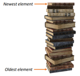
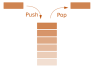
LINEAR STRUCTURES

-- heterogeneous --

STACK (LIFO)

LIFO => Last In, First Out



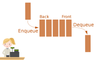


Operation	Stack content	Returned value
s = Stack()	[]	
s.isEmpty()	[]	True
s.push(2)	[2]	
s.push('hello')	[2, 'hello']	
s.push(5.4)	[2, 'hello', 5.4]	
s.size()	[2, 'hello', 5.4]	3
s.peak()	[2, 'hello', 5.4]	5.4
s.pop()	[2, 'hello']	5.4

Note as peak() return the top element but does not modify the stack

QUEUE (FIFO)

FIFO => First In, First Out



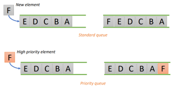
Example

Operation	Queue content	Returned value
q = Queue()	[]	
q.isEmpty()	[]	True
q.enqueue(2)	[2]	
q.enqueue("hello")	['hello', 2]	
q.enqueue(5.4)	[5.4, 'hello', 2]	
q.size()	[5.4, 'hello', 2]	3
q.dequeue()	[5.4, 'hello']	2
q.isEmpty()	[5.4, 'hello']	False

PRITORITY QUEUE

Particular kind of queue. **Each element of the queue has a priority** and the position of the element in the queue depends by its priority.

Highest priority => removed first



DEQUE (double-ended queue)

A double-endend queue where **items can be added and removed at either the front or the rear**. Hybrid between a stack and a queue

Operation	Deque content	Returned value
d = Deque()	[]	
d.isEmpty()	[]	True
d.addRear(2)	[2]	
d.addRear('hello')	['hello', 2]	
d.addFront(5.4)	['hello', 2, 5.4]	
d.size()	[5.4, 'hello', 2]	3
d.removeFront()	[5.4, 'hello']	2
d.removeRear()	['hello']	5.4

-- homogeneous (array) --

Average computational complexity for common operations

Operation	Unordered linked list	Ordered linked list	Python list
isEmpty()	O(1)	O(1)	O(1)
Size()	O(n)	O(n)	O(1)
Add(item)	O(1)	O(n)	O(1)
Remove(item)	O(n)	O(n)	O(n)
Search(item)	O(n)	O(n)	O(n)

LIST (numpy)

Collection of items where each item holds a relative position wrt the others.
Head: first item of the list
End/tail: last item of the list

Ordered list

Operation	Effect
OrderedList()	Creates a new empty ordered list
add(item)	Adds a new item to the list making sure that the order is preserved
remove(item)	Remove the item from the list
pop()	Removes and returns the last item in the list
pop(pos)	Removes and returns the item at position pos
search(item)	Returns *True* if the item is present, *False* otherwise
index(item)	Return the position of the item in the list
isEmpty()	Returns *True* if the deque is empty, *False* otherwise
size()	Returns the number of items in the deque

Each item holds a relative position that is based upon some underlying characteristic of the item

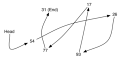
Unordered list

Operation	Effect
List()	Creates a new empty list
add(item)	Adds a new item to the list
append(item)	Adds a new item to the end of the list
insert(pos, item)	Add a new item to the list at position pos
remove(item)	Remove the item from the list
pop()	Removes and returns the last item in the list
pop(pos)	Removes and returns the item at position pos
search(item)	Returns *True* if the item is present, *False* otherwise
index(item)	Return the position of the item in the list
isEmpty()	Returns *True* if the deque is empty, *False* otherwise
size()	Returns the number of items in the deque

LINKED LIST

Unordered list

Each element is connected to the next one until the last item. Must maintain **relative position** of the items (no need of contiguous memory positioning)

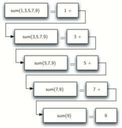


Insert one element at the end of the list => **O(n)**
Insert one element at the beginning of the list => **O(1)**

RECURSION and DYNAMIC PROGRAMMING

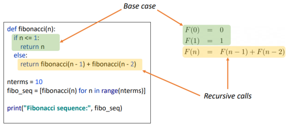
-- recursion --

Problem solving method: breaking a problem in smaller and smaller subproblems until you get enough problem that it can be solved trivially



