Table of Contents

The goal of this notebook is to document the learning process of different data structures such as arrays, linked lists, stacks, hash maps, queues, trees and graphs. Description, operations, time complexity, advantages and disadvantages and application have been covered for each data structure.

In [1]:

```
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.

1.2 Operations and time complexity

Example array arr = ['elem1', 'elem2', 'elem3', 'elem4', 'elem5'].

Getting element by index is fast operation for array, we just need to specify index, for example for the second element arr[1]. Indexing is an advantage of array.

Searching element in the array is taking linear time, because in order to find an element we need to go through the array starting from the first element.

Adding element to the end of the array is taking constant time. We need just to give to the new element the index, that by 1 greater then the last index of the array.

Inserting an element in the beginning or at any i-th position could be complicated and inefficient and takes linear time or O(n). In this case, we need to shift all other elements one by one.

Deletion has the same problem as insertion, because we get an empty place and need to move other elements of the array so it takes linear time. If we need to do insertions and deletions often it is better to consider other data structures with more efficient time complexity.

Operations	Time Complexity(average and worst)	Comments
add	$O(1)$	fast operation, constant time complexity
delete	$O(n)$	shift all elements which come after the removed one
insert	$O(n)$	shift all element in order to make space
get	$O(1)$	fast operation, constant time complexity
search	$O(n)$	linear time

1.3 Advantages

- indexing: easy access to the elements by index,
- easy to traverse,
- easy searching for element,
- easy to use or implement

1.4 Disadvantages

- static structure, which means fixed size so we need to decide in the beginning how big the array should be. Sometimes the size is planned too big so some positions are unused, which causes inefficient memory usage.
- insertion and deletion are complicated and have linear time complexity.

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 data tables as one-dimensional arrays.

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

In [2]:

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

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

2 Singly Linked List

2.1 Description

Singly linked list is a data structure represented by a list objects (nodes), each node contains data and a single reference to the next node in the list.

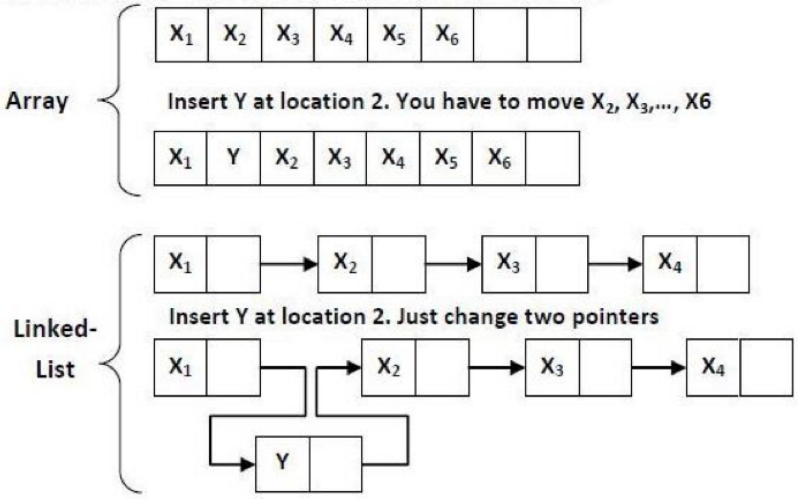
- extension of the list;
- there is order but there are no indexes
- characterized by links: each element has information about the next element

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/)

Picture 1 shows how the structure of linked list looks, each node has data and pointer (reference) to the next node. The pointer goes in one direction.

We can see that insertion at the beginning or any i-th position for linked list takes O(1), because we just need to link new element with the next and previous nodes. The same is happening for deletion operation.

But since linked list does not have indexing, accessing element takes linear time. We need to check the first node and see the address to the next node, then the same for the next and so on. The same behavior with addition, it takes linear time, we need to traverse from the first node to the last.

