BASICS OF ALGO & DATA STRUCTURES

BUBBLE SORT

Bubble sort algorithm starts by comparing the first two elements of an array and swapping if necessary, i.e., if you want to sort the elements of array in ascending order and if the first element is greater than second then, you need to swap the elements but, if the first element is smaller than second, you mustn't swap the element. Then, again second and third elements are compared and swapped if it is necessary and this process go on until last and second last element is compared and swapped. This completes the first step of bubble sort.

If there are n elements to be sorted then, the process mentioned above should be repeated n-1 times to get required result. But, for better performance, in second step, last and second last elements are not compared because, the proper element is automatically placed at last after first step. Similarly, in third step, last and second last and second last and third last elements are not compared and so on.

INSERTION SORT

1. Step 1: The second element of an array is compared with the elements that appears before it (only first element in this case). If the second element is smaller than first element, second element is inserted in the position of first element. After first step, first two elements of an array will be sorted.

2. Step 2: The third element of an array is compared with the elements that appears before it (first and second element). If third element is smaller than first element, it is inserted in the position of first element. If third element is larger than first element but, smaller than second element, it is inserted in the position of second element. If third element is larger than both the elements, it is kept in the position as it is. After second step, first three elements of an array will be sorted.

3. Step 3: Similary, the fourth element of an array is compared with the elements that appears before it (first, second and third element) and the same procedure is applied

and that element is inserted in the proper position. After third step, first four elements of an array will be sorted.

SELECTION SORT

SELECT VALUE AT FIRST INDEX AS MIN VALUE OR FIRST INDEX AS MIN INDEX, COMPARE , SWAP AND UPDATE MIN VALUE AT EXCH ITERATION. TOTAL n-1 ITERATION.

