Table of Contents

In [2]:

```
import matplotlib.pyplot as plt
import numpy as np
from binarytree import tree, bst, heap
import ipycytoscape
```

1 Array

1.1 Description

List is a data structure that is a mutable, ordered sequence of elements. There are two implementations of the list: array and linked list.

Array is the most common implementation of the lists. The main difference from the list - each element in the array has a location - index (starts from 0).

Structure and order of the list depends on programing language:

- same types of objects in the same array or different types,
- fixed size or not.

1.2 Operations and time complexity

Inserting an element could be very complicated and inefficient and could take linear time or O(n).

Deletion has the same problem, because we will get an empty place and need to move other elements of the array.

Operations	Time Complexity	Comments
add(element)	$O(1)$	fast operation
remove(element)	$O(n)$	shift all elements which come after the removed one
insert(element, index)	$O(n)$	shift all element in order to make space
get(element)	$O(1)$	fast operation

1.3 Advantages

- easy access to the elements by index
- easy to traverse
- easy searching for element

1.4 Disadvantages

- static structure, which means fixed size
- insertion and delition are complicated since we need to manage memory space

1.5 Applications

We can use arrays in order to implement other data structure for ex. linked lists,stack,queue.

Arrays is used in many datatables as one-dimensional arrays.

Also we can implement 2-dimensional arrays for storing data:

In [6]:

```
x = np.array([[2, 2, 3], [1, 10, 6]], np.int32)
print(x)
```

```
[[ 2  2  3]
 [ 1 10  6]]
```

---

2 Linked List

2.1 Description

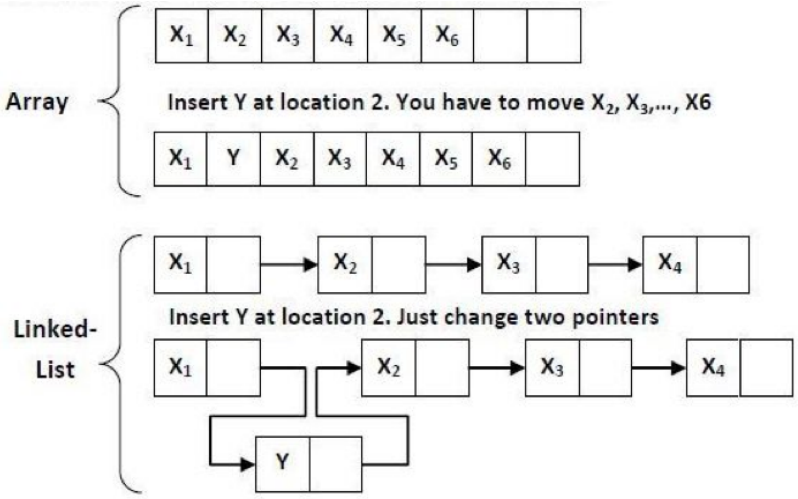
- extension of the list;
- there is order but there are no indexes
- characterized by links: each element has information about the next element

The main difference between linked lists and array is that in linked lists we store a reference to the next element in the list.

Difference between array and linked list:

	Array	Linked list
size	Fixed number, it has to be specific during declaration	Max size depends on heap
order	Stored consecutively	Stored randomly
searching	Linear and binary	Linear
accessing	Direct or random	Sequential access so need traverse starting from the first node

2.2 Operations and time complexity



Picture 1. Arrays Vs Linked Lists source (https://www.interviewbit.com/tutorial/arrays-vs-linked-lists/)

Operations	Time Complexity	Comments
add(element)	$O(n)$	
remove(element)	$O(1)$	
insert(element, index)	$O(1)$	link new element with next and previous
get(element)	$O(n)$	traverse from the first node
search	$O(n)$	traverse from the first node

2.3 Advantages

- we don't need to know the size in advance,
- inserting and deleting elements is much easier compared to the lists as can be seen from Figure 1. To insert an element, we need to change the next reference to point to the new object and assign next pointer.

2.4 Disadvantages

- cannot provide fast random access of element,
- linear look up time.

