Analysis of Algorithms

Complexity of an algorithm = computational resources it consumes (execution time and memory space).

When analyzing an algorithm we investigate its complexity properties of this algorithm. To analyze an algorithm we use 3 asymptotic notations, that allow us to classify the functions "in long-term", for large values of n, where n is the size of the input function.

- $O(n)$ → Asymptotic upper bound (Big-Oh) → f grow no faster than $O(n)$
- $\Omega(n)$ → Asymptotic lower bound → f grow at least as fast as $\Omega(n)$
- $\Theta(n)$ → Asymptotically tight bound → f grow with the same rate of $\Theta(n)$

Rate of Growth

$\log n < \sqrt{n} <$	$n <$	$n \log n <$	$n^2 <$	$n^3 <$	k^n	$\Theta(1)$
Logarithmic	Linear	Quasi-linear	Quadratic	Cubic	Exponential	Constant

Properties (f and g are functions):

- The cost of an elementary operation is $\Theta(1)$, this include:
  - An assignment of basic types (`int, bool, double, …`)
  - An increment or decrement of a variable of basic type
  - An arithmetic operation (addition, subtraction, multiplication and division)
  - A read or write of basic type
  - A comparison between basic types
  - The acces to an element of a vector
- Additions of costs: $\Theta(\mathrm{f}) + \Theta(\mathrm{g}) = \Theta(\max\{f, g\})$
- Product of costs: $\Theta(\mathrm{f}) \cdot \Theta(\mathrm{g}) = \Theta(\mathrm{f} \cdot \mathrm{g})$
- For a constant $c > 0 \rightarrow O(f) = O(c \cdot f)$
- Pass a parameter by value → $\Theta(\mathrm{n})$ Because the parameter is copied
- Pass a parameter by reference (&) → $\Theta(1)$
- Changing the basis of a logarithm $\log_c x = \frac{\log_b x}{\log_b c}$

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0 \Rightarrow g \in O(f) \land g \notin \Omega(f)$$

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = c \Rightarrow g \in \Theta(f) \Rightarrow g \in O(f) \land g \in \Omega(f)$$

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty \Rightarrow g \notin O(f) \land g \in \Omega(f)$$

Analysis of iterative algorithms

The cost of an elementary operation is $\Theta(1)$

`if B then S1`
`else S2`
	worst case → $O(\max\{f + h,\ g + h\})$	The cost of `B` is h
		The cost of `S1` is f
		The cost of `S2` is g

`while B {`
`    S`
`}`
	If $p = \max\{f + h\} \rightarrow T = O(p \cdot g)$	The cost of `B` is f
		The cost of `S` is h
		g = number of iterations

Analysis of recursive algorithm

$$T(n) = a \cdot T(n-c) + f(n) \rightarrow T(n) = \begin{cases} \Theta\left(n^k\right) & If\ a < 1 \\ \Theta\left(n^{k+1}\right) & If\ a = 1 \\ \Theta\left(a^{\frac{n}{c}}\right) & If\ a > 1 \end{cases}$$

Where a is the number of recursive calls, $c \geq 1$ is the number that we subtract from the parameter we pass in the recursive call and $f(n) = \Theta\left(\mathrm{n}^{\mathrm{k}}\right)$ is the cost of the non-recursive part of the algorithm.

Examples: if $f(n) = \Theta(1) \rightarrow n^k = 1 \rightarrow k = 0$, if $f(n) = \Theta(\mathrm{n}) \rightarrow n^k = n \rightarrow k = 1$

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) \rightarrow T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right) & If\ \ k < \log_b a \\ \Theta\left(n^k \cdot \log n\right) & If\ k = \log_b a \\ \Theta\left(n^k\right) & If\ k > \log_b a \end{cases}$$

Where $a \geq 1$ is the number of recursive calls, $b > 1$ is the number we divide to the parameter we pass in the recursive call and $f(n) = \Theta\left(\mathrm{n}^{\mathrm{k}}\right)$ is the cost of the non-recursive part of the algorithm.

Recurrence can also be expressed as follows:

$$T(n) = \begin{cases} base\ case & si\ 0 \leq n < n_0 \\ subtract\ or\ divider\ recurrence & si\ n \geq n_0 \end{cases}$$

Divide and Conquer

Divide and conquer is a strategy that solves a problem in 3 steps:

1. Dividing the principal problem into subproblems (smaller instances of the same type of problem)
2. Recursively solving the subproblems
3. Combining the answers in the right way

Divide and conquer algorithms

MergeSort → Always: $\Theta(\mathrm{n} \log n)$

1. Divide the input sequence in two halves.
2. Sort each half recursively
3. Merge the two sorted sequences

```
void merge(vector<elem>& T, int l, int m, int u) {
   vector<elem> B(u - l + 1);
   int i = l, j = m + 1, k = 0;
   while (i <= m and j <= u) {
      // If the element of the lower part of the sequence is smaller than the
      // upper one then add the element of the lower part to the sequence
      if (T[i] <= T[j]) B[k++] = T[i++];
      // Otherwise add the element of the upper part to the sequence
      else B[k++] = T[j++];
   }
   // Add to the new sequence B the elements that hasn't been added yet
   while (i <= m) B[k++] = T[i++];
   while (j <= u) B[k++] = T[j++];
   // Overwrites the part of the sequence that has been processed
   for (k = 0; k <= u - l; ++k) T[l + k] = B[k];
}

// Initial call: mergesort(T, 0, T.size() - 1)
void mergesort(vector<elem>& T, int l, int u) {
   if (l < u) {
      int m = (l + u) / 2;
      mergesort(T, l, m);        // Sort the half downwards positions
      mergesort(T, m + 1, u);    // Sort the half upwards positions
      merge(T, l, m, u);         // Merge the two sorted sequences
   }
}
```

QuickSort → $\Omega(\mathrm{n} \log n)$, $\mathrm{O}(n^2)$ and on average $\Theta(n \log n)$.