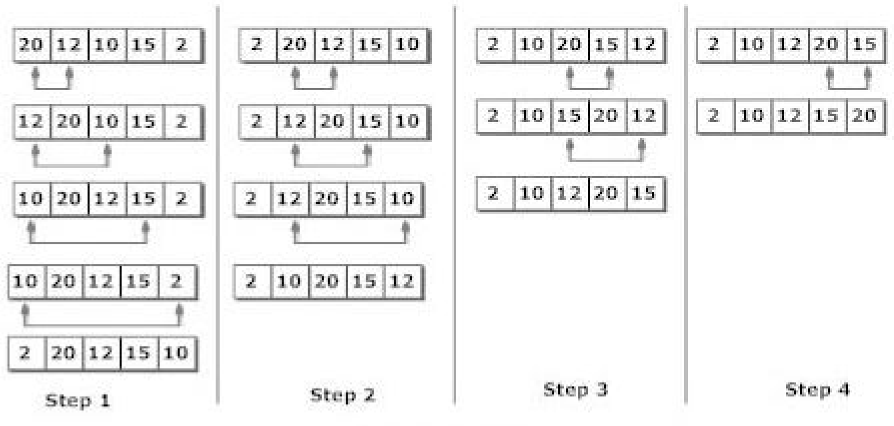


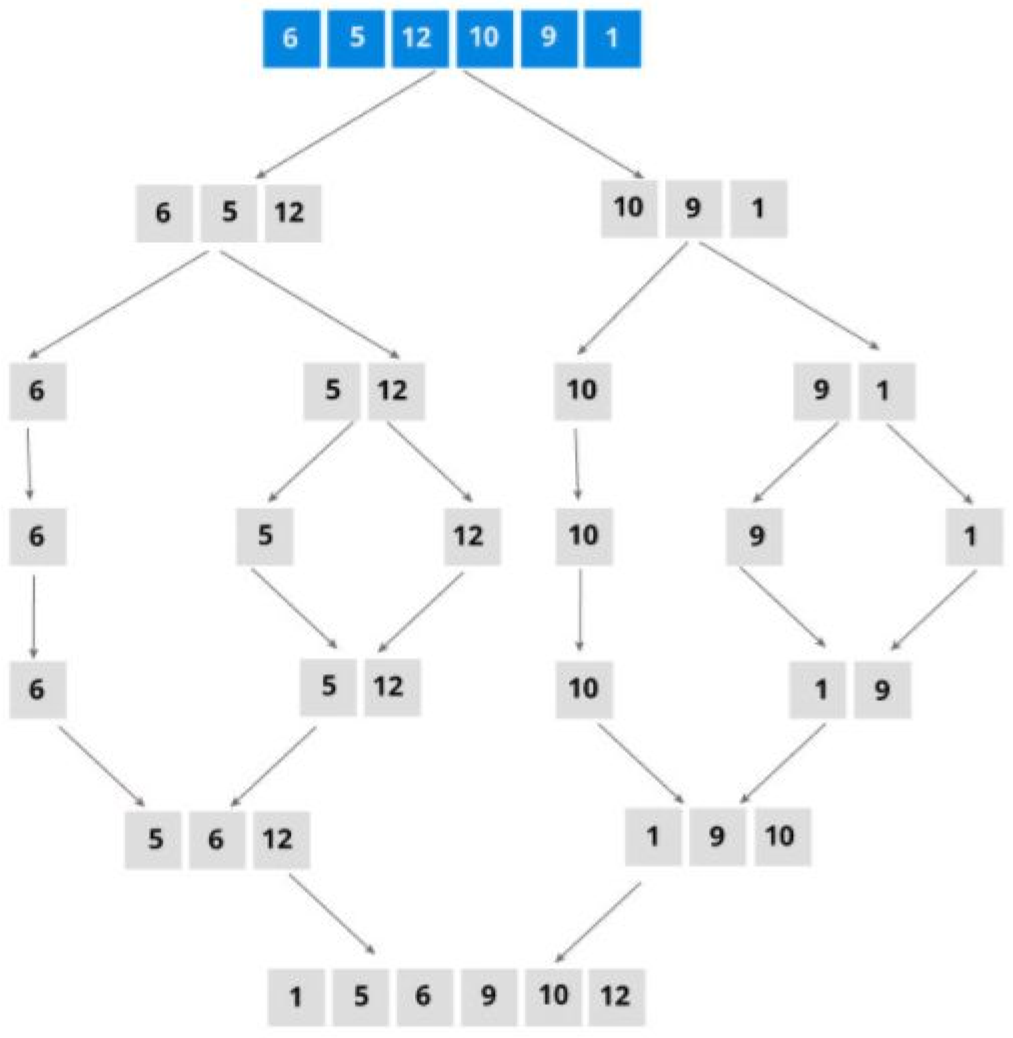
Figure: Selection Sort

MERGE SORT

The MergeSort function repeatedly divides the array into two halves until we reach a stage where we try to perform MergeSort on a subarray of size 1

Using the Divide and Conquer technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.

Suppose we had to sort an array A. A subproblem would be to sort a sub-section of this array starting at index p and ending at index r, denoted as A[p..r].



If q is the half-way point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

When the conquer step reaches the base step and we get two sorted subarrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creating a sorted array A[p..r] from two sorted subarrays A[p..q] and A[q+1, r]

QUICK SORT

To get started right away, just tap any placeholder text (such as this) and start typing to replace it with your own.

Want to insert a picture from your files or add a shape, text box, or table? You got it! On the Insert tab of the ribbon, just tap the option you need.

Find even more easy-to-use tools on the Insert tab, such as to add a hyperlink, insert a comment, or add automatic page numbering.

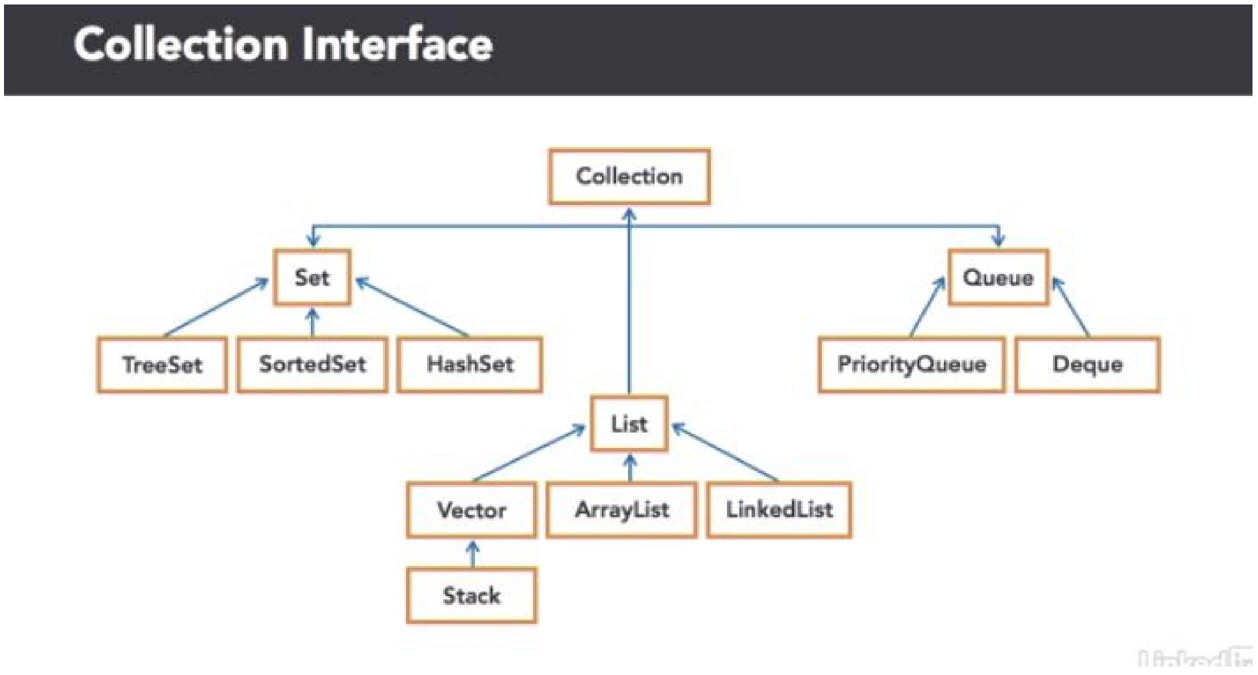
HEAP SORT

To get started right away, just tap any placeholder text (such as this) and start typing to replace it with your own.

Want to insert a picture from your files or add a shape, text box, or table? You got it! On the Insert tab of the ribbon, just tap the option you need.

Find even more easy-to-use tools on the Insert tab, such as to add a hyperlink, insert a comment, or add automatic page numbering.