2.5 Applications

- implementation of stacks, queues, graphs,
- keeping directory of names,
- dynamic memory allocation

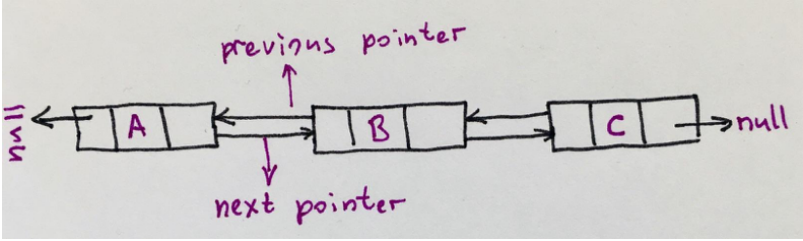
2.6 Code example

linked-list.py (data-structure/linked-list.py)

3 Doubly linked list

3.1 Description

Doubly linked list - is a linked list which has an extra pointer (previous pointer). It has the same rules as linked list, but in addition we can traverse the list in both directions.



Picture 2. Doubly linked lists

3.2 Operations and time complexity

Time complexity is the same as Singly-Linked Lists.

3.3 Advantages

- allows traverse in both direction,
- simple reversing,
- allocate or reallocate memory easily.

3.4 Disadvantages

- uses extra memory,
- elements are accessed sequentially so there is no direct access.

3.5 Applications

- one of the examples is system where both front and back navigation is required (songs in music player, pages (previous and next) in web browser)

4 Stack

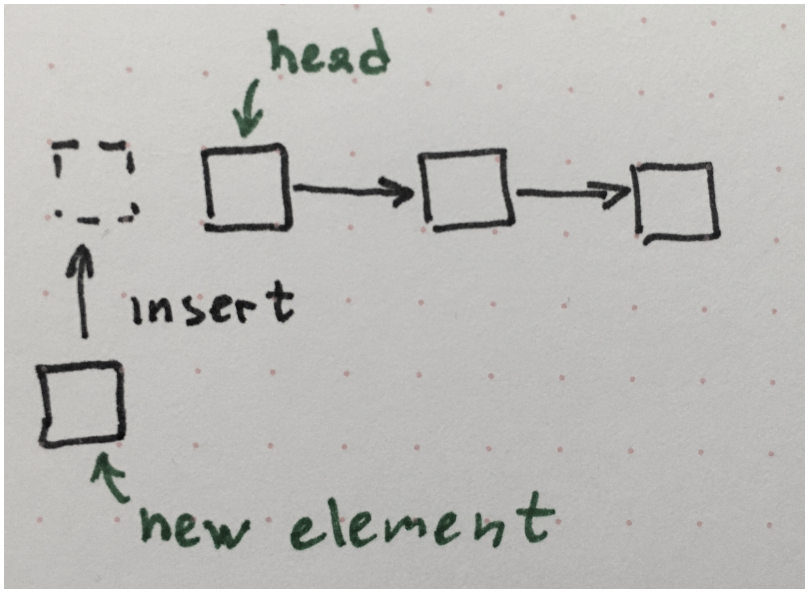
4.1 Description

Stack - is a stack of objects when you place objects on top of each other. Useful when we care about the most recent elements or the order in which we save element matter.

L.I.F.O. - last in first out principle, which means, when we add an item to a stack, we place it on top of the stack. When we remove an item from a stack, we remove the top-most item.

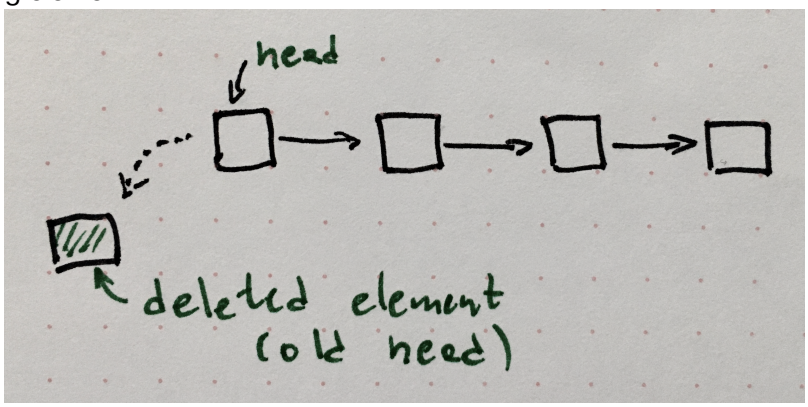
Size of stack is not fixed and changes with each operation.

4.2 Operations and time complexity



Picture 3. Adding element to a stack

Push element - is adding element.



Picture 4. Deleting element from a stack

Pop element - take element off the stack (instead remove).