1. Chooses an element from the list that will act as a pivot.

   a. If pivot is the first element of the list → If the input is sorted the algorithm loses time $\Theta(n^2)$.
   b. If pivot is a random element → On average it splits the problem into similar subproblems, but it does not always make the algorithm faster.
   c. If pivot is the median of three elements (because doing the median of everything would be very expensive) → It makes a good estimate, usually the first, last and middle element is chosen.
2. Make two groups
   a. Elements greater than the pivot go to one side.
   b. Elements smaller than the pivot to the other side, elements equal to the pivot can go to either side.
3. It makes two recursive calls dividing the list in two (major and minor).
4. Repeats the process for each sublist.
5. Returns the two ordered sublists in a row.

```
int partition(vector<elem>& T, int l, int u) {
   int x = T[l]; // In this case we take the first element as pivot
   int i = l - 1;
   int j = u + 1;
   while (1) {
         while (x < T[--j]);
         while (T[++i] < x);
         if (i >= j) return j;
         // If there is an element in the lower and upper part that is not
         // in the correct part, exchange them.
         swap(T[i], T[j]);
   }
}

// Initial call: quicksort(T, 0, T.size() - 1);
void quicksort(vector<elem>& T, int l, int u) {
   if (l < u) {
         int q = partition(T, l, u);
         // Elements smaller than the pivot
         quicksort(T, l, q);
         // Elements bigger than the pivot
         quicksort(T, q + 1, u);
   }
}
```

BinarySearch → Return the position of a number in an array in increasing order → $\Omega(1)$, $O(\log n)$ and on average $\Theta(\log n)$.

1. Get the number of the middle position of all the array
2. Check if the number of the step 1 is bigger or lower than the number we are searching
   a. If the number is bigger it will take the half upwards positions
   b. If the number is lower it will take the half downwards positions
3. Repeat the same process until find the number that we are searching. If the number that we are looking for it’s not in the array, return the closest above number.

```
// Initial call: binarySearch(T, x, 0, T.size() - 1)
int binarySearch(const vector<elem>& T, const int& x, int l, int u) {
   if (l > u) return u;      // The element is not in the array

   int m = (l + u) / 2;
   if (x == T[m]) return m;  // The position of the number has been found
   else if (x < T[m]) return binarySearch(T, x, l, m - 1);     // [----]------
   else return binarySearch(T, x, m + 1, u);                   // -----[-----]
}
```

QuickSelect → Find the $j-th$ smallest element in a given set not ordered → $\Omega(\text{n})$, $O(n^2)$ and on average $\Theta(n)$.

1. Take the middle position element in the set as a pivot (The first time is the first element).
2. All the elements bigger than pivot go to the right side of the set.
3. Take other element of the left side of the set as a pivot.
4. Repeat the process while pivot $k > j$.

```
// Initial call: quickSelect(T, 0, j, T.size() - 1)
int quickSelect(vector<int> T, int l, int j, int u) {
   if (l == u) return T[l];

   int k = partition(T, l, u);    // Same function as quickSort algorithm
   if (k == j) return T[k];       // The j-th smallest element has been found
   else if (j < k) return quickSelect(T, l, j, k - 1);   // [------]-----
   else return quickSelect(T, k + 1, j, u);              // -------[----]
}
```

InsertionSort and BubbleSort → $\Omega(n), O(n^2)$ y on average $\Theta(n^2)$.

Karatsuba's Algorithm → Computes the product of two natural numbers of *n* bits → $\Theta\left(n^{\log_2 3}\right)$.

1. Let suppose that x and y are two natural numbers, we pass them to binary and we x and y in half of n bits, if x and y doesn't have the same numbers of bits (n) or n is not a power of 2 → add 0's.
   a. $x = [x_I][x_D] = 2^{n/2}x_I + x_D$
   b. $y = [y_I][y_D] = 2^{n/2}y_I + y_D$

   $x = \begin{array}{|c|c|} \hline 01\ldots & 11\ldots \\ \hline \end{array}$ with x_I under the first half and x_D under the second; $y = \begin{array}{|c|c|} \hline 10\ldots & 10\ldots \\ \hline \end{array}$ with y_I and x_D beneath.
2. Compute $x \cdot y = 2^n x_I y_I + 2^{n/2}\,(x_I y_D + x_D y_I) + x_D y_D$

Exponenciación rápida → $\Theta(\log n)$.

```
double fast_exponentiation(double x, int n) {     // Computes x^n
   if (n == 0) return 1;

   double y = exponential (x, n / 2);
   if (n % 2 == 0) return y * y;
   else return y * y * x;
}
```

Strassen's Algorithm → computes the product of two matrices of size $n \times n$ → $\Theta\left(n^{\log_2 7}\right)$.