Operations	Time Complexity	Comments
add	$O(n)$	traversing the list
delete	$O(1)$	link new element with next and previous
insert	$O(1)$	link new element with next and previous
access	$O(n)$	traverse the list
search	$O(n)$	traverse the list

2.3 Advantages

- we don't need to know the size in advance,

- inserting and deleting elements is easier compare to the arrays as it 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,
- no unused memory.

2.4 Disadvantages

- cannot provide fast random access of element,
- more complicated to use and implement,
- extra memory for pointer variables.

2.5 Applications

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

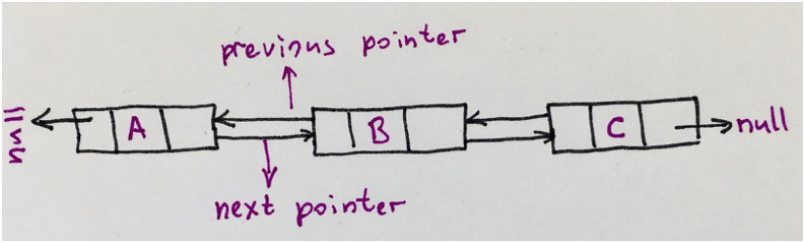
2.6 Code example

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).



Picture 2. Doubly linked lists

3.2 Operations and time complexity

Time complexity is the same as Singly-Linked Lists. Elements are accessed sequentially so there is no direct access.

3.3 Advantages (same as for singly linked list)

- allows traverse in both direction,

3.4 Disadvantages (same as for singly linked list)

- uses extra memory for pointers,

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

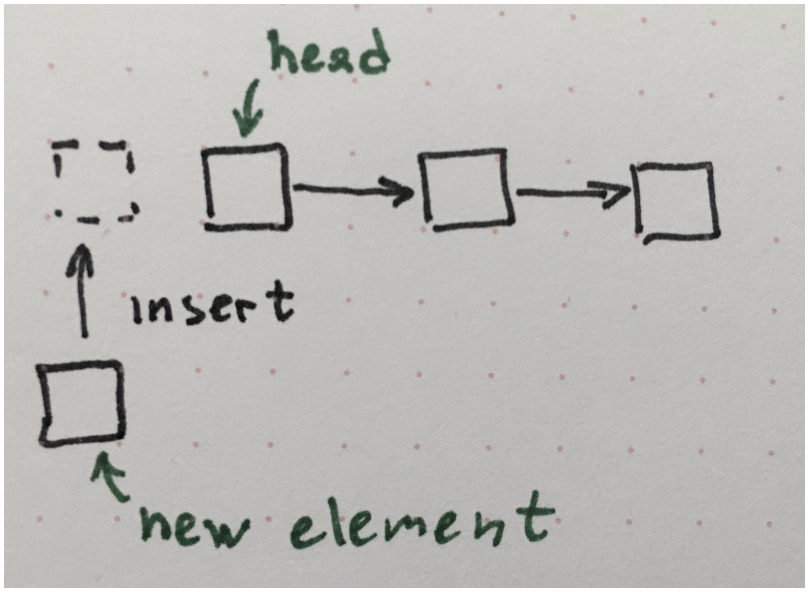
Stack - is a stack of objects when you place objects on top of each other. Useful when we care about the most recent elements or the order in which we save element matter.

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.

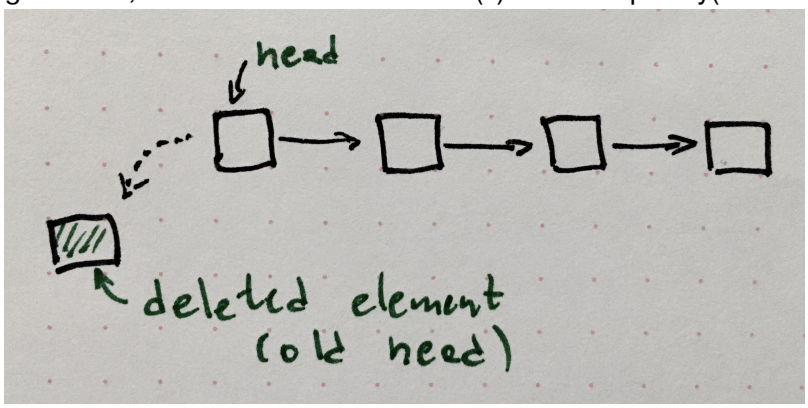
4.2 Operations and time complexity

Elements of stack can be inserted and deleted from one side of the list, called the heard (or top).



Picture 3. Push element to a stack

Push element - is adding element, it takes constant time or O(1) time complexity(Picture 3).



Picture 4. Pop element from a stack

Pop element - take element off the stack, it takes constant time or O(1) time complexity (Picture 4).

Getting (accessing) element or searching element is taking linear time, because we need to go through the whole list one by one.

Operations	Time Complexity (the worst and average cases)	Comments
pop (delete)	$O(1)$	Since the current top position is known
push (add)	$O(1)$	the current position is known
get	$O(n)$	traverse from the first node
search	$O(n)$	traverse from the first node

4.3 Advantages

- it is useful when we care about the most recent elements
- efficient in adding and removing items
- flexible 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()

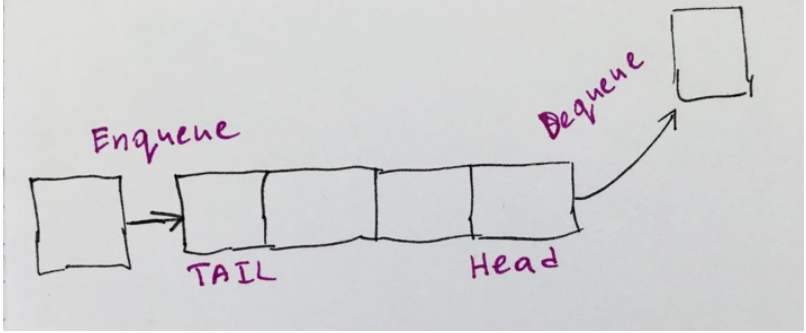
stack.py (data-structure/stack.py)

5 Queue

5.1 Description

Queues - is a linear data structure which follows F.I.F.O. order.

F.I.F.O. - First in first out structure (opposite to stack), the same principle which works with simple queue of people. 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

Elements can be inserted only from one side of the list called tail (or rear), and the elements can be deleted only from the other side called the head (or front).

Enqueue – adding element to the tail is the fast operation and takes constant time.

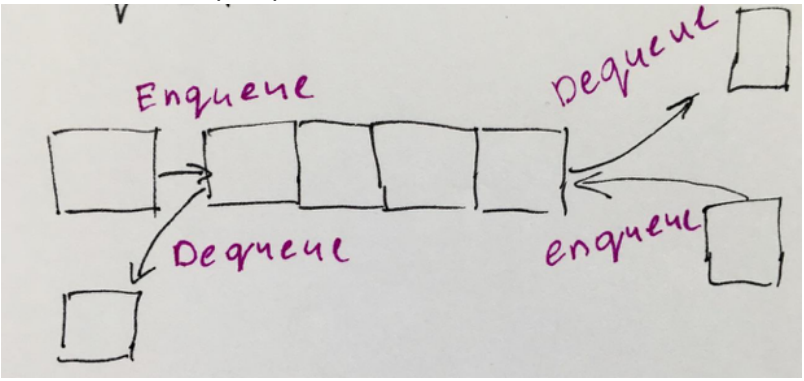
Dequeue – removing head element is also fast operation and takes O(1).

Peek – looking at the head without removing.

> Insertion and deletion in queues takes place from the opposite ends of the list. The insertion takes place at the rear of the list and the deletion takes place from the front of the list. (c) *GeeksforGeeks. Difference between Stack and Queue Data Structures.*