Law of recursion

1. Must have a <u>base case</u>
2. Must change its state and <u>move toward the base case</u>
3. <u>Call itself recursively</u>



```
def fibonacci(n):
  if n <= 1:
    return n
  else:
    return fibonacci(n - 1) + fibonacci(n - 2)

nterms = 10
fibo_seq = [fibonacci(n) for n in range(nterms)]

print("Fibonacci sequence:", fibo_seq)
```

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n-1) + F(n-2) \end{aligned}$$

```
Fibonacci sequence: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

	Recursion	Iteration
Definition	Recursion is when a function calls itself repeatedly	An iteration is when a loop is repeatedly executed until a certain condition is met
Time complexity	High	(Relatively) Low
Memory	High memory occupation	Low memory occupation
Overhead	There is an extensive overhead due to the repetitive calls of the function	There is no overhead in iteration
Usage	Produces short code. In many case is more comprehensible and similar to the way in which we will solve the problem	Long code, potentially difficult to write. Useful when we have to balance the time complexity against a large code size

NOTE: It is always possible to convert an iterative solution into a recursive solution and vice versa

SEARCH AND SORT

-- search methods --

Sequential (or linear) search

Easiest searching algorithm. Check each element one by one

Computational complexity for an unordered list

	Best case	Worst case	Average case
Item is present	O(1)	O(n)	O(n)
Item is not present	O(n)	O(n)	O(n)

Computational complexity for an ordered list

	Best case	Worst case	Average case
Item is present	O(1)	O(n)	O(n)
Item is not present	O(1)	O(n)	O(n)

Binary search

ONLY FOR ORDERED LIST

Divide and conquer strategy

	WORST CASE	AVERAGE CASE	BEST CASE
Item is present	O(1)	O(log n)	O(log n)
Item is not present	O(log n)	O(log n)	O(log n)

Hashing

Hash function: mapping between an item and the slot where that item belongs. Simple one is remainder between the item and the size of the table

$$h(item) = item \,\%\, len(table)$$

0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

Perfect hash table => no collision (almost impossible in real scenario). Are efficient O(1) only when the number of entries is large

Dinctionary are the build-in implementation of hash table in Python.

-- sorting methods --

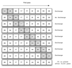
Bubble sort

Computational complexity O(n^2)

Simplest and less efficient sorting algorithms

Adjacent items are compared and swapped if they are out of order. After the first pass the largest element is in the correct position.

Each pass place the next largest value in place



Selection sort

Computational complexity O(n^2) but faster then bubble sort

Similar to bubble sort but it make only one exchange for every pass

With each interaction you select the biggest that has not yet being sorted

When passing the list we look for the largest value, after completing the pass place the largest value in the proper location

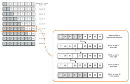
Insertion sort

Computational complexity O(n^2) but faster then selection and bubble sort

The algorithm maintains a sorted sub-list in the lower positions of the list

The algorithm maintains a sorted sub-list in the lower position of the list. Each new item is inserted into the sub-list as:

- Items in the sorted sub-list that are greater than the current item are shifted one position to the right.
- When we reach a smaller item or the end of the sub-list the new item can be inserted

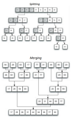


Merge sort (recursive)

Computational complexity O(n log n), better then bubble, selection and insertion

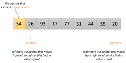
Divide and conquer strategy. Works recursively by dividing the list to be sorted in half

- If the list is empty or has one item, it is sorted (**base case**)
- If the list has more than one item, the algorithm splits the list in half and recursively invoke a merge sort on both halves
- Once the two halves are sorted, the **merging** is performed
- **Merging** consists in taking two smaller sorted lists and combining them together into a single sorted new lists



Quick sort (recursive)

- Select a value in the list called **pivot**
- Apply the **partition** process
  - Rearrange the list until the pivot reach a position (**split point**) in which all the elements less than the pivot will be at its left and all the elements greater that the pivot will be at its right
  - Split the list at the split point to obtain two smaller sub-lists
- Recursively call the previous steps to sort the sub-lists

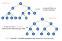


When rightmark < leftmark: change the pivot value with the rightmark

TREES

-- Binary tree --

A node can have at most two child (left and right child)



Definition

Height Balanced