Operations	Time Complexity	Comments
pop(element)	$O(1)$	Since the current top position is known
push	$O(1)$	the current position is known
get(element)	$O(n)$	traverse from the first node
search	$O(n)$	traverse from the first node

4.3 Advantages

- it is usefull when we care about the most recent elements
- efficient in adding and removing items
- no fixed size
- easy to use for reversing

4.4 Disadvantages

- memory is limited,
- random access is not possible.

4.5 Applications

- mobile development (like IOS, react native) uses the navigation stack to push and pop views screens
- web development (navigation, most recent posts, emails etc)

4.6 Code example

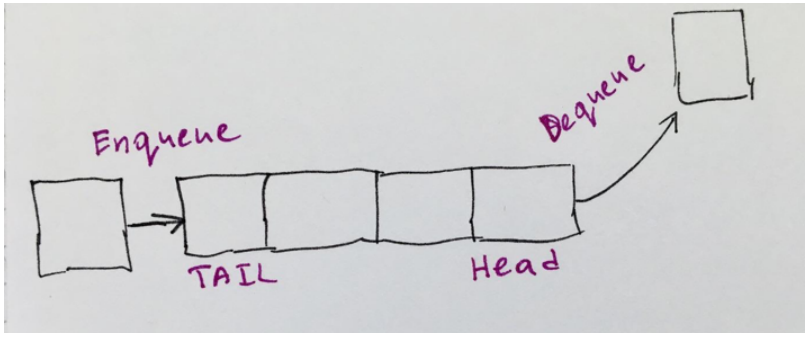
For example in python, we can use linked lists and class to implement stacks and then to use methods like push() and pop()

stack.py (data-structure/stack.py)

5 Queue

5.1 Description

Queues has opposite to stack structure - F.I.F.O. (First in first out structure). Simple queue is the most basic queue. In this queue, the enqueue operation takes place at the tail, while the dequeue operation takes place at the head.



Picture 5. Adding and removing data in queue

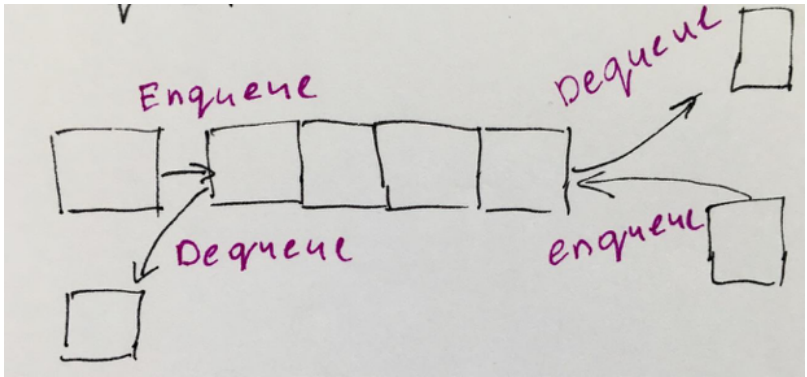
Enqueue – adding element to the tail.

Dequeue – removing head element.

Peek – looking at the head without removing.

Other types of queue:

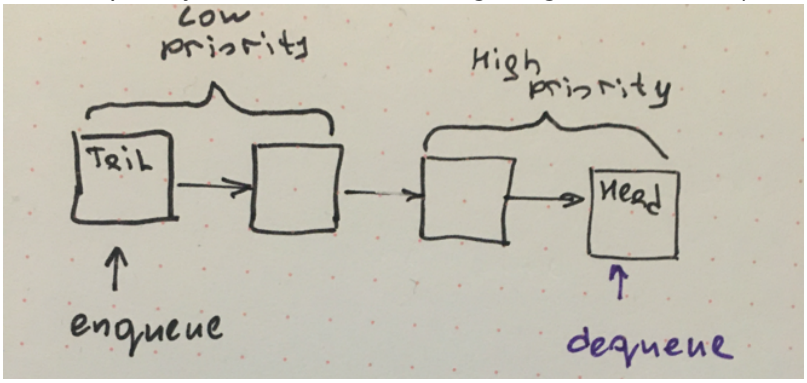
Deque or double-ended queue is the queue that serves both ways.



Picture 6. Adding and removing data in dequeue.