Other types of queue:

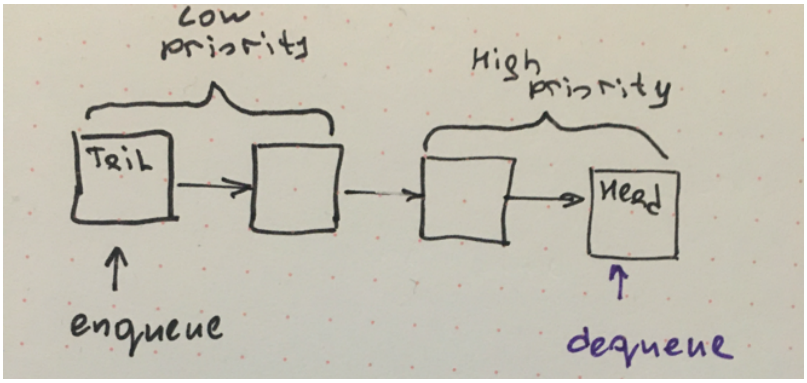
Deque or double-ended queue is the queue that serves both ways. It is not effecting time complexity, so operation takes the same time as for simple queue.



Picture 6. Adding and removing data in dequeue.

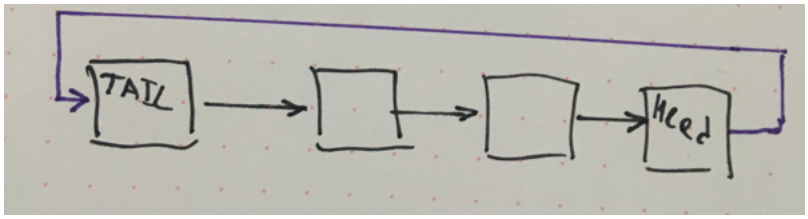
Priority queue – is a special type of queue in which each element is assigned to a priority and is deleting happens according to its priority with the following rules:

- 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 worst (same for average)	Comments
Insertion	$O(1)$	enqueue
Deletion	$O(1)$	dequeue
Get	$O(n)$	traverse through all elements
Search	$O(n)$	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 repetition of elements.

Map is set-base data structure of key-value pairs, where keys are a set, because 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 might be working?

1. take the last 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 - 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 (using chaining). In that case we use closed addressing. Each slot - bucket. Bucket is a more efficient approach, but it is better to have not more than 3 values stored in the bucket.
3. There is also Linear probing technique - if calculated address is occupied, then we use linear search to find the next available slot. In that case we use open addressing.

There is different hashing algorithms in order to find how far we are looking to get an available slot:

- "plus three rehash",
- quadratic probing is when $(failedAttempts)^2$,
- double hashing.

Load factor shows when we need to rehash values.

> *Load Factor = Number of Entries / Number of Buckets.* The closer load factor 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 is time to resize the hash table. In programming we can use automatic rehashing when load factor is getting closer to 0.7.

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).

Goals of hashing function:

- minimize collisions;
- uniform distribution of hash values;
- easy to calculate;
- resolve collisions
  (c) Hash Tables and Hash Functions. Youtube. source

6.2 Operations and time complexity

In the average case, hash tables take constant time - $O(1)$ for any operation. Because when we store key-value pairs, lookup is happening very fast. For insertion operations we can just specify new key and new value. For search, get operations, since we know that keys are the set (unique), we can easily get any value by key. Deletion is similarly needs just a key of value to be deleted. All this operations run with constant time in average cases.

For the worst case the time is linear. The worst case is when we store more data in one index. If we store more data in one index, we use linked list. In this case we perform, for example, get operation by key, we need to go through each item in the list stored with this key. That's why the worst time complexity for insertion, search, deletion is $O(n)$

The main goal is to try to implement hash tables in a way that it uses only average time, which means it has to be without collisions. If we know our data before, we can use perfect hash function to store data without collisions and with good use of space.

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

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
- cashing data instead of making your server do work
- error checking
- program compilation

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

In [3]:

```
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

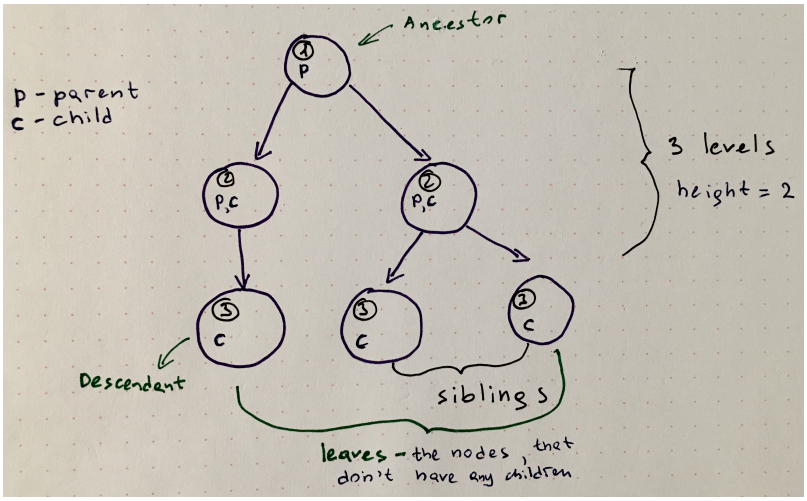
Tree - is non-linear data structure, where data organized hierarchically from the root (kind of linked list extension), it is collection of nodes linked together.

Tree:

- has connected nodes
- has no cycles

We can see on the Picture 9 the basic representation 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.

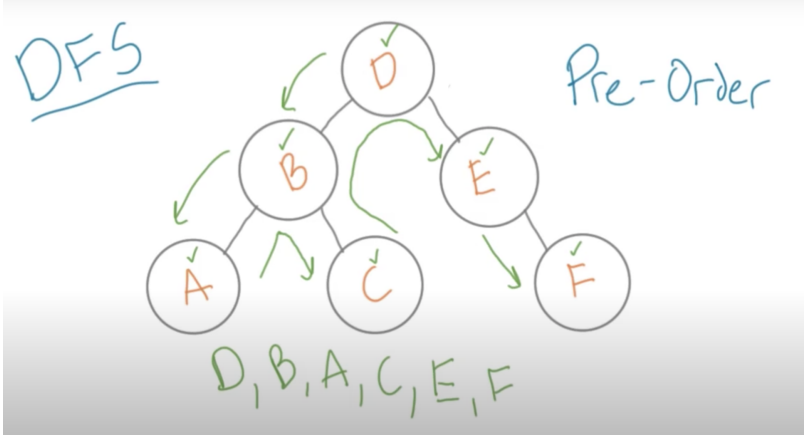
7.2 Tree traversal

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

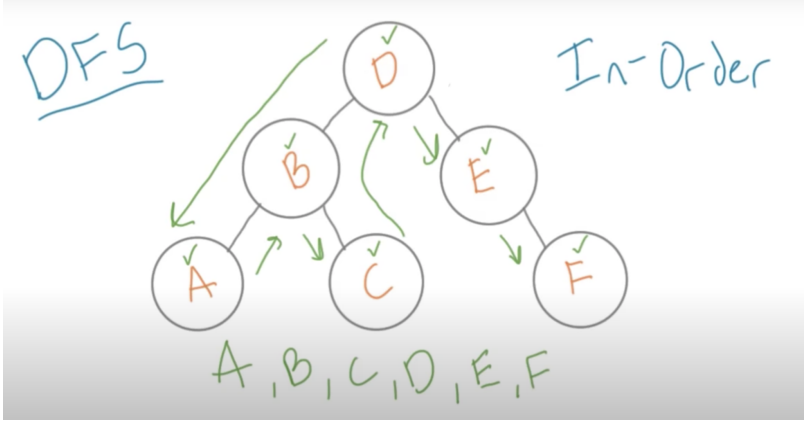
Two approaches for tree traversal:

1. DFS - depth-first search - exploring children nodes is priority:
    - pre-order way - check node before traversing further or node-left-right (Picture 10)