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} = M_2 + M_3 & C_{12} = M_1 + M_2 + M_5 + M_6 \\ C_{21} = M_1 + M_2 + M_4 - M_7 & C_{22} = M_1 + M_2 + M_4 + M_5 \end{pmatrix}$$

$$M_1 = (A_{21} + A_{22} - A_{11}) \cdot (B_{11} + B_{22} - B_{12})$$
$$M_2 = A_{11} \cdot B_{11}$$
$$M_3 = A_{12} \cdot B_{21}$$
$$M_4 = (A_{11} - A_{21}) \cdot (B_{22} - B_{12})$$
$$M_5 = (A_{21} + A_{22}) \cdot (B_{12} - B_{11})$$
$$M_6 = (A_{12} - A_{21} + A_{11} - A_{22}) \cdot B_{22}$$
$$M_7 = A_{22} \cdot (B_{11} + B_{22} - B_{12} - B_{21})$$

Dictionaries

A dictionary is a data structure formed by different keys, which are used as an identifier for a piece of information or value. These keys are unique, they cannot be repeated.

Element = (key, information)

The operations allowed are:

- Assign: add an element (key, information) to the dictionary. If an element with the same key existed before, information is overwritten.
- Delete: given a key, the element with that key is removed. If there is no such element, nothing is done.
- Present: given a key, it returns a boolean indicating whether the dictionary contains an element with that key.
- Search: given a key, it returns a reference to the element with that key.
- Consult: given a key, it returns a reference to the information associated with that key.
- Size: it returns the number of pairs (key, information) in the dictionary

Next we will see the different data structures that implement the dictionaries:

Direct-access tables (a vector): Are structures that each position of the vector corresponds to a key, so it gains in efficiency, since the worst case operations will be $\Theta(1)$, but we sacrifice space.

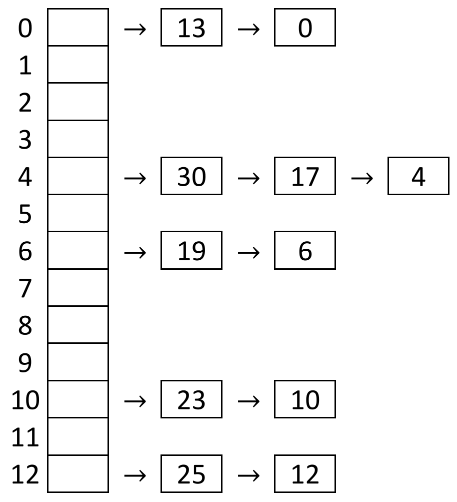
Hash tables: are efficient structures for implementing dictionaries. → $O(n)$ and on average $\Theta(1)$ (Under reasonable hypotesis). Suppose we have at most n keys and declare a table T of m positions, where $m \leq n$.

- If $m = n$, we use direct access: the element of the key k goes to the space $T[k]$.
- If $m < n$, we use scattering tables: the element of the key k goes to the space $T[h(k)]$, where h is a scattering function and tells us among the m positions where the key k will be.

A method of dispersion function is $h(k) = k \bmod m$

We call **collisions** to two keys that coincide in the same space, this is due to the fact that there are less positions than keys, therefore it is inevitable, although ideally there should be as few collisions as possible. There are different methods to deal with collisions:

- Separate Chaining: the solution to chaining collisions is solved by making each table entry contain a chained list of the different scatter keys. In this way the scattering function h indicates where each key should go.



The cost per query:

- Worst-case $\Theta(n)$: assuming all elements are at the same scatter value, forming a single list.
- On average $\Theta(1)$: because the scatter function takes $\Theta(1)$ and then returns a list key.

Assign and delete (with prior search), in the worst case has cost $\Theta(n)$.

The **load factor**: indicates the average number of elements per position $\alpha = n/m$. Thus we know that the average cost per search, assignment or deletion (both with prior search) is $\Theta(1+\alpha)$.

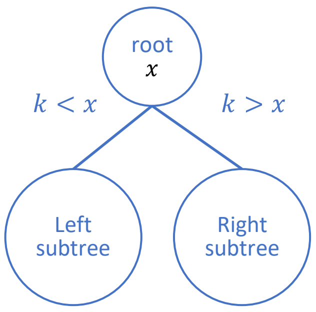
- **Open Addressing**: All elements are stored in the scatter table, so the table will always be greater than or equal to the number of existing keys (the table can increase in size). There are different ways to apply this method:
    - **Linear probing**: initially checks the position corresponding to the dispersion function of the given key, if not try the next position, so on and so forth.

0	0		0	0	occupied		0	0	occupied
1			1	13	occupied		1	13	occupied
2			2		free		2	25	occupied
3			3		free		3		free
4	4		4	4	occupied		4	4	occupied
5			5	17	occupied		5	17	occupied
6	6	→	6	6	occupied	→	6	6	occupied
7			7	19	occupied		7	19	occupied
8			8		free		8	30	occupied
9			9		free		9		free
10	10		10	10	occupied		10	10	occupied
11			11	23	occupied		11	23	occupied
12	12		12	12	occupied		12	12	occupied
$+\{0,4,6,10,12\}$			$+\{13,17,19,23\}$				$+\{25,30\}$		

If the **load factor** is too large → throw an exception or resize (usually doubling the size) the scatter table and do a **rehash**, which consists of copying the contents of the initial table but readapting the keys to the new positions, it is a costly operation.

Binary Search Tree (BST): is a binary tree that has a key associated with each node and that satisfies:

- Nodes greater than the root go to the right subtree.
- Nodes smaller than the root go to the left subtree.