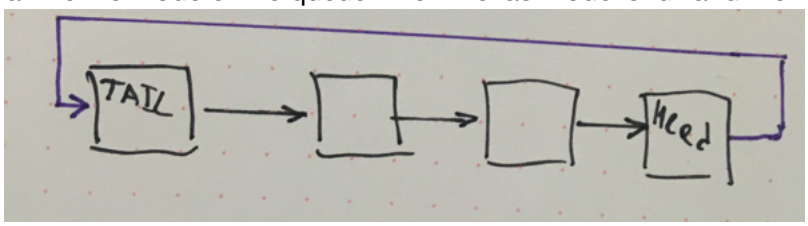
Priority queue – is a special type of queue in which each element is assigned to a priority and is deleting happens according to its priority with the following rules:

- removing the element with highest priority first
- if two elements has the same priority the oldest element is getting removed first (according to the order)



Picture 7. Priority queue.

Circular queue – the queue where the last node points to the first node and creates a circular connection. It allows to insert an item at the first node of the queue when the last node is full and the first node is free.



Picture 8. Circular queue.

5.2 Operations and time complexity

Similar to stacks data structure

Operations	Time Complexity	Comments
Insertion	$O(1)$	Enqueue
Deletion	$O(1)$	Dequeue
Get(element)	$O(n)$	We need to traverse through all elements
Search	$O(n)$	We need to traverse through all elements

5.3 Advantages

- handle multiple data types,
- flexible

5.4 Disadvantages

- random access is not possible.
- not easy to search elements

5.5 Applications

- Serving different kind of requests like a printer
- Different operating systems processes (CPU tasks).
- Memory management

5.6 Code example

queue.py (data-structure/queue.py)

6 Hash maps

6.1 Description

Set is data structure which similar to lists, but does not have ordering of the elements and in addition does not allow repeatition of elements.

Map is set-base data structure of key-value pairs, where keys are a set, bacause the keys has to be unique. For example, Python has built-in data type called a dictionary, which contains key-value pairs.

Hash Function is a function which transforms any kind of data into a number. Value ---> Hash value.

> The purpose of hashing function is to transfer some value into one that can be stored and retrieved easily (c) Udacity Data structure and algorithms.

One of the ways how hashing mignt be working?

1. take last a few digits of the number, because they are more random
2. divide by constant number (like 10)
3. reminder is going to be an index

Good hashing function:

- is consistent means that it maps the same name to the same index every time,
- knows the size of the array and only returns valid indexes

Collisions - is when the result of the hash function is the same for two different numbers.

Two ways to fix it:

1. To change constant number by which we divide the last digits or to change hash function. This is not a very efficient approach because we have to change every time we have collisions.
2. To change the structure of the array - instead of storing only one value in each slot we can store multiple values or collections. Each slot - bucket. Bucket is a more efficient approach, but it is better to have not more than 3 values stored in the bucket.

Load factor shows when we need to rehash values.

> *Load Factor = Number of Entries / Number of Buckets.* The closer our load factor is to 1 (meaning the number of values equals the number of buckets), the better it would be for us to rehash and add more buckets. Any table with a load value greater than 1 is guaranteed to have collisions. source

There is a rule if the load factor is greater than 0,7, it's time to resize the hash table.

We can use hashing with string keys. In that case hash function converts letters into the numbers. Individual letters can be transformed into ASCII values (most of the programming languages have build in function)

6.2 Operations and time complexity

6.2 Operations and time complexity

In the average case, hash tables take constant time - O(1) for any operation, which is very fast. But for worst case the time is linear, when we have to store more data in one index. The main goal is to try to implement hash tables in a way it use only average time, which means it has to be without collisions.

Operations	Average Time Complexity	Worst Time Complexity
Insertion	$O(1)$	$O(n)$
Deletion	$O(1)$	$O(n)$
Search	$O(1)$	$O(n)$

6.3 Advantages

- easy to model a relationship from one item to another,
- fast search, insert, and delete,
- avoiding duplications

6.4 Disadvantages

- avoiding collisions is hard,
- many collisions make operations slow

6.5 Applications

- using hash tables for lookups in large scale (for example DNS resolution),
- preventing duplicate entries
- database indexing
- caching data instead of making your server do work

Python implementation of hash maps is dictionary. We can just use dict() function to create a new hash map.