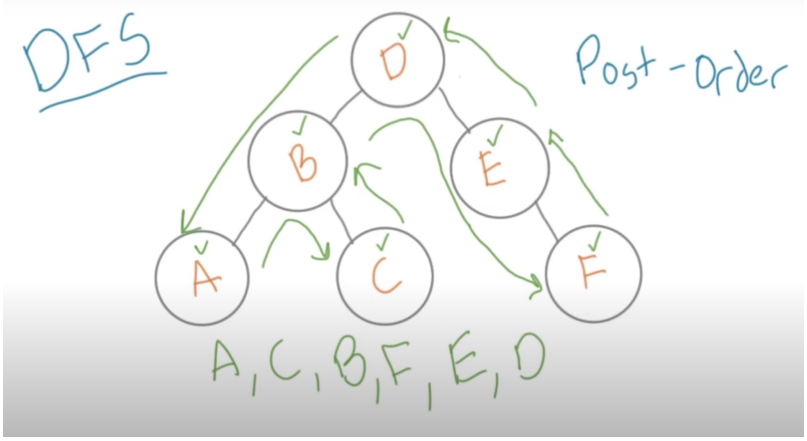
Picture 10. Pre-order DFS screenshot from "Data Structures & Algorithms in Python" Udacity.

- in-order way or left-node-right (Picture 11)



Picture 11. In-order DFS screenshot from "Data Structures & Algorithms in Python" Udacity.

- post-order way (Picture 12)



Picture 12. Post-order DFS screenshot from "Data Structures & Algorithms in Python" Udacity.

2. BFS - breadth-first search - exploring 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 visit all leaves.

7.3 Operations and time complexity of binary search 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) #example of binary tree
```

```
        ________4_____
       /              \
    __7___             _12
   /      \           /   \
  3        _6        10    1
 / \      /  \      /
0   5    11   2    9
```

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 [5]:

```
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
```

In the worst case we can get following right skewed binary search tree:

In [6]:

```
from binarytree import Node

root = Node(1)
root.right = Node(2)
root.right = Node(3)
root.right.right = Node(4)
root.right.right.right = Node(5)
root.right.right.right.right = Node(6)

print(root)
```

```
1
 \
  3
   \
    4
     \
      5
       \
        6
```

It consists of 6 nodes. At each level only one node is present, and it increases height of BST and this is maximum height that BST can have for 6 nodes. We cannot benefit of any BST advantages in this case. Any operation (insertion, deletion, access, search) is taking linear time, since we need to go through each node of element. For example, if we need to find 6 or bigger number, we have to compare each node. Similar with insertion and deletion. Height of BST becomes 'n' and time complexity for this operation is $O(n)$

In the best(average) case, we will get balanced BST:

In [7]:

```
bst_tree = bst(height=3, is_perfect=True)
print(bst_tree)
```

```
          _______7_______
         /               \
     __3__               ___11___
    /     \             /        \
   1       5           9         _13
  / \     / \         / \       /   \
 0   2   4   6       8   10     12    14
```

Height (h) of BST in this case is log(n) (examples of recursion and halving operation has been done in algorithms portfolio) and we can use BST structure benefits. For example if we need to find value 14, it will require following comparisons:

1. 14 and 7, move right
2. 14 and 11, move right
3. 14 and 13, move right

4. 14 and 14, found

For a balanced binary tree the size of the problem to be solved is halved with every iteration which means log(n) iterations are needed to solve a problem of size n.

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

7.4 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 Algorithms and data structure with Python.

Bellow we can see the example of max heap.

In [8]:

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