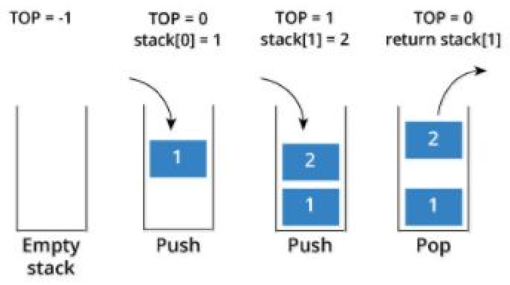
STACK



A stack is an object or more specifically an abstract data structure(ADT) that allows the following operations:

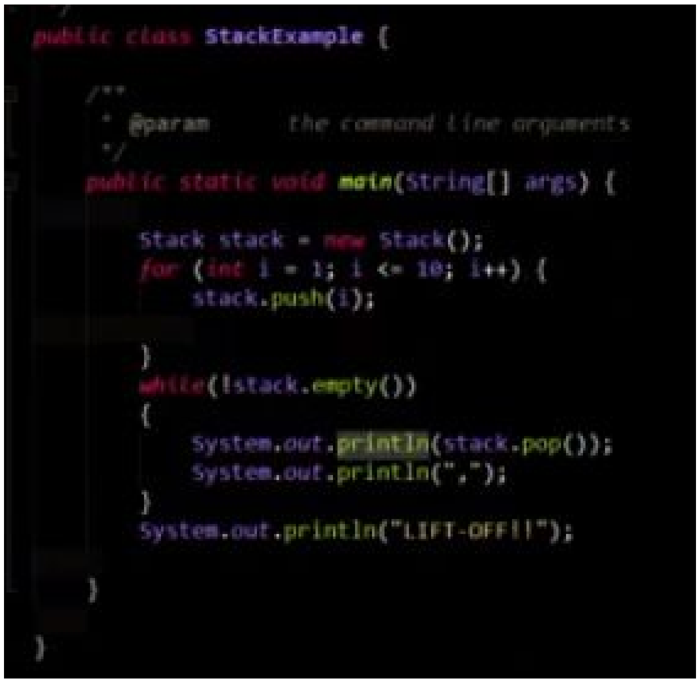
- Push: Add element to top of stack
- Pop: Remove element from top of stack
- IsEmpty: Check if stack is empty
- IsFull: Check if stack is full
- Peek: Get the value of the top element without removing it

  A stack is an easy waye to reverse a collection of values.



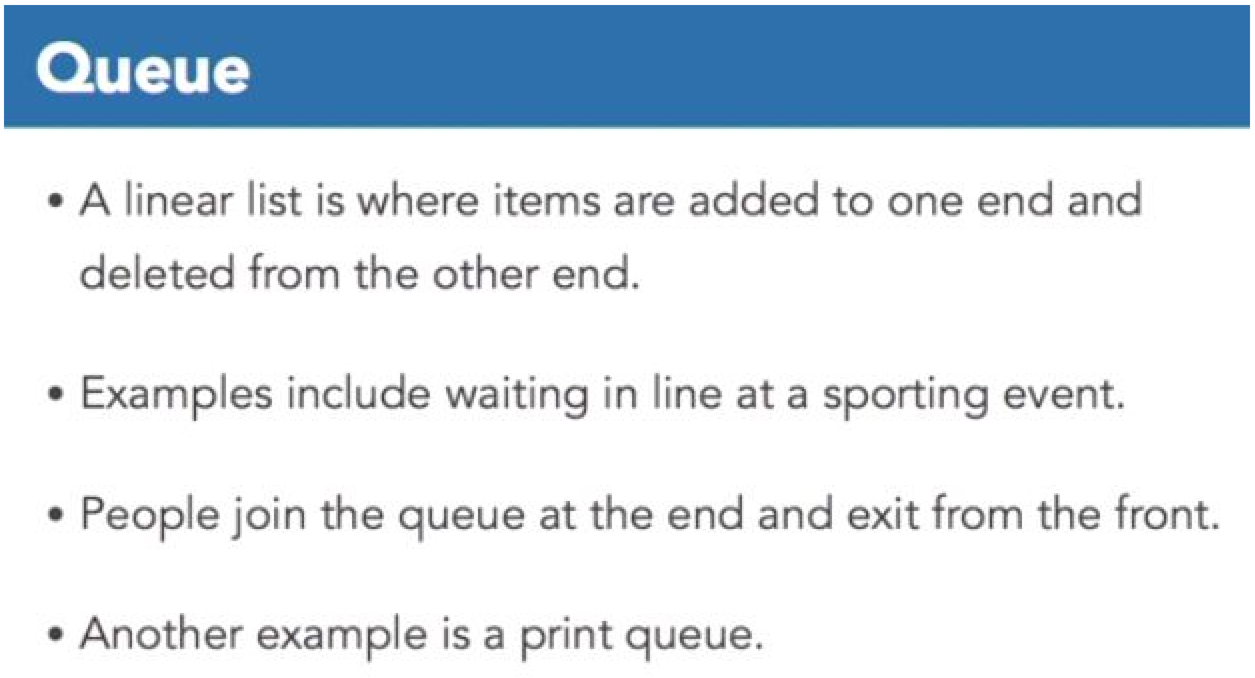
The operations work as follows:

1. A pointer called TOP is used to keep track of the top element in the stack.
2. When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing TOP == -1.
3. On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.
4. On popping an element, we return the element pointed to by TOP and reduce its value.
5. Before pushing, we check if stack is already full
6. Before popping, we check if stack is already empty