If the height of the left and right subtree of any node differ by not more than 1

Perfectly Height Balanced
If the left and right subtree of any node have the same height

Complete Binary Tree
If all its levels are completely filled, except eventually the last one which is filled from left to right

Tree Traversal

Preorder
ONE TOUCH to be enlisted

Inorder
TWO TOUCH to be enlisted (if leaf node then enlisted right away)

Postorder
THREE TOUCH to be enlisted (if leaf node then enlisted right away)

Level order (breadth-first)
Node are visited level-by-level

Binary Heap

Is a binary tree that is:

1. **Complete** (all its levels are completely filled, except eventually the last one which is filled from left to right)
2. Is **min/max heap** each node has a key smaller/greater than its children's keys

They can be implemented in Python using a list. Make it easy to find the position of parent/child. First element never used (allows to use integer division when computing the index)

Min (or max) heapify

Process of reshaping a (complete) binary tree in a min or max heap

1. Load the list at it is
2. Rearrange al the node starting from the middle of the tree

Insert new element (min heap)

1. Add the new element at the end of the list (left before right)
2. If the new element is less than its parent swap their position
3. Repeat the swap until the new element is in the right position

Extract the min element (min heap)

1. Remove the root node (that is the min element)
2. Move the last node to root
3. Restore the heap order by pushing the new root node down the tree to its proper position (heapify down)

We want to preserver the shape of the min heap. That's why we took the last element in the list (27) that we put on the root.

Also we swapped with 9 and not with 11 because it must stay a MIN HEAP and so the root node needs to be as small as possible. Again we swapped 27 with 14 because 14<18 and so on

Binary Search Tree (BST)

Is a binary tree in which the nodes are ordered so that

- Nodes with keys that are less than the parent key are in the left subtree
- Nodes with keys that are greater than the parent key are in the right subtree

Search, insert, delete operations on average **O(log n)**

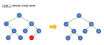
1. Ordered, or sorted, binary trees.
2. Nodes can have 2 subtrees.
3. Items to the left of a given node are smaller.
4. Items to the right of a given node are larger.

ADD A NODE

Just go left it it smaller, right if it bigger

DELETE A NODE

1. Delete a leaf node



2. Delete a node with only 1 child

3. Delete a node with 2 children

We must search among the children the node that will preserve the binary search tree relationship for both the existing left and right subtree

use the node that has the **next-largest key** in the tree and it called successor node

Basically first **go right** and **then go left** as much as possible

If the right child of the node to be removed has no left child, the right child is the successor

If the right child of the node to be removed has a left child, the successor is always the left most node

Self-Balancing-Search Tree

Particular type of search tree that keeps its height small after an insertion or deletion.

Such as: Red-Black tree, AVL Tree...

- AVL trees can be seen as a particular type of Red-Black tree that is more "strictly" balanced, at expense to a higher number of rotations during insertion and deletion
- Red-Black trees should be preferred if in our current application we have frequent insertions and deletions
- AVL tree should be preferred search is a more frequent operation then insertion and deletion

AVL TREE

Adelson-**V**elsky and **L**andis. Make sure a tree remains balanced after insertion or deletion.

The difference between the left and right subtree heights can not be more than one
Each node have a **balanced factor (BF)** = *height(leftSubTree) - height(rightSubTree)*

When a new node is added/removed we update the balance factor of all the nodes

Left rotation (BF < -1)

1. Promote the right child (B-10) of the unbalanced node (A-5) to be the new root node
2. Move the old unbalanced root (A-5) to be the left child
3. If (B-10) had a left child (8) make it the right child of the old root (A-5)

Right rotation (BF > 1)

1. Promote the left child (C-5) of the unbalanced node (E-10) to be the new root node
2. Move the old root (E-10) to be the right child of the new root (C-5)
3. If the new root (C-5) already had a right child (D-8), then make it the left child of the old root node (E-10)

Complex case

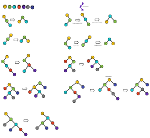
- Node A has BF=-2 so, in theory, we had to perform a left rotation to balance the tree

But it is still unbalanced

- The correct solution in this case is to first perform a right rotation of the right subtree, then a left rotation on the new tree