In [9]:

```
modules = dict()
modules['algorithms'] = 'SE_02'
modules['basics'] = 'SE_01'
print(modules)
print(modules['algorithms'])
```

```
{'algorithms': 'SE_02', 'basics': 'SE_01'}
SE_02
```

7 Tree

7.1 Description

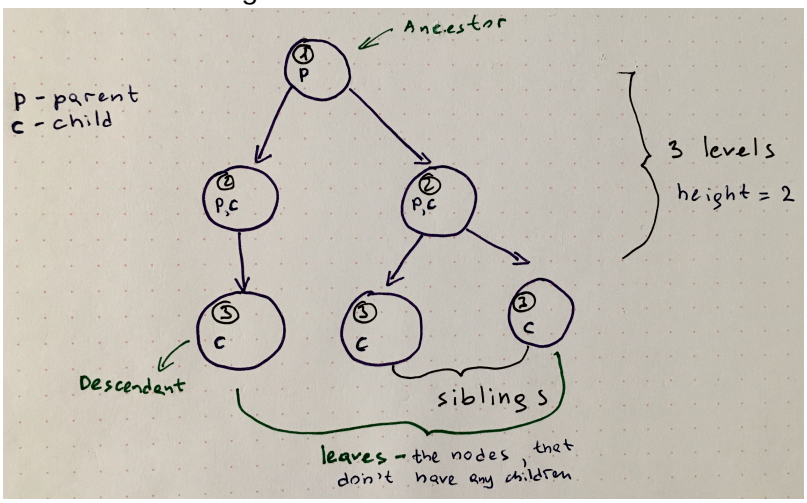
Tree - is non-linear data structusre, where data organized hierarchically from the root (kind of linked list extention), it is collection of nodes linked together. Tree:

- has connected nodes
- has no cycles

We can see on the Picture 9 the basic represantation of the tree structure. Tree has some specific terms:

- **root** - the first element

- **leaf** - the node that has no children
- **edge** - connection between nodes
- **height of the node** - is the number of edges between the node and the furthest leaf on the tree.
- **depth of the node** - is the number of edges to the root.



Picture 9. Basic representation of tree.

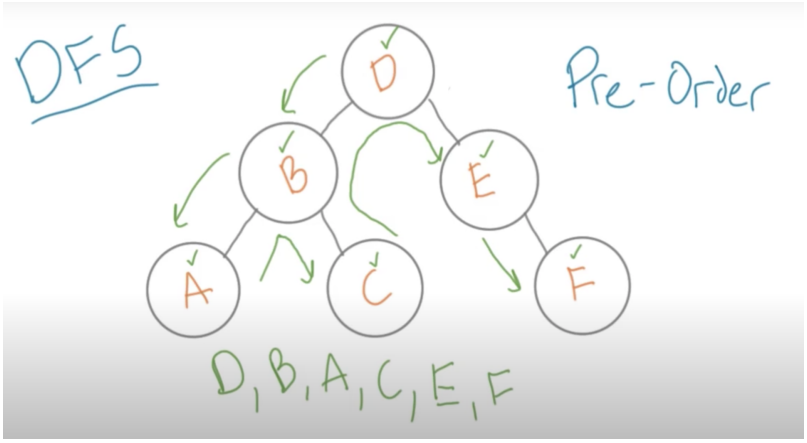
7.2 Tree traversal

Since trees are not linear, it is not so easy to traverse.

> We can't search or sort elements unless we have a way to make sure we can visit all elements first. (c) Udacity

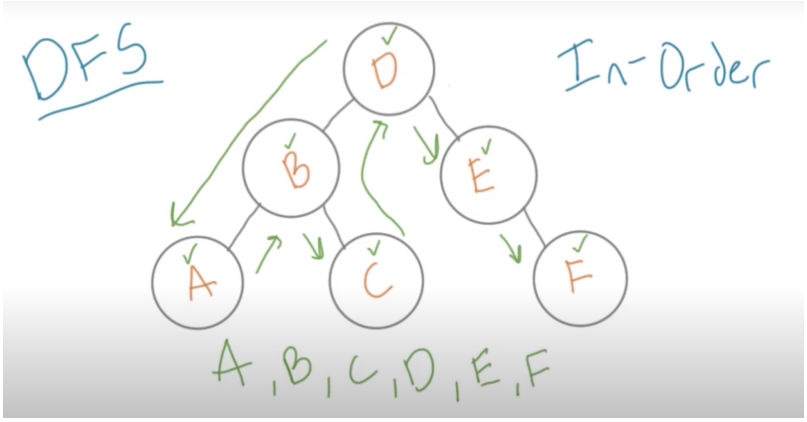
Two approaches for tree traversal:

1. DFS - depth-first search - exploring children nodes is priority:
   - pre-order way - check node before traversing further (node-left-right)