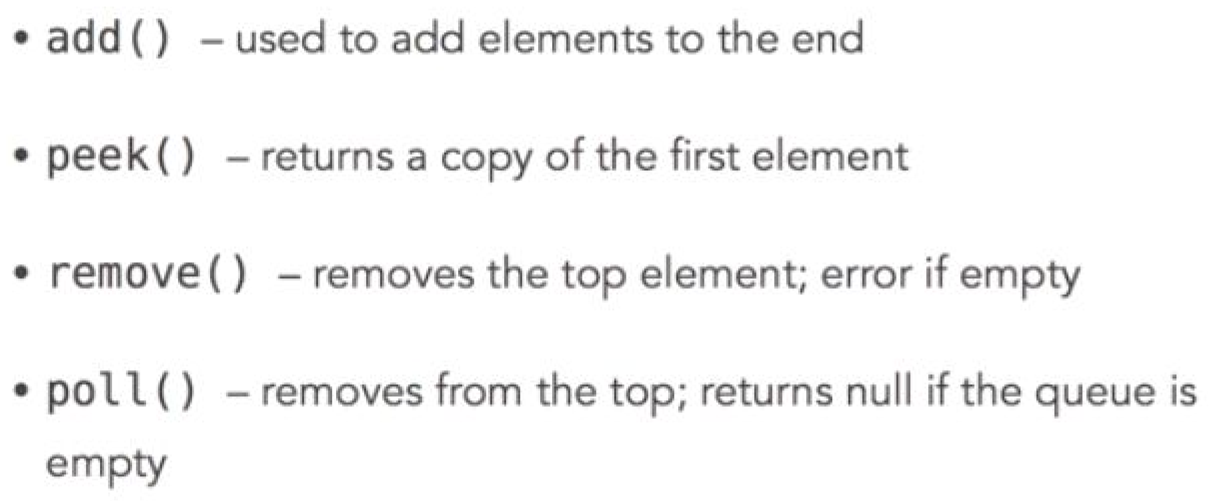
```
public class StackExample {

    /**
     * @param      the command line arguments
     */
    public static void main(String[] args) {

        Stack stack = new Stack();
        for (int i = 1; i <= 10; i++) {
            stack.push(i);

        }
        while(!stack.empty())
        {
            System.out.println(stack.pop());
            System.out.println(",");
        }
        System.out.println("LIFT-OFF!!");

    }

}
```

QUEUE



Queue

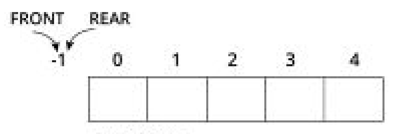
- A linear list is where items are added to one end and deleted from the other end.
- Examples include waiting in line at a sporting event.
- People join the queue at the end and exit from the front.
- Another example is a print queue.



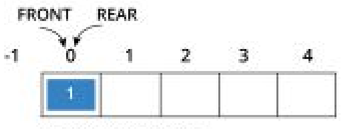
- `add()` – used to add elements to the end
- `peek()` – returns a copy of the first element
- `remove()` – removes the top element; error if empty
- `poll()` – removes from the top; returns null if the queue is empty

A queue is an object or more specifically an abstract data structure(ADT) that allows the following operations:

- Enqueue: Add element to end of queue
- Dequeue: Remove element from front of queue
- IsEmpty: Check if queue is empty
- IsFull: Check if queue is full
- Peek: Get the value of the front of queue without removing it

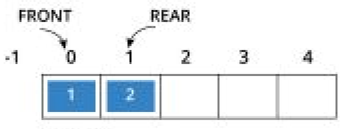


Empty Queue



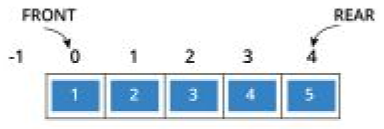
EnQueue first element

Queue operations work as follows:



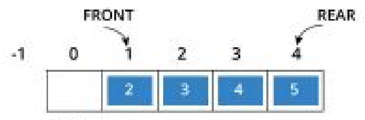
EnQueue

1. Two pointers called FRONT and REAR are used to keep track of the first and last elements in the queue.
2. When initializing the queue, we set the value of FRONT and REAR to -1.
3. On enqueing an element, we increase the value of REAR index and place the new element in the position pointed to by REAR.

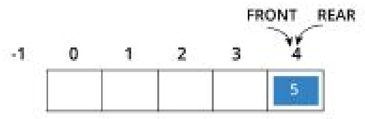


EnQueue

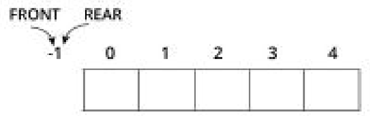
4. On dequeueing an element, we return the value pointed to by FRONT and increase the FRONT index.
5. Before enqueing, we check if queue is already full.
6. Before dequeuing, we check if queue is already empty.
7. When enqueing the first element, we set the value of FRONT to 0.
8. When dequeing the last element, we reset the values of FRONT and REAR to -1.