The basic operations are:

Search

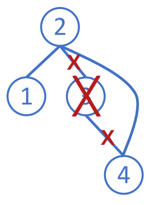
- If $k = KEY(x)$ → The search is successful.
- If $k < KEY(x)$ → It makes a recursive call on the left subtree.
- If $k > KEY(x)$ → It makes a recursive call on the right subtree.

Insertion

- If $k = KEY(x)$ → Overwrite value.
- If $k < KEY(x)$ → It makes a recursive call on the left subtree to insert it there.
- If $k > KEY(x)$ → It makes a recursive call on the right subtree to insert it there.

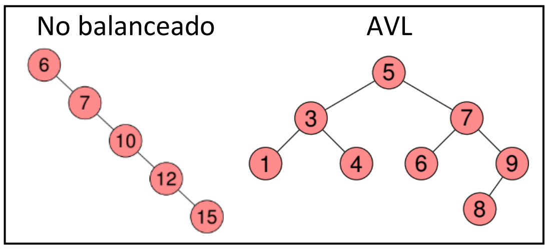
Deletions

- If the node that we want to delete is a leaf (both subtrees are empty) → Remove the node.
- If the node has only one non-empty subtree → Replace the non-empty subtree by father.
- If the node has both subtrees →
    1. Return pointer of the largest key in left subtree (inside this search in both subtrees).
    2. Replace this pointer by father, and join the left and right subtrees of the father.



The expected height of a tree with n nodes is $\Theta(\log n)$, so the cost of the above operations is on average $\Theta(\log n)$, and in the worst case $\Theta(n)$, if the height matches with the number of nodes.

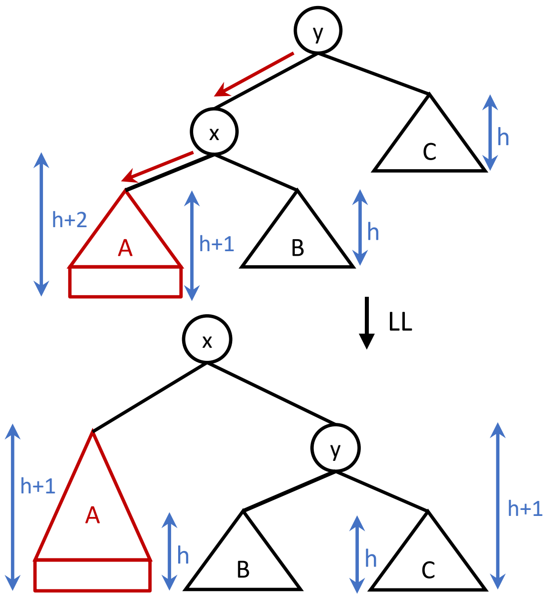
AVLs is a BST balanced: the maximum difference that can exist between the left $\boldsymbol{L}$ and the right $\boldsymbol{R}$ subtrees is 1: $|\boldsymbol{height}(\boldsymbol{L}) - \boldsymbol{height}(\boldsymbol{R})| \leq \mathbf{1}.$



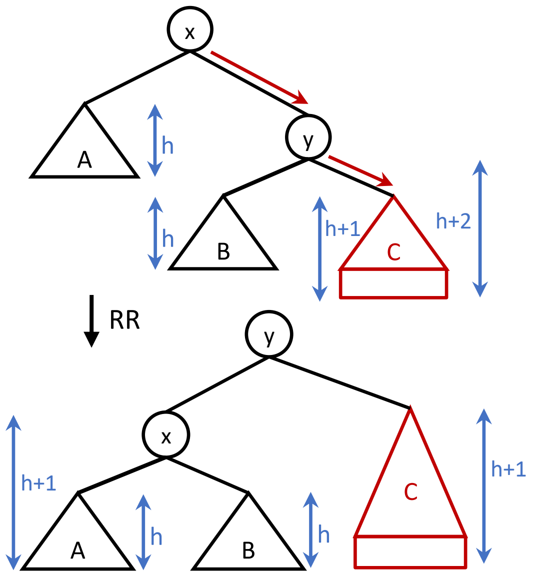
The **search** works in the same way as an BST not balanced, but the cost in the worst-case cost is $\Theta(\log n)$, because the height of an AVL of size n $\Theta(\log n)$.

Update → The insertion and deletions operations act as their counterparts for an unbalanced BSTs, but after, we check if the tree is balanced (respect the rule of height of AVLs). To re-establish the balance invariant in a node where it didn't hold, we will use **rotations**, which have cost $\Theta(1)$, thus a worst-case update has cost $\Theta(\log n)$.