```
          _________14________
         /                   \
     __13__                 ___12__
    /      \               /       \
   6        10           11         7
  / \      /  \         /  \       / \
 2   0    3    9       5    8     4   1
```

Heapify - reordering the tree, based on heap property. It is the process of rearranging the heap by comparing each parent with its children recursively.

> To add an element to a heap, we can perform this algorithm:
>
> 1. Add the element to the bottom level of the heap at the leftmost open space.
> 2. Compare the added element with its parent; if they are in the correct order, stop.
> 3. If not, swap the element with its parent and return to the previous step. Wikipedia

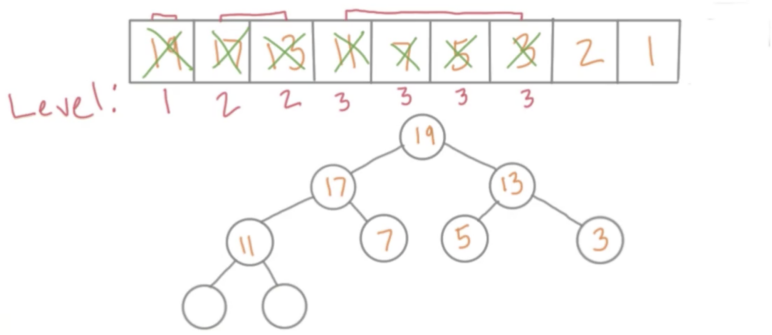
When we insert new element in the any open sport, we do then comparisons with its parent element and swapping until it is ordered. We do not do comparisons with each node therefore we just need to traverse through the tree where height of the tree is $log(n)$ and time complexity in this case is $O(log(n))$

In case deletion of the root, "new root has to be swapped with its child on each level until it reaches the bottom level of the heap, meaning that the delete operation has a time complexity relative to the height of the tree, or O(log n)" Wikipedia

When we delete any node, the number of swaps is going to be still proportional to the number of the levels in the tree or logarithmic, so time complexity in this case is $O(log(n))$

Search operation requires going through every element in the heap in order to determine if an element is inside, therefore time complexity is $O(n)$

Sorted array can be represented as a heap (Picture 13).



Picture 13. Representation of heap in the array from Udacity.

When we build heap, new node always goes in the next empty spot. The basic strategy to build a binary heap of N elements takes $O(n)$ time.

Time complexity for heap:

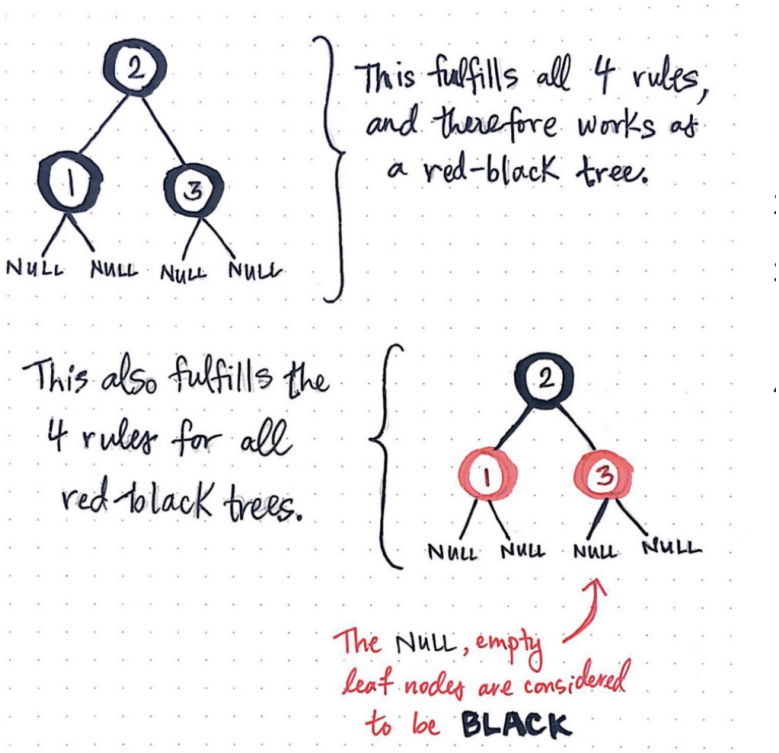
Operations	Time Complexity	Comments
Insertion	$O(log(n))$	We put element in the open spot and then heapify. In the worst case it takes as many operations as height of the tree.
Deletion	$O(log(n))$	Same here, swapping will be done h(hight) times in the worst case scenario.
Get(element)	$O(n)$	Traverse though all nodes
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)

7.5 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 and to follow algorithmic time complexity:

- 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



Picture 14. Red-Black Tree example source

We need to follow the rules above when we do operations of inserting or deleting from the tree. There are good examples of violations of rules in Medium article "Painting Nodes Black With Red-Black Trees" . Also this article has a good explanation of insertion and deletion logic.

So when we are inserting new node, we need always check if it violates any rule by by doing:

- recoloring nodes is a technique for handling insertions and deletions;
- rotations (left-rotation, right-rotation) - is a technique of moving and restructuring subtrees

Operations	Time Complexity worst (same for average)	Comments
Insertion	$O(log(n))$	the maximum height of a red-black tree is the same as a perfectly-balanced binary search tree so it takes logarithmic time, since rotate/recolor takes O(1)
Deletion	$O(log(n)$	same rule
Search	$O(log(n)$	same rule

7.6 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).

7.7 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

7.8 Applications

- storing information hierarchically (file system, folder structure, organization structure)
- heaps are used to implement priority queues
- decision-making processes
- organizing 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

Nodes are also called **vertices**

Edges can have direction:

- directed graph (digraph) - graphs with directions on the edges between nodes.