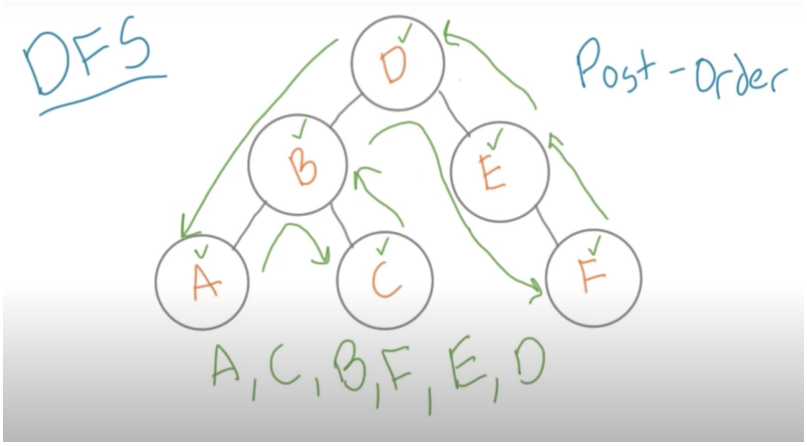
Picture 10. Pre-order DFS screenshot from Udacity.

   - in-order way (left-node-right)



Picture 11. In-order DFS screenshot from Udacity.

- post-order way



Picture 12. Post-order DFS screenshot from Udacity.

2. BFS - breadth-first search - expoloring nodes on the same level is priority. We can call it level order traversal. We start from the root node and go to children from left to the right and so on till we viset all leaves.

7.3 Operations and time complexity of binary tree

Binary tree - is when a node can have maximum two children

Maximum number of nodes at level $i = 2^i$

- full- every node has 0 or 2 children
- complete - all the levels are completely filled except possibly the last level and the last level has all keys as left as possible
- perfect - all the internal nodes have two children and all leaf nodes are at the same level.

In [4]:

```
binary_tree = tree(height=3, is_perfect=False)
print(binary_tree)
```

```
  ______9__
 /         \
14          5
  \        / \
   4      3   2
    \          \
     0          1
```

Operations	Time Complexity (worst)	Comments
Insertion	$O(n)$	
Deletion	$O(n)$	We need search for the element at first. Deleting a leaf is easy. Deleting internal node needs moving child up (if one) or choosing and promoting one of the child (if two).
Get(element)	$O(n)$	We need to traverse through all elements
Search	$O(n)$	We need to traverse through all elements by using any traversal algorithm mentioned above

7.4 Operations and time complexity of binary search tree

Binary search tree(BST) - is a binary tree in which left child of a node has value less than the parent and right child has value greater than parent.

- balanced - height of the tree is O(log n) where n is the number of nodes, minimising number of nodes.
- unbalanced - nodes spread out on many leveles (could look like linked list)

Example of perfect. complete, balanced binary search tree:

In [23]:

```
bst_tree = bst(height=4, is_perfect=True)
print(bst_tree)
print(bst_tree.is_balanced)
print(bst_tree.is_complete)
```

```
                        _______________________15_______________________
                       /                                               \
               _______7_______                                 ___________23_________
              /               \                               /
\
     __3__             ___11___                 _____19___                  ___
_27___
    /     \           /        \               /         \              /
\
  1       5       9          _13          _17          _21          _25
_29
 / \     / \     / \        /   \        /   \        /   \        /   \
/   \
0   2   4   6   8   10     12    14     16    18     20    22     24    26
28    30

True
True
```

Height of BST is log(n)

Operations	Time Complexity(worst)	Time Complexity(average)	Comments
Insertion	$O(n)$	$O(log(n))$	We can say that in the average case time complexity is O(h) where h is height of BST, bacause we start at the root and move down doing comparisons until there is an open spot
Deletion	$O(n)$	$O(log(n))$	same rule
Get(element)	$O(n)$	$O(log(n))$	same rule
Search	$O(n)$	$O(log(n))$	same rule

7.5 Operations and time complexity of heaps

> *Heap* is the type of the tree where elements are arranged in decreasing or increasing order, such that a root element is either maximum or minimum value in a tree.
>
> - max heaps - parent always has bigger value than child
> - min heaps - parent always has lower value than child (c) Udacity

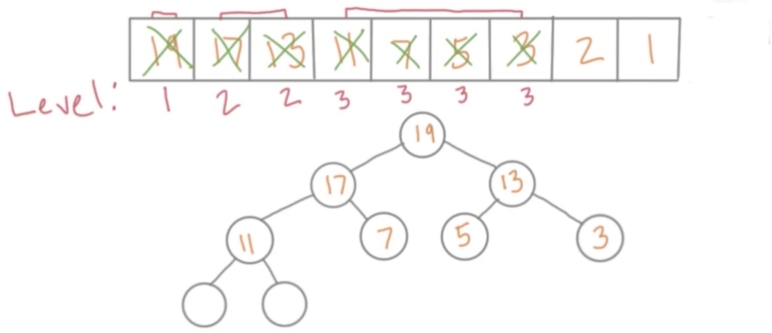
Heapify - reodering the tree, based on heap property. When we insert new element in the any open sport, we do then comparisons with its parent element and swapping untill it is ordered.