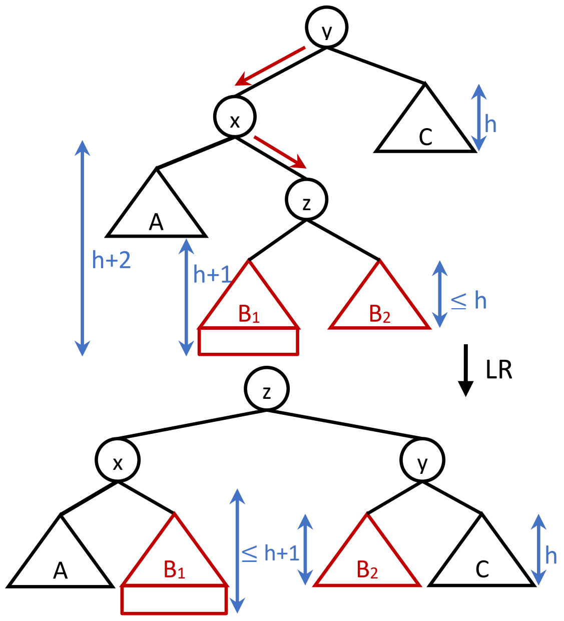
LL (left-left) simple rotation



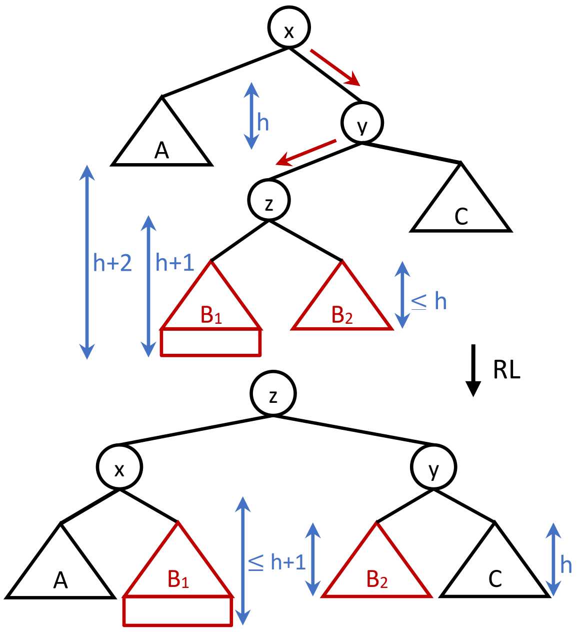
RR (right-right) simple rotation



LR (left-right) double rotation



RL (right-left) double rotation



Priority queues

Stores a collection of elements, each one has a priority associated with it, and we order by this value. Priority queues support the insertions of new elements and the query and removal of an element of minimum (or maximum) priority.

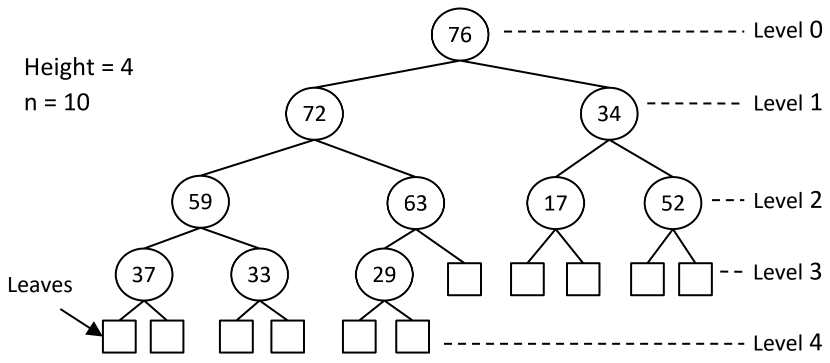
Dictionaries techniques (not hash tables) can be also used for priority queues, AVLs can be used to implement a PQ with cost $\Theta(\log n)$.

Heap → Is a binary tree that:

- All empty subtrees are located in the last two levels of the tree.
- If a node has an empty left subtree then its right subtree is also empty.

There are two types:

- Max-heaps → The priority of an element is larger or equal than that of its descendants.
- Min-heaps → The priority of an element is smaller or equal than that of its descendants.



A heap of size n has height $h = \log(n+1)$

Process of **deletion** $\Theta(\log n)$:

1. Replace the root or element to be deleted by the last element.
2. Delete the last element from the Heap.
3. Compare the left and right siblings to see which is **lower (max-heap) or grater (min-heap)** and **swap** nodes until the heap property it is fulfilled.

```
                                  10              4               5
                                 /  \            /  \            /  \
Example: delete the root (10)    5    3   →     5    3   →      4    3
                                /\              /               /
                               2  4            2               2
```

Process of **insertion** $\Theta(\log n)$**:**

1. Insert the **new node** as last leaf **from left to right**.
2. Compare **new node** value with its **parent node**.
3. If **new node value is greater (max-heap) or lower (min-heap)** than its parent, then **swap** both of them. Repeat step 2 and step 3 until the priority is fulfilled or reaches the root.

```
                                10            10             10             15
                               /  \          /  \           /  \           /  \
Example: Insert the node 15    5    3   →    5    3   →     5    15   →    5    10
                              /\            /\   /         /\   /         /\   /
                             2  4          2  4 15        2  4 3         2  4 3
```

Heapsort

Sorts an array of n elements building a heap with the n elements and extracting them, one by one, from the heap. The cost is $\Theta(n \log n)$.

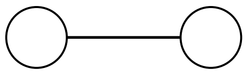
Graphs

Basic Graph Theory

A **graph** (undirected graph) is a pair $G = \langle V, E\rangle$ where V is a finite set of vertices (nodes) and E is a set of edges; each edge $e \in E$ is an unordered pair (u, v) with $u \neq v$ and $u, v \in V$.

The number of edges $|E| = m;\ \ 0 \leq m \leq \frac{n(n-1)}{2}$

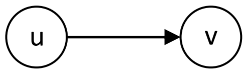
$\sum_{v \in V} in - \deg(v) = \sum_{v \in V} out - \deg(v) = |E|$



A **digraph** (directed graph) is a graph but the edges have a direction associated with them.