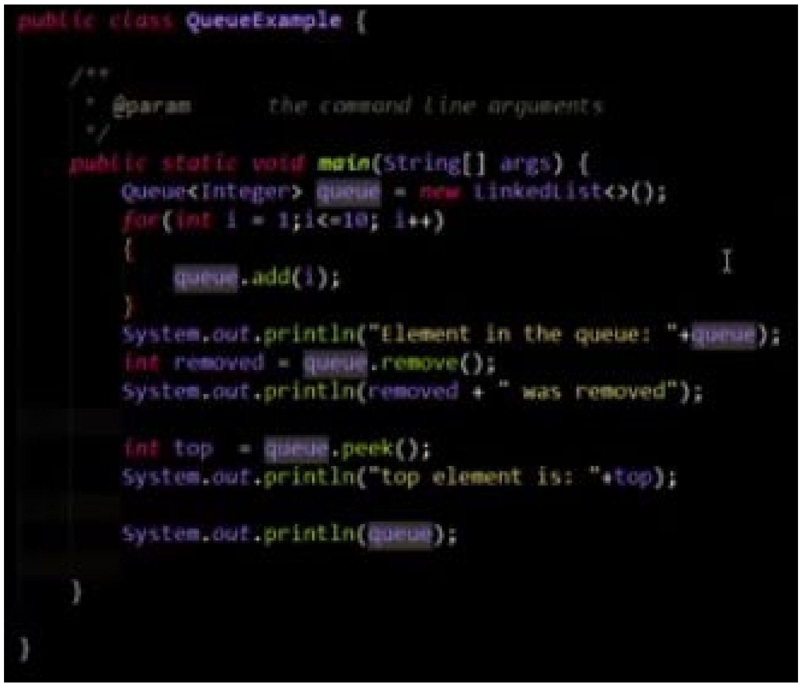
DeQueue



DeQueue last element



Empty Queue



```
public class QueueExample {

    /**
     * @param      the command line arguments
     */
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();
        for(int i = 1;i<=10; i++)
        {
            queue.add(i);
        }
        System.out.println("Element in the queue: "+queue);
        int removed = queue.remove();
        System.out.println(removed + " was removed");

        int top  = queue.peek();
        System.out.println("top element is: "+top);

        System.out.println(queue);

    }

}
```

Queue implementation

PRIORITY QUEUE

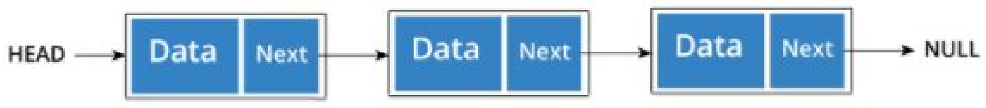
To get started right away, just tap any placeholder text (such as this) and start typing to replace it with your own.

Want to insert a picture from your files or add a shape, text box, or table? You got it! On the Insert tab of the ribbon, just tap the option you need.

Find even more easy-to-use tools on the Insert tab, such as to add a hyperlink, insert a comment, or add automatic page numbering.

LINKED LIST

It is a series of connected "nodes" that contains the "address" of the next node. Each node can store a data point which may be a number, a string or any other type of data.



You have to start somewhere, so we give the address of the first node a special name called HEAD.

Also, the last node in the linkedlist can be identified because its next portion points to NULL.

You can add elements to either beginning, middle or end of linked list.

Add to the beginning:-

- Allocate memory for new node
- Store data
- Change next of new node to point to head
- Change head to point to recently created node

Add to the end:-

- Allocate memory for new node
- Store data
- Traverse to last node
- Change next of last node to recently created node

Add to the middle

- Allocate memory and store data for new node
- Traverse to node just before the required position of new node
- Change next pointers to include new node in between

You can delete either from beginning, end or from a particular position.

- Delete from beginning:- Point head to the second node

Delete From end: -

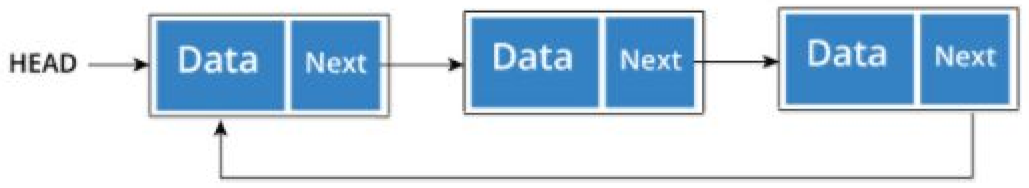
- Traverse to second last element
- Change its next pointer to null

Delete from middle

- Traverse to element before the element to be deleted
- Change next pointers to exclude the node from the chain

CIRCULAR LINKED LIST

A circular linked list is a variation of linked list in which the last element is linked to the first element. This forms a circular loop.

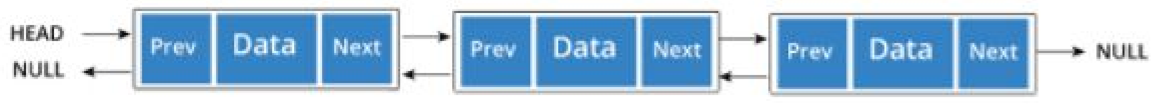


A circular linked list can be either singly linked or doubly linked.

- for singly linked list, next pointer of last item points to the first item
- In doubly linked list, prev pointer of first item points to last item as well.

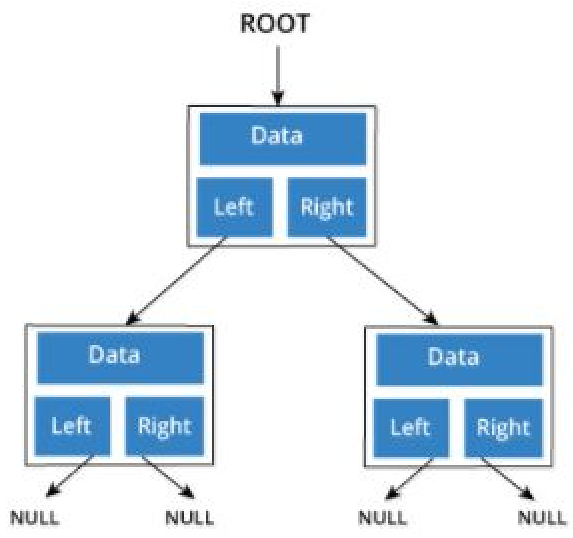
DOUBLY LINKED LIST

We add a pointer to the previous node in a doubly linked list. Thus, we can go in either direction: forward or backward.



BINARY TREE

A node of a binary tree is represented by a structure containing a data part and two pointers to other structures of the same type.



A special pointer called ROOT points to the node that is the parent of all the other nodes.

Also, the nodes that don't have any children have their left and right pointers point to NULL.