Complete rules for re-balancing an AVL tree:

1. Left case: If a node has BF = -2 and its right child has BF = -1 or 0 perform a left rotation
2. Right case: If a node has BF = 2 and its left child has BF = 0 or 1 perform a right rotation
3. Right-Left case: If a node has BF = -2 and its right child has BF = 1 perform a right rotation of the right child followed by a left rotation
4. Left-Right case: If a node has BF = 2 and its left child has BF = -1 perform a left rotation of the left child followed by a right rotation



RED-BLACK TREE

Self-balancing binary tree where each node has also a color (either black or red). It have "virtual" leaf nodes (called NIL) that do not contain any data. Average complexity of **O(log n)**

Criteria

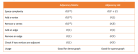
1. A node is either red or black
2. The root and leaves (NIL) are black => (newly inserted nodes always start as red)
3. If a node is red, then its children are black
4. All paths from a node to its NIL descendants contain the same number of black nodes (shortest: all black, longest: black and red alternating, twice as long)

GRAPH

Operations of Graph ADT

Operation	Effect
Graph()	Creates a new empty graph
addVertex(vert)	Adds an instance of Vertex to the graph.
addEdge(fromVert, toVert)	Adds a new, directed edge to the graph that connects two vertices
addEdge(fromVert, toVert, weight)	Adds a new, weighted, directed edge to the graph that connects two vertices.
getVertex(vertKey)	Finds the vertex in the graph named *vertKey*
getVertices	Returns the list of all vertices in the graph
in	Returns *True* if the given vertex is in the graph, *False* otherwise

-- How to implement a graph --



Adjacency matrix

Each row and columns represents a vertex, the value in the cell indicate the weight. If no wait is empty or 0 (aka no edge). When two vertices are connected by an edge we say that they are adjacent

Pros
- Easy to read
- Good representation for dense graphs (lots of edges/connections)
- O(1) time for adding/removing an edge and to verify existence

Cons
- High memory usage O(V^2)
- Very inefficient representation for graph with few edges/connections
- Adding or removing a vertex requires O(V^2= time

Adjacency list

We keep a master list of all the vertices in a graph object Each vertex object in the graph maintains a list of the other vertices that it is connected to (plus the weight if present)

Pros
- Memory efficient, O(|V| + |E|) space complexity
- Good for representing sparse graphs
- O(1) time for adding a vertex or an edge

Cons
- Not efficient for dense graph
- Removing an edge or verify if it exists requires O(|V|) time

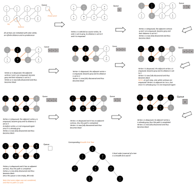
FROM ONE SOURCE TO ALL DESTINATIONS (for unweighted graphs)

-- Breadth First Search (BFS) --

- BFS finds all the vertices that are at distance k from s before it finds any vertices that are at distance k+1
- It computes the **shortest path** (smallest number of edges) from s to each reachable vertex
- It produces a **breadth first tree** with root s that contains all the reachable vertices

BFS use a Queue to decide which vertex to explore next

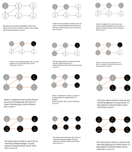
Colors
- **WHITE**: not yet discovered (each vertex is initialized as white)
- **GRAY**: discovered (aka visited for the first time) but not yet explored. Each adjacent vertices can be both white or black
- **BLACK**: completely explored (no adjacent white vertices)



-- Depth First Search (DFS) --

Explore the graph as deeply as possible along each branch before backtracking
- When DFS reaches a vertex, it marks it as visited, then moves to an adjacent unmarked vertex and continues recursively until there are no more adjacent unmarked vertexes
- DFS backtracks and check different edges for finding other unmarked nodes following the same recursive procedure
- The process continues until all the vertexes have been visited

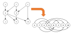
DFS keeps track of two timestamps for each vertex
- **Discovery time**: records the number of steps in the algorithm before a vertex is first encountered
- **Finish time**: record the number of steps in the algorithm before a vertex is fully explored



-- Topological sorting --

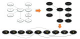
Takes Direct Acyclic Graph (DAG) and produces a linear ordering of all its vertices.

On the same graph multiple topological sorts are possible but the first vertex must always be a vertex with no incoming edges



Procedure
1. Run DFS to compute the finish times for each vertex
2. Store the vertices in stack in decreasing order of finish time
3. Return the stack and pop all the elements to obtain the result of the topological sort

Example: how to make pancakes

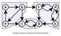


-- Strongly connected components --

Strongly connected