The number of arcs m: $\ 0 \leq m \leq n(n-1)$

$\sum_{v \in V} \deg(v) = 2 \cdot |E|$



For an arc $e = (u, v)$, vertex u is called the **source** and a vertex v the **target**. We say that v is a **successor** of u; conversely, u is a **predecessor** of v. For an edge $e = \{u, v\}$, the vertices are called its **extremes** and we say u and v are **adjacent**. We also say that the edge e is **incident** to u and v.

Degree: number of edges incident to a vertex u (number of vertices v adjacent to u)→ deg (v).

- **In-degree:** number of successors of the vertex u → in-deg(u) .
- **Out-degree**: number of predecessors of the vertex u → out-deg(u).

A graph is:

- **Dense** if the number of edges is close to the maximal number of edges, $m = \Theta(n^2)$. Ex: complete graphs.
- **Sparse** otherwise. Ex: cyclic graphs and d-regular graphs.
- A **path** is a sequence of edges connected by a vertex (except the first and the last one).
- A **simple path** is a path that no vertex appears more than once in a sequence (except the first and the last one).
- A **cycle** is a simple path that the first and the last node are the same.
- **Acyclic** if does not have any cycle.
- **Hamiltonian** if contains at least a simple path that visits all vertices of the graph.
- **Eulerian** if only a path contains all edges/arcs of the graph.
- **Connected** if and only if there exists a path in G form u to v for all pair of vertices $u, v \in V$.

Diameter of a graph: Is the maximal distance between a pair of vertices. To compute the diameter, we need to perform a BFS starting from each vertex v in G. $\ \Theta(n \cdot m)$

Center of a graph: is the vertex such that the maximal distance to any other vertex is minimal .

Is **subgraph** of G if the graph is the same but without some edges or vertex.

- **Subgraph induced**: if is a subgraph with the same edges/arcs than G.
- **Spanning subgraph**: if is a subgraph with the same vertices than G.

A **connected component** C of a graph G is a maximal connected induced subgraph of G. By maximal we mean that adding any vertex v to $V(C)$ the resulting induced subgraph is not connected.

Trees

- A **(free) tree** is a connected acyclic graph. Then $|E| = |V| - 1$.
  - A **spanning tree** is a spanning subgraph and a tree.
- A **forest** is an acyclic graph. Then $|E| = |V| - c$ (c = connected components).

- A **spanning forest** is an acyclic spanning subgraph consisting of a set of trees, each a spanning tree for the corresponding connected components of G.

Often, we will see (di)graphs in which each edge/arc bears a **label**, these are numeric (integers, rational, real numbers). When a graph is labelled then we called **weighted graphs**, and the label of the edge is called its weight.

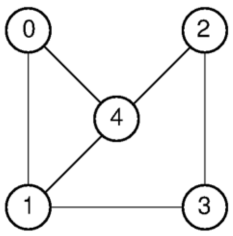
Implementation of Graphs

Adjacency matrices: are too costly in space; if $|V(G)| = n \rightarrow$ it needs space $\Theta(n^2)$ to represent the graph. Are fine to represent dense graphs or when we need to answer very efficiently whether an edge $(u, v) \in E$ or not.

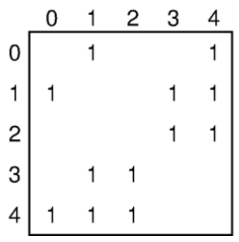
- Entry $A[i][j]$ is a Boolean indicating whether (i, j) is an edge/arc or not.
- For a weighted (di)graph $A[i][j]$ stores the label assigned to the edge/arc (i, j).

Adjacency lists: They require space $\Theta(n + m)$ where $n = |V|$ and $m = |E|$, in general rule use this implementation.

- We use an array or vector T such that for a vertex u, $T[u]$ points to a list of edges incident to u or a list of the vertices $v \in V$ which are adjacent to u.
- For digraphs, we will have the lists of successors of a vertex u, for all vertices u, and, possibly, the list of predecessors of a vertex u, for all vertices u.

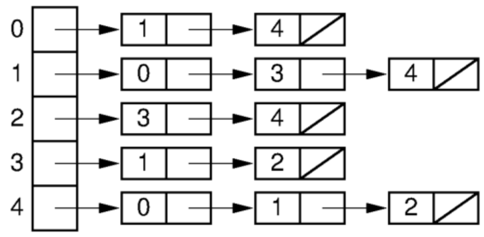


Graph



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

Adjacency matrix



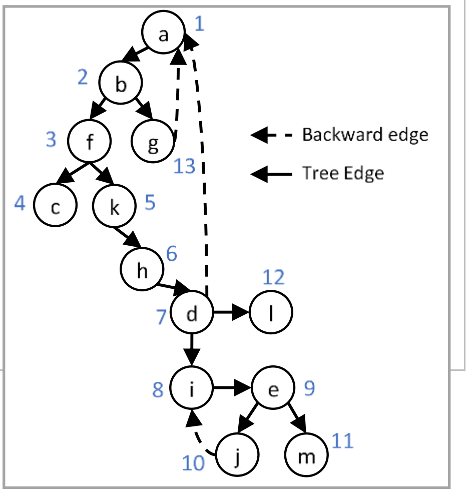
Adjacency list

DFS (Depth-First Search)

We visit a vertex v and from there we traverse recursively, each non-visited vertex w which is adjacent/a successor of v. A vertex remains **open** until the recursive traversal of all its adjacent/successors has been finished, after that the vertex gets **closed**. $\Theta(n + m)$

```
 void DFS_REC(const Graph& graph, Visited& vis, int u) {
   vis[u] = true;
   for (int v : graph[u])
         if (not vis[v])
            DFS_REC(graph, vis, v);
}

void DFS(const Graph& graph) {
   Visited vis(graph.size(), false);
   for (int u = 0; u < graph.size(); u++)
      if (not vis[u])
         DFS_REC(graph, vis, u);
}
```



Applications of DFS:

- Detecting cycles or find Connected Components (CCs)
- Decide if a graph is bipartite.
- Decide if a graph is N-colorable.