In [35]:

```
heap_tree = heap(height=3, is_max=True, is_perfect=True)
print(heap_tree)
```

```
          _________14_______
         /                  \
      __13__                __12__
     /      \              /      \
    8        11          7        10
   / \      /  \        / \      /  \
  3   2    9    0      4   1    6    5
```

Sorted array can be represented as a heap.



Picture 13. Represantation of tree in the array from Udacity.

For max binary heap:

Operations	Time Complexity	Comments
Insertion	$O(log(n))$	We put element in the open spot and then heapify. In the worst case it takes as many operations as height of the tree.
Deletion	$O(n)$	
Get(element)	$O(n)$	
Search	$O(n)$	If we have max or min heap we know which direction to move. Generally the search takes O(n/2) but it is still aprrox O(n)

7.6 Operations and time complexity of Red-Black Tree

Red-Black Tree - is extension BST where there is some specific rules to make sure that the tree is balanced:

- nodes are assigned to additional color property, so values must be red or black.
- red node cannot have a red parent or red child
- null leaf node (every node if it does not have 2 children, has to have null children). All leaf nodes are black
- root node must be black
- every path from the node to the descendant node must contain the same number of back nodes

Operations	Time Complexity	Comments
Insertion	$O(log(n))$	

Operations	Time Complexity	Comments
Deletion	$O(log(n)$	
Search	$O(log(n)$	

7.7 Advantages

- flexibility
- great for storing hierarchically organized data
- red-Black trees guarantee O(log(n)) for insertion, deletion and search
- quick search, insertion, deletion (if tree remains balanced).

7.8 Disadvantages

- more complicated than linear search, and benefit only from large numbers.
- accessing any node in the tree requires sequentially processing all nodes that appear before it in the node list
- difficult to control memory space
- complex process of deletion

7.9 Applications

- storing information hierarchally (file system)
- heaps are used to implement priority queues
- decision-making processes
- organising data for quick search, insertion, deletion (if tree is balanced)
- for network routing algorithms

8 Graphs (networks)

8.1 Description

> Graph is data structure designed to show relationships between objects, in addition, edges can store data.

Graphs can have cycles so there is no root node, but we should be careful, because it could lead to infinite loops:

- cyclic graph
- acyclic graph

Edges can have direction:

- directed graph
- undirected graph

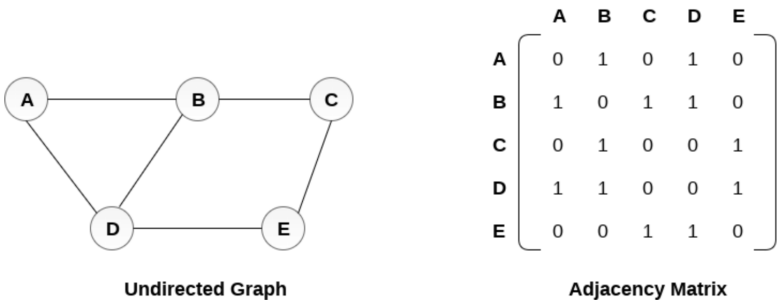
Depends on connectivity:

- disconnected graph - if some node can't be reached by the others nodes.
- connected graph - does not have disconnected nodes

> *Connectivity* - measures the minimum number of elements that need to be removed for graph to be become dissconeted.

Representation of graphs:

1. edge list - 2D list shows connections between nodes [[0,1], [1,2],[1,3], [2,3]]
2. adjacency list [[1], [0,2,3], [1, 3], [1, 2]] for example, second position has index 1 and stores node 1 which has edges to 0,2,3
3. adjacency matrix, where matrix could be considered as 2D array with same lingth lists.