- undirected graph - graph without any direction on the edges between nodes.

> Some graphs can have edges that contain weight to represent an arbitrary value such as cost, distance, quantity , etc, this type of graphs called **weighted** graph.(c) freeCodeCamp.org. Algorithms Course - Graph Theory Tutorial. YouTube

Depends on connectivity:

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

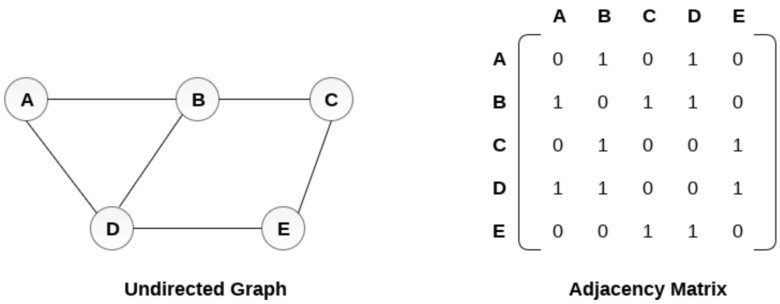
> *Connectivity* - measures the minimum number of elements that need to be removed for graph to be become disconnected. (c) Wikipedia source

One of the worst case (in terms of time complexity) is complete graph, where there is a unique edge between every pair of nodes.

Representation of graphs:

1. edge list - 2D list shows connections between nodes for example [[0,1], [1,2],[1,3], [2,3]]. So we keep keep track of all the nodes and edges is a single list of pairs.

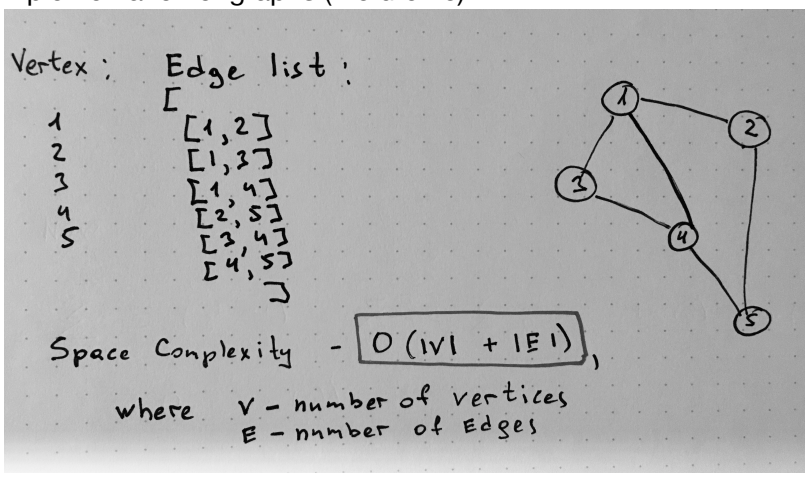
2. adjacency list [[1], [0,2,3], [1, 3], [1, 2]] for example, second position has index 1 and stores node 1 which has edges to 0,2,3
3. adjacency matrix, where matrix could be considered as 2D array with same length lists (Picture 15)



Picture 15. Source: [https://www.javatpoint.com/graph-representation (https://www.javatpoint.com/graph-representation)](https://www.javatpoint.com/graph-representation)

8.2 Time and space complexity of edge list

Edge list is the easiest implementation of graphs (Picture 16).



Picture 16. Edge list representation of graph

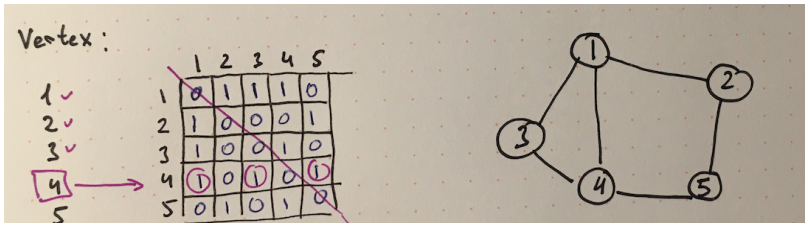
We can see that space complexity of the edge list is $O(|V| + |E|)$.

The most common operation is to find for example if vertex 2 is connected to 4, for this the only way is to iterate through arrays and look for existing pair [2, 4] or [4, 2]. This would take linear time of $O(|E|)$, because we need to check in the worst case all edges.

The same time complexity for another common operation of finding adjacent nodes. It will take $O(|E|)$. This is very costly operation since number of edges could be as squired number of vertices.

8.3 Time and space complexity of adjacency matrix

When we create adjacency matrix representation of graph, we follow the rule: "1" if there edge from one node to another, "0" otherwise (Picture 17).



Picture 17. Adjacency matrix.

We can see that space complexity of an adjacency matrix always takes O(V^2).

In the example (Picture 17), if we need to find the adjacent nodes to "4", we need to go through vertex list and scan related edges,number of which is equal to vertices so the time complexity will be $O(|V|) + O(|V|)$

As it can be seen from picture that finding adjacent nodes (querying) would be very fast and just take constant time.

The same for looking up, adding and deleting an edge can be done with constant time of $O(1)$.

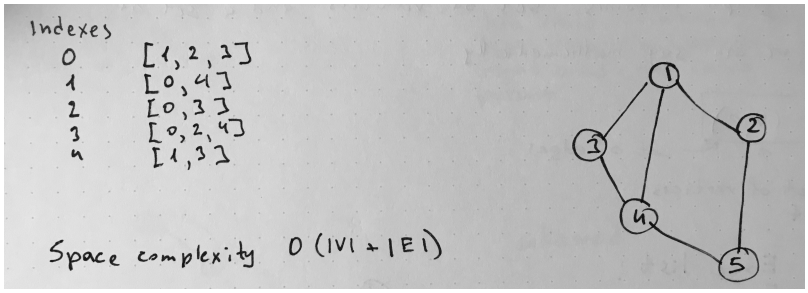
Adding and removing vertex will take O(|V|^2) which is also could be seen from the representation of V*V matrix. In order to add a new vertex the storage must be increased or decreased and achieve this we have to copy the whole matrix.

Operations	Time Complexity
Add vertex	$O(\|V\|^2)$
Delete vertex	$O(\|V\|^2)$
Add edge	$O(1)$
Delete edge	$O(1)$
Query	$O(1)$
Space	$O(\|V\|^2)$

8.4 Time and space complexity of adjacency list

We saw that space complexity is bad for adjacency matrix, because we store information not only where we have connection, but also where we do not have connections with "0".
In adjacency list, we take some advantages from adjacency matrix and some from edge list so we have a hybrid. Adjacency list is an array of linked lists and its shape makes it easy to see which vertices are adjacent to any other vertices so that each vertex can easily reference its neighbor node through a linked list.

The adjacency list representation of the same graph can be seen on Picture 18.



Picture 18. Adjacency list.