DAG (Directed Acyclic Graph): where in a large complex software system, each node is a subsystem and each arc (A, B) indicates that subsystem A uses subsystem B.

A **Topological sort** of a DAG G is a sequence of all its vertices such that for any vertex w, if v precedes w in the sequence then $(w, v) \notin E$ (no vertex is visited until all its predecessors have been visited).

```
// The graph is an adjacency list
vector<int> topological_sort(const Graph& graph) {
   // Array of predecesors
   vector<int> pred(graph.size(), 0);

   // For all adjacent nodes of u to v add a predecesor to v
   for (int u = 0; u < graph.size(); u++)
      for (int v : graph[u])
         pred[v]++;

   /** Priority queue because we need it to be in lexicographical order **/
   priority_queue<int, vector<int>, greater<int>> q;

   stack<int> s;
   for (int u = 0; u < graph.size(); u++) // Add all nodes without predecesors
      if (pred[u] == 0)
         s.push(u);

   vector<int> nodes;
   while (not s.empty()) {
      int u = s.top();
      s.pop();
      nodes.push_back(u);     // Adds the node to list

   for (int v : graph[u])     // For all Adjacent nodes
      if (--pred[v] == 0)     // If num of predecesors not visited = 0
         s.push(v);           // Adds to stack
   }
   return nodes;
}
```

BFS (Breadth-First Search)

Given a vertex s we visit all vertices in the connected component of s in increasing distance from s. When a vertex is visited, all its adjacent non-visited vertices are put into a queue of vertices yet to be visited. $\Theta(|V| + |E|)$.

```
// Finds the distance between a given source and destination
int BFS(const graph& G, int source, int destination) {
   // Vector that helps us to know the visited vertices
   vector<int> distance(G.size(), -1); // -1 indicates that hasn't been visited
   queue<int> q;
   q.push(source);         // Initialize the queue
   distance[source] = 0;   // Set the source as visited

   while (not q.empty()) {
      int aux = q.front();
      q.pop();

      // The node destination has been found!
      if (aux == destination) return distance[aux];
```

```
    for (int s : adjacent(aux)) {
      // Check if has been visited
      if (distance[s] == -1) {
        // Mark as visited and set the distance from source
        distance[s] == distance[aux] + 1;
        q.push(s);
      }
    }
  }
  return -1;   // Cannot go to destination from source
}
```

Applications of BFS:

- Shortest path between nodes
- The diameter of a given graph.
- The center of a given graph

It Works with graphs and unweighted digraphs.

It is used as model for the following algorithms:

Dijkstra’s Algorithm

Find all the shortest paths from a given vertex to all other vertices in the digraph. The worst-case cost is $\Theta\big((m+n)\log n\big)$.

```
int dijkstra_algorithm(const graph& G, int source, int destination) {
  vector<bool> dist(G.size(), false);    // Sets all nodes as not visited
  vector<int> dist(G.size(), INT_MAX);   // Sets distance as ∞ for all nodes
  dist[source] = 0; // Sets the distance from the origin node to 0

  // Queue that stores the elements increasingly by distance <dist, Node>
  priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>> q;
  q.push({0, source}); // Adds the source node to the queue

  while (not q.empty()) {
    pair<int, int> p = q.top();
    q.pop();

    int u = p.second;
    if (not vis[u]) {
      vis[u] = true;

      // For all adjacent nodes
      for (int s : adjacent(aux)) {
        // If the path is weightless
        if (dist[s] > dist[u] + weight(u_to_v)) {
          dist[s] = dist[u] + weight(u_to_v);
          q.push({dist[s], s});
        }

        /*** If not only exists a unique path ***/
        else if (dist[v] == dist[u] + weight(u_to_v))
          // do something
      }
    }
  }
  return dist[destination]; // Returns the distance of shortest path
}
```

Minimum Spanning Trees: Prim’s Algorithm

Obtain a spanning subgraph with the minimum weight of edges that contains all the vertices of the graph with no-cycles. The cost if we use a priority queue giving a started bound is . $\Theta(m \log n)$.

Procedure: start with a vertex and look for the weightless edge, once you are on the second edge, we look in all visited edges for the lighter edge without a visited vertex. Do the same until all vertices of the original graph are in our subgraph.

```cpp
// pair<weight, node>
typedef pair<int, int> P;

int mst(const vector<vector<P>>& G) {
   vector<bool> vis(G.size(), false);
   // queue that order the inputs by wieght (first element weightless)
   priority_queue<P, vector<P>, greater<P>> q;

   // Push to queue all adjacent vertices of first node
   for (auto p : G[0]) q.push(p);
   vis[0] = true;  // Sets first node as visited

   int sum = 0;    // Sum of weights
   int count = 1;  // Count to visit all nodes
   while (count < G.size()) {
      P p = q.top();
      q.pop();

      int w = p.first;  // Get weight
      int v = p.second; // Get node
      if (not vis[v]) {
         vis[v] = true;
         for (auto p : G[v]) q.push(p);
         sum += w;
         count++;
      }
   }
   return sum;
}
```

Exhaustive Search and Generation

The brute-force algorithms (or exhaustive search) try all the possible combinations.

