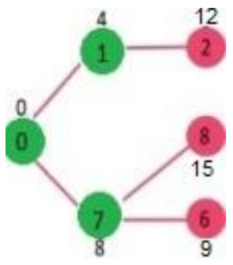
The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.

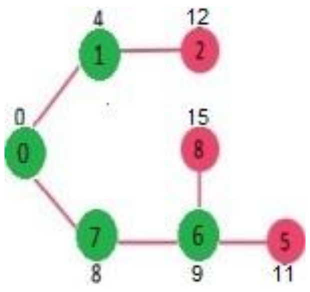


Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).





Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.

We repeat the above steps until *sptSet* does include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT

**Notes**

**1)** The code calculates shortest distance, but doesn't calculate the path information. We can create a parent array, update the parent array when distance is updated (like prim's implementation) and use it show the shortest path from source to different vertices.
**2)** The code is for undirected graph, same dijkstra function can be used for directed graphs also.
**3)** The code finds shortest distances from source to all vertices. If we are interested only in shortest distance from the source to a single target, we can break the for the loop when the picked minimum distance vertex is equal to target (Step 3.a of the algorithm).
**4)** Time Complexity of the implementation is O(V^2).
If the input graph is represented using adjacency list, it can be reduced to O(E log V) with the help of binary heap. Please see
Dijkstra's Algorithm for Adjacency List Representation for more details.
**5)** Dijkstra's algorithm doesn't work for graphs with negative weight cycles, it may give correct results for a graph with negative edges. For graphs with negative weight edges and cycles, Bellman–Ford algorithm can be used, we will soon be discussing it as a separate post.