BINARY SEARCH TREE

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

- It is called a binary tree because each tree node has maximum of two children.
- It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

The properties that separates a binary search tree from a regular binary tree is

1. All nodes of left subtree are less than root node
2. All nodes of right subtree are more than root node
3. Both subtrees of each node are also BSTs i.e. they have the above two properties

Find Value

If the value is below root, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree and if the value is above root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.

If the value is found, we return the value so that it gets prorogated in each recursion until the root returns the final value.

If the value is not found, we eventually reach the left or right child of a leaf node which is NULL and it gets propagated and returned.

Insert Value

We keep going to either right subtree or left subtree depending on the value and when we reach a point left or right subtree is null, we put the new node there.

GRAPH DATA STRUCTURE

A graph data structure is a collection of nodes that have data and are connected to other nodes.

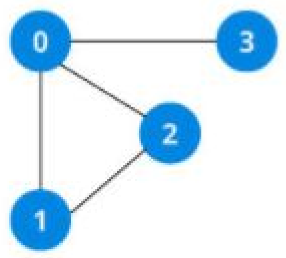
More precisely, a graph is a data structure (V,E) that consists of

- A collection of vertices V
- A collection of edges E, represented as ordered pairs of vertices (u,v)

An adjacency matrix is 2D array of V x V vertices. Each row and column represent a vertex.

If the value of any element a[i][j] is 1, it represents that there is an edge connecting vertex i and vertex j.

The adjacency matrix for the graph we created above is -



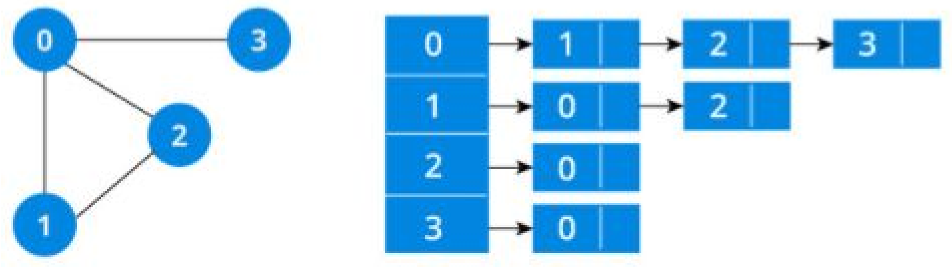
	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

Figure 2. Adjacency Matrix

An adjacency list represents a graph as an array of linked list.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

The adjacency list for the graph we made in the first example is as follows:



An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space.

The most common graph operations are:

- *Check if element is present in graph*
- *Graph Traversal*
- *Add elements(vertex, edges) to graph*
- *Finding path from one vertex to another*

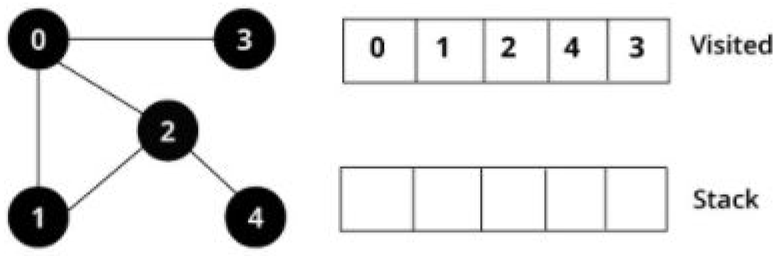
DFS ALGORITHM

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of stack.

4. Keep repeating steps 2 and 3 until the stack is empty.



BFS ALGORITHM

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

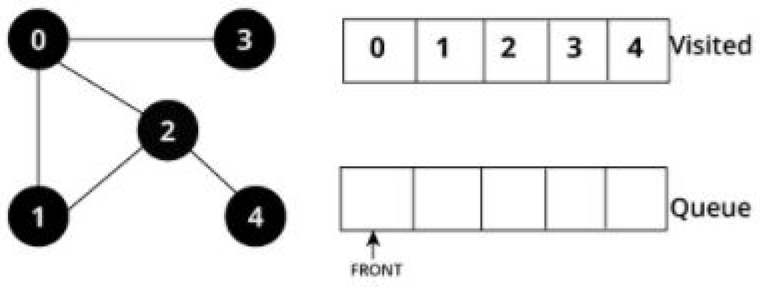


Figure 3. BFS

DYNAMIC PROGRAMMING

Dynamic Programming is a technique in computer programming that helps to efficiently solve a class of problems that have overlapping subproblems and optimal substructure property.

Such problems involve repeatedly calculating the value of the same subproblems to find the optimum solution.

Dynamic programming works by storing the result of subproblems so that when their solutions are required, they are at hand and we do not need to recalculate them.

This technique of storing value of subproblems is called memoization. By saving the values in the array, we save time for computations of sub-problems we have already come across.

Unless there is a presence of overlapping subproblems like in the fibonacci sequence problem, only recursion is enough to reach the solution using a divide and conquer approach.