In the worst case O(V) is required for a vertex and O(E) is required for storing neighbours corresponding to every vertex which means space complexity is $O(|V| + |E|)$.

> There are two pointers in adjacency list first points to the front node and the other one points to the rear node. Thus insertion of a vertex can be done directly in O(1) time. (c) Geeksforgeeks.org, Comparison between adjacency list and adjacency matrix.

The same for adding an edge, it takes $O(1)$ since we have pointers.

To delete any vertex, first it is required to search for the vertex, it takes $O(|V|)$ time in the worst case. After we find the vertex, we need to traverse the edges it takes $O(|E|)$. So deletion of vertex in total takes $O(|V| + |E|)$.

To delete any edge, we just need to traverse through all the edges, which takes $O(|E|)$.

Operations	Time Complexity
Add vertex	$O(1)$
Delete vertex	$O(\|V\| + \|E\|)$
Add edge	$O(1)$
Delete edge	$O(\|E\|)$
Query	$O(\|V\|)$
Space	$O(\|V\| + \|E\|)$

8.5 Advantages

- with graphs we can represent almost any problem where we have relationships,
- we can use the different graph algorithms to solve it: for example with Breadth-First Search we can find how many connections away I am from some celebrity, or find the best path using Shortest Path Algorithms,
- allow to store complex related information.

8.6 Disadvantages

- memory usage,
- adjacency matrix representation consumes a lot of memory on sparse graphs,
- complex data structure.

8.7 Applications

- social network, where members of the network can be represented as the nodes
- maps where we have cities with distances as weights,
- computer networks,
- internet networks,
- transport networks where stations are vertices and routes are edges,
- visualization of organized data.

9 Resources:

1. Tech With Tim. Array Data Structure Tutorial - Array Time Complexity. YouTube. https://www.youtube.com/watch?v=B2KusJcbVIg (https://www.youtube.com/watch?v=B2KusJcbVIg)
2. GeeksforGeeks. Difference between Stack and Queue Data Structures. https://www.geeksforgeeks.org/difference-between-stack-and-queue-data-structures/ (https://www.geeksforgeeks.org/difference-between-stack-and-queue-data-structures/)
3. HackerRank. Data Structures: Hash Tables. YouTube. https://www.youtube.com/watch?v=shs0KM3wKv8 (https://www.youtube.com/watch?v=shs0KM3wKv8)

4. MIT OpenCourseWare. 8. Hashing with Chaining. YouTube. https://www.youtube.com/watch?v=0M_kIqhwbFo (https://www.youtube.com/watch?v=0M_kIqhwbFo)
5. freeCodeCamp.org. Algorithms Course - Graph Theory Tutorial. YouTube. https://www.youtube.com/watch?v=09_LlHjoEiY (https://www.youtube.com/watch?v=09_LlHjoEiY)
6. Back To Back SWE. YouTube. https://www.youtube.com/c/BackToBackSWE (https://www.youtube.com/c/BackToBackSWE)
7. Data Structures & Algorithms in Python. Udacity. https://classroom.udacity.com/courses/ud513 (https://classroom.udacity.com/courses/ud513)
8. Algorithms. Khan Academy. https://www.khanacademy.org/computing/computer-science/algorithms (https://www.khanacademy.org/computing/computer-science/algorithms)
9. Bhargava, A. (2016). Grokking Algorithms: An Illustrated Guide for Programmers and Other Curious People (1st ed.). Manning Publications.
10. freeCodeCamp.org JavaScript Algorithms and Data Structures.https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/#basic-javascript (https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/#basic-javascript)
11. Painting Nodes Black With Red-Black Trees. Medium. https://medium.com/basecs/painting-nodes-black-with-red-black-trees-60eacb2be9a5#:%7E:text=Thus%2C%20the%20power%20of%20a,nodes%3A%20O(n) (https://medium.com/basecs/painting-nodes-black-with-red-black-trees-60eacb2be9a5#:%7E:text=Thus%2C%20the%20power%20of%20a,nodes%3A%20O(n)).
12. Graph (abstract data type).Wikipedia. https://en.wikipedia.org/wiki/Graph_%28abstract_data_type%29#Representations (https://en.wikipedia.org/wiki/Graph_%28abstract_data_type%29#Representations)