Backtracking

That type of algorithms build partial solutions and the main advantage is that if the partial solution is no longer a valid solution it does not continue with the partial solution.

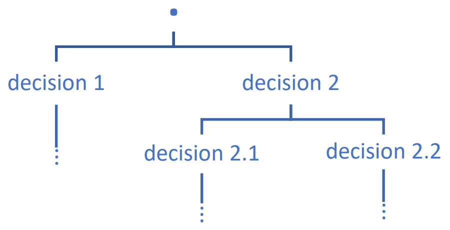
1- Explore the partial solution doing an additional one, moving one step closer to final solution, if the new partial solution does not violate any constraint, we continue with that solution.
2- If the partial solution not satisfy the constrains, then we undo the last decision made.

Branch & Bound

It is an improved version of the Backtracking variant, mainly used to solve optimization problems. This variant is usually interpreted as a tree of solutions where each branch leads to a possible solution after the current one and its main difference to the previous variant is that it detects when a branch is no longer optimal to "prune" that branch and not continue wasting resources with that solution.

This version is more expensive in memory than Backtracking.

```
procedure BRANCH-AND-BOUND(z)
   Alive: a priority queue of nodes
   y := ROOT_NODE(z)
   Alive.INSERT(y, ESTIMATED-COST(y))
   Initialize best_solution and best_cost, e.g., best_cost := +1
   while ¬Alive:IS_EMPTY() do
      y := Alive.MIN_ELEM(); Alive.REMOVE_MIN()
      for y’ ∈ SUCCESSORS(y, z) do
         if IS_SOLUTION(y’, z) then
            if COST(y’) < best_cost then
               best_cost := COST(y’)
               best_solution := y’
               Purge nodes with larger priority = cost
            end if
         else
            feasible := IS_FEASIBLE(y’, z)^
               ESTIMATED-COST(y’) < best_cost
            if feasible then
               Alive.INSERT(y’, ESTIMATED-COST(y’))
            end if
         end if
      end for
   end while
end procedure
```



Notions of Intractability

Complexity Theory aims at finding the complexity of computational problems and classify them according to their complexity. The **analysis of algorithms** studies the amount of resources needed by an algorithm to solve a problem, instead the complexity theory consider the possible algorithms to solve a problem. Classifications of problems:

- A **decisional problem** is a computational problem in which for every possible input x, one must determine if a certain property is satisfied or not (the output is "yes/no", "true/false", "0/1").
- The solution of a decisional problems often can be used for an "efficient" solution to the **non-decisional problem**.

A decisional problem $\boldsymbol{P}$ **is decidable in time** $\boldsymbol{t}$, because it depends if the problem can be solved.

- **Polynomial time**: a problem belongs to P if is decidable in time n^k by some k
- **Exponential time**: a problem is EXP if it is decidable in time 2^{n^k} by some k.

Examples:

CONNECTIVITY ∈ P
REACHABILITY ∈ P
PRIMALITY ∈ P
SHORTEST-PATH ∈ P
2-COLORABILITY ∈ P
3-COLORABILITY ∈ EXP; also ∈ P? (not known to be P)
LONGEST-PATH ∈ EXP; also ∈ P? (not known to be P)
TSP ∈ EXP; also ∈ P? (not known to be P)
GENERALIZED-CHESS ∈ EXP
GENERALIZED-CHECKERS ∈ EXP
GENERALIZED-GO ∈ EXP

The algorithms seen so far are **deterministic algorithms**: the computation path from the input to the output is unique .

The **non-deterministic algorithms** have many distinct computational paths, forming a tree. These start the algorithm as deterministic algorithm until the first function CHOOSE(y) which returns a value between 0 and y, each value corresponding to one possible solution.

P are "easy" because it's easy to do an algorithm in polynomial time and it's easy to solve an example of it.

NP are "hard" problems that are easy to verify if somebody give us an example, but it is more complicated to find an easy algorithm.

NP-hard problems are at least as hard as any other problem in NP. To reduce a problem, we need a problem A that is more difficult than a problem B, if we solve A easily, we can solve B as well.

NP-complete is the most difficult problem in NP, to prove it is NP-complete we look if it is in both NP and NP-hard. Are a set of NP problems Y that can be reduced by others NP problems X in polynomial time.

A computational problem that for now don't know the answer is if $P = NP$ or $NP \neq P$.

Diagram if $P \neq NP$:

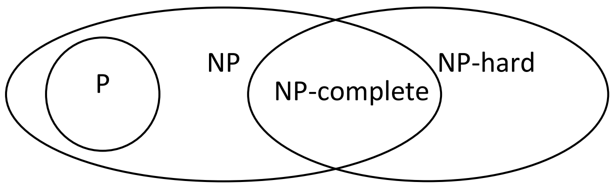
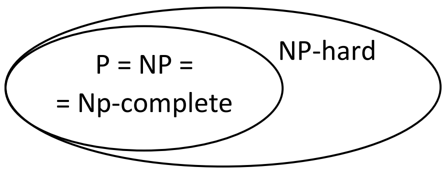


Diagram if $P = NP$:


P = NP =
= Np-complete
NP-hard