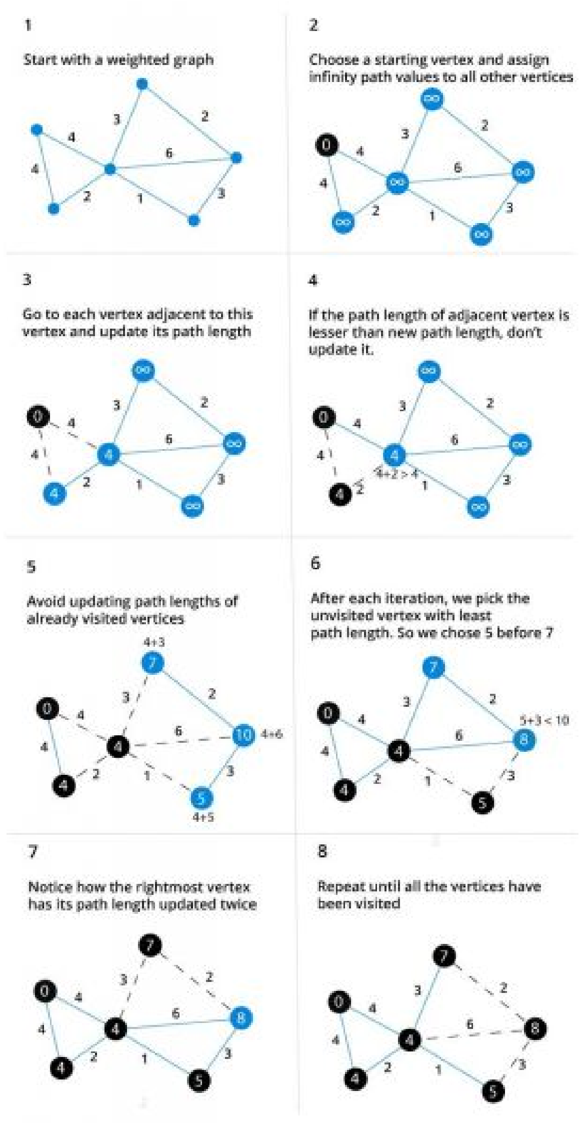
Dynamic programming, on the other hand, find the optimal solution to subproblems and then making an informed choice to combine the results of those subproblems to find the most optimum solution.

DIJKISTRA's PROGRAMMING

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph. It differs from minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph. Dijkstra's Algorithm works on the basis that any subpath B -> D of the shortest path A -> D between vertices A and D is also the shortest path between vertices B and D.

Djikstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbours to find the shortest subpath to those neighbours.

The algorithm uses a greedy approach in the sense that we find the next best solution



1
Start with a weighted graph
2
Choose a starting vertex and assign infinity path values to all other vertices
3
Go to each vertex adjacent to this vertex and update its path length
4
If the path length of adjacent vertex is lesser than new path length, don't update it.
4+2 > 4
5
Avoid updating path lengths of already visited vertices
4+3
4+6
4+5
6
After each iteration, we pick the unvisited vertex with least path length. So we chose 5 before 7
5+3 < 10
7
Notice how the rightmost vertex has its path length updated twice
8
Repeat until all the vertices have been visited

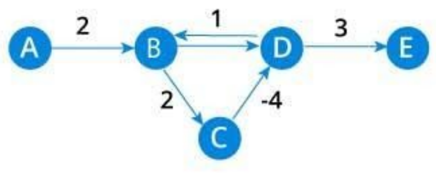
hoping that the end result is the best solution for the whole problem.

Bellman Ford Programming

Bellman Ford algorithm helps us find the shortest path from a vertex to all other vertices of a weighted graph.

It is similar to Dijkstra's algorithm but it can work with graphs in which edges can have negative weights. For example, cashflow, heat released/absorbed

Negative weight edges can create negative weight cycles i.e. a cycle which will reduce the total path distance by coming back to the same point.



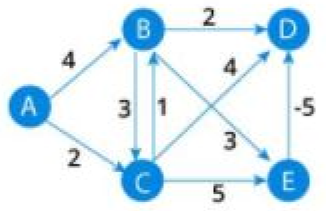
Shortest path algorithms like Dijkstra's Algorithm that aren't able to detect such a cycle can give an incorrect result because they can go through a negative weight cycle and reduce the path length

Bellman Ford algorithm works by overestimating the length of the path from the starting vertex to all other vertices. Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths. By doing this repeatedly for all vertices, we are able to guarantee that the end result is optimized.

Cont.d on next page

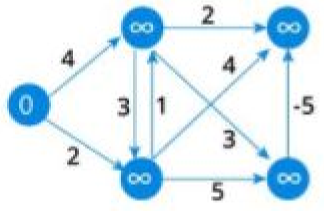
1

Start with a weighted graph



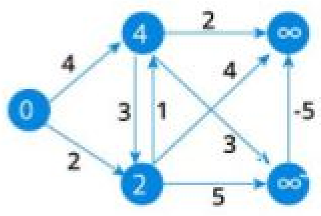
2

Choose a starting vertex and assign infinity path values to all other vertices



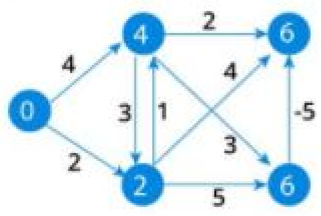
3

Visit each edge and relax the path distances if they are inaccurate



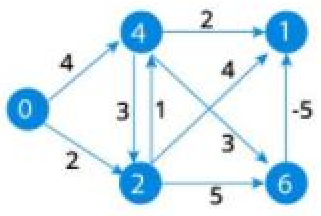
4

We need to do this V times because in the worst case, a vertex's path length might need to be readjusted V times



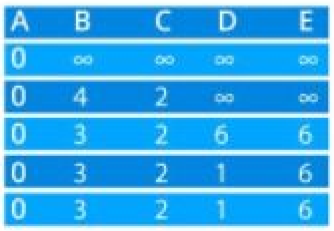
5

Notice how the vertex at the top right corner had its path length adjusted



6

After all the vertices have their path lengths, we check if a negative cycle is present.