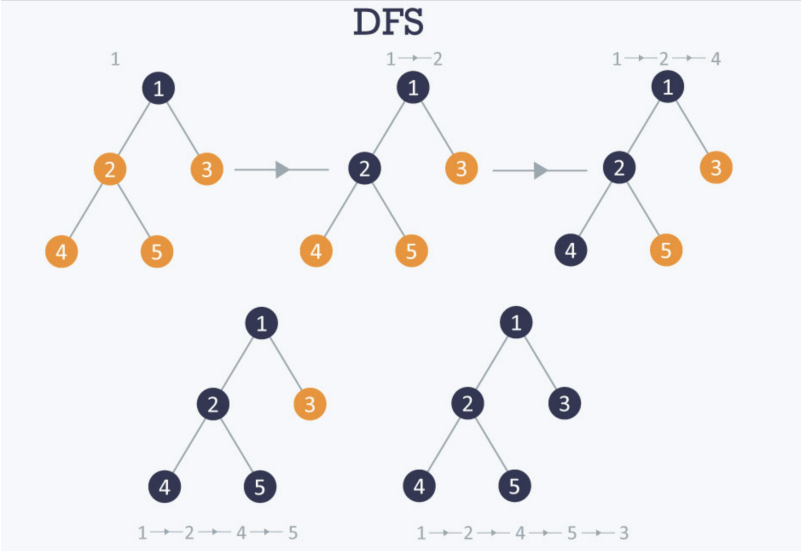


Picture 14. Source: https://www.javatpoint.com/graph-representation (https://www.javatpoint.com/graph-representation)

8.2 Operations

Graph traversal

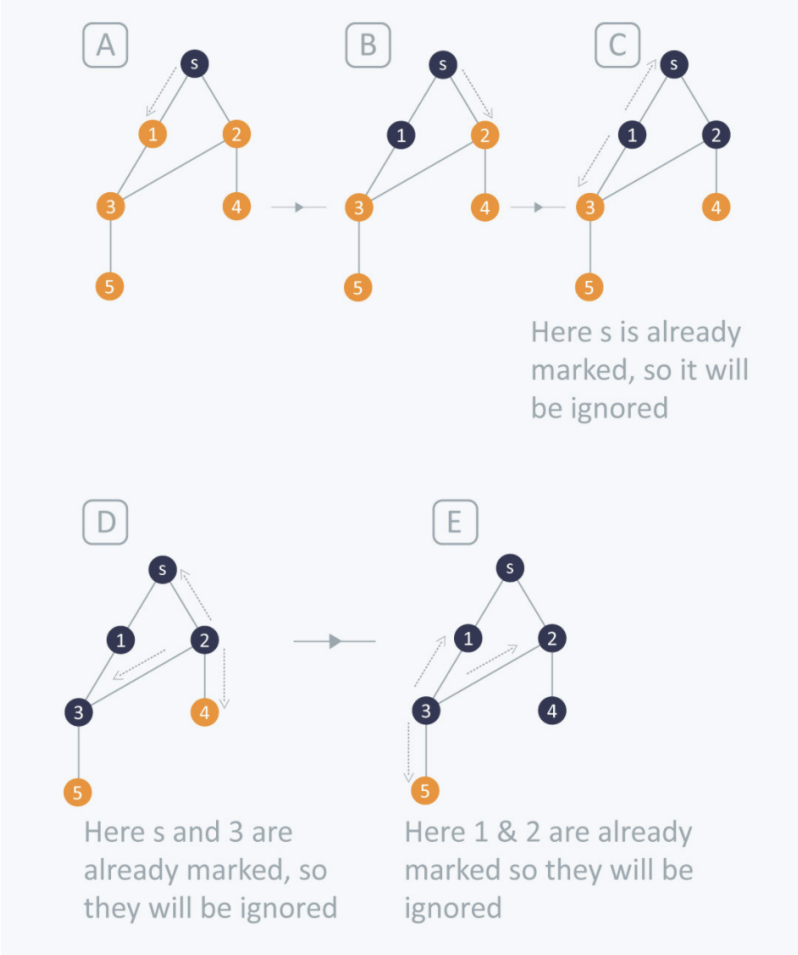
- DFS - depth first search - follow one path till the end. We store seen nodes in stack



Picture 15. Source

Visual representation: https://www.cs.usfca.edu/~galles/visualization/DFS.html (https://www.cs.usfca.edu/~galles/visualization/DFS.html)

- BFS - breadth first search - check all nodes next to the current node



Picture 16. Source

Visual representation: https://www.cs.usfca.edu/~galles/visualization/BFS.html (https://www.cs.usfca.edu/~galles/visualization/BFS.html)

8.3 Time complexity

9 References:

1.