Every vertex is reachable by any other vertex (exist a path from each vertex to each other vertex in the graph)

Strongly connected component of a direct graph G is a sub-graph of G that is strongly connected



-- Kosaraju's algorithm --

1. Run DFS on the graph G to compute the finishing time of each vertex
2. Compute the graph G^t the transposition of G (reverse the direction of all edges)
3. Run DFS on G^t but consider the vertexes in order of decreasing finishing time as computed in step 1
4. Each tree in the forest computed in step 3 is a strongly connected component

-- Minimum spanning tree --

Given an undirected graph a **spanning tree** is a acyclic subgraph that connects all the vertices of the graph. (A graph can have multiple spanning tree)

Minimum spanning tree: spanning tree in which the sum of all the edge weights is minimized

-- Shortest Path --

Shortest path: from a vertex u to a vertex v is a path in which the sum of the weight of its constituent edges is minimized

-- Prim's algorithm --

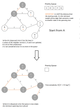
Used to compute the minimum spanning tree of a graph. A **Greedy algorithm** (at each step it adds to the tree the safe edge with the lowest weight)

Start from a node and add it to the visited list. Then add the node that have the lowest weight, and keep repeating (chooses the smallest from all the visited notes) until all the nodes are all visited/connected

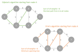
While the tree T is not yet a spanning tree:
find edges that are "safe" to add to T
Add to T the safe edge with the lowest weight

An edge is safe it it connects a vertex that is already in the spanning tree to a vertex that is not in the spanning tree (avoid cycles)

Use a **priority queue** to keep track of the vertices not yet added



- Both Prim's and Dijkstra's algorithms derive from Breadth First Search
- Prim's algorithm works with both positive and negative weights, while Dijkstra's work only with positive ones
  - NOTE: to get the shortest path in a graph with both positive and negative weights use the Bellman-Ford's algorithm
- Dijkstra's works with both directed and undirected graphs, while Prim's works only with undirected ones
- Both the algorithms are greedy, but they minimize different things:
  - Prim's algorithm minimizes the total weight of the spanning tree
  - Dijkstra's algorithm minimizes the path from a source to a destination
- If it reaches all nodes, also Dijkstra's creates a spanning tree, but it is not guarantees that it is a minimum spanning tree



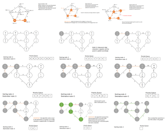
FROM ONE SOURCE TO ALL DESTINATIONS (for weighted graphs without negative weights)

-- Dijkstra's algorithm --

Provides shortest path from a starting vertex to all other reachable vertices on a weighted graph. **Commonly used to find the shortest path from a vertex A to a vertex B**

Is a **greedy algorithm** since at each step it chooses the path with lower distance from the source

Work similar to Breadth First Search (BFS) keeping track of the distance of each vertex from the starting vertex s
- In BFS the distance of a vertex is the **minimum number of edges from s**
- In Dijkstra the distance of a vertex is the **total weight of the smallest weighted path from s**



FROM ONE SOURCE TO ONE DESTINATION (for weighted and unweighted graphs)

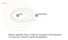
-- A* search algorithm --

Extension of Dijkstra's algorithm that uses a heuristic to efficiently find the shortest path from a source to a destination in a graph. Is **Optimally efficient**: there is no other optimal algorithm that is guaranteed to expand fewer nodes than A*

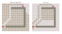
- At each iteration Dijkstra choose the path with the lowest cost from the source node
- At each iteration A* selects the path that minimizes the following equation *f(n) = g(n) + h(n)*
  - n is a possible next node on the path
  - g(n) is the cost of the path from the starting node to n
  - h(n) is a heuristic function that estimates the cost from n to the destination node

Heuristic function is problem specific. There is no function optimal in all the cases
Exact heuristic: function that returns for each node its exact distance to the destination. Extremally time consuming, approximation are generally food and faster than Dijkstra's algorithm

Will find the shortest path only if the heuristic function is **admissible**, an overestimated (non-admissible) heuristic can lead to find suboptimal paths and do unnecessary exploration



Dijkstra's algorithm checks "uniformly" the graph in all the directions
A* moves more "directly" towards the destination



Dijkstra's algorithm checks "uniformly" the graph in all the directions
A* moves more "directly" towards the destination
NOTE: this is only a graphical representation, the numbers are not weights