A	B	C	D	E
0	∞	∞	∞	∞
0	4	2	∞	∞
0	3	2	6	6
0	3	2	1	6
0	3	2	1	6

Figure 4 bellman

PRIM's ALGORITHM

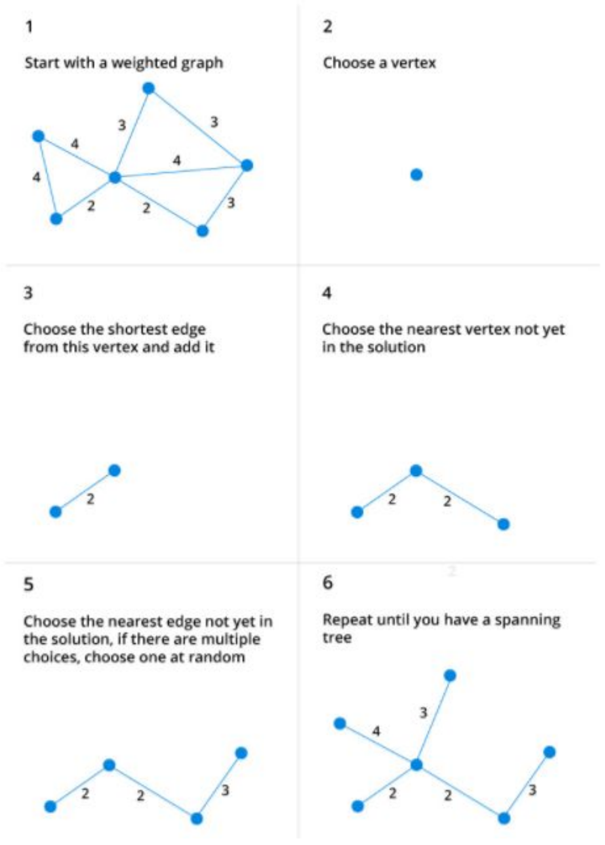
Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

It falls under a class of algorithms called greedy algorithms which find the local optimum in the hopes of finding a global optimum. We start from one vertex and keep adding edges with the lowest weight until we we reach our goal.

The steps for implementing Prim's algorithm are as follows:

1. Initialize the minimum spanning tree with a vertex chosen at random.
2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
3. Keep repeating step 2 until we get a minimum spanning tree


1
Start with a weighted graph
2
Choose a vertex
3
Choose the shortest edge from this vertex and add it
4
Choose the nearest vertex not yet in the solution
5
Choose the nearest edge not yet in the solution, if there are multiple choices, choose one at random
6
Repeat until you have a spanning tree
4
4
3
3
4
2
2
3

KRUSKAL'S ALGORITHM

Kruskal's algorithm is another popular minimum spanning tree algorithm that uses a different logic to find the MST of a graph. Instead of starting from an vertex, Kruskal's algorithm sorts all the edges from low weight to high and keeps adding the lowest edges, ignoring those edges that create a cycle.

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which form a tree that includes every vertex has the minimum sum of weights among all the trees that can be formed from the graph.

It falls under a class of algorithms called greedy algorithms which find the local optimum in the hopes of finding a global optimum.We start from the edges with the lowest weight and keep adding edges until we we reach our goal.

The steps for implementing Kruskal's algorithm are as follows:

- Sort all the edges from low weight to high
- Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
- Keep adding edges until we reach all vertices.

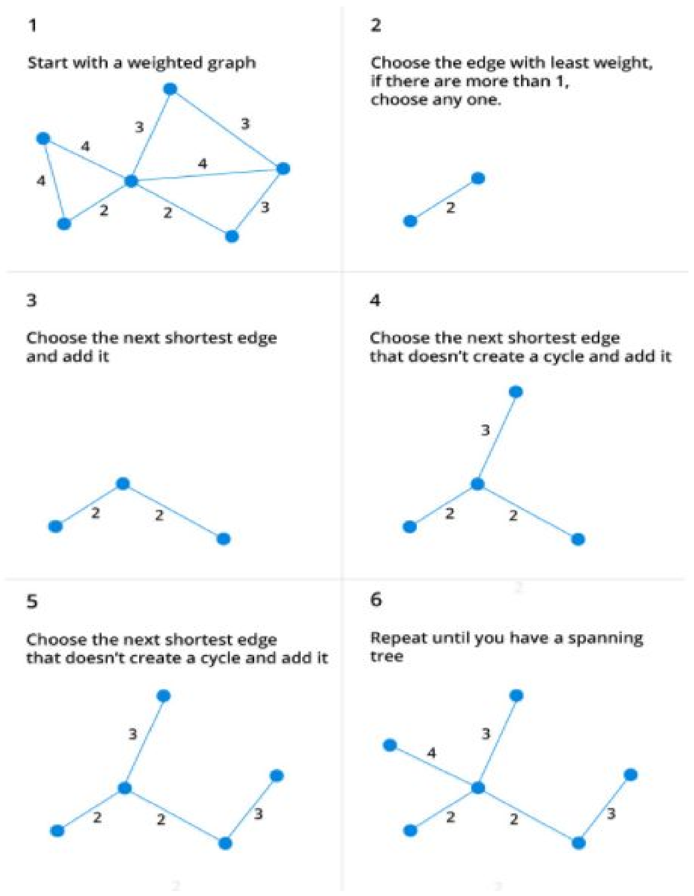

1
Start with a weighted graph
4
4
3
3
4
2
2
3
2
Choose the edge with least weight, if there are more than 1, choose any one.
2
3
Choose the next shortest edge and add it
2
2
4
Choose the next shortest edge that doesn't create a cycle and add it
3
2
2
5
Choose the next shortest edge that doesn't create a cycle and add it
3
2
2
3
6
Repeat until you have a spanning tree
4
3
2
2
3

Figure 5 Kruskal

______________________________ENDS_______________HERE___s______________________