Algorithm and data structures

Roberto Antoniello

August 1, 2023

In this file I will resume all the concepts I liked while studying the course of Algorithm and data structures.

1 Binary search

This algorithm can be used only if you have a sorted array. Here how it works: BinarySearch take a sorted array as input and return an index as output. So it returns the index of the found element or -1 if not found.

When it starts execution, the algorithm saves three variables **sx**, **dx** and **m**. The **m** variable is the index in the middle of the array, **sx** and **dx** are the first and the last index. It asks if the element is less or more than the element in **m** position.

Basically, if the element is x the question is: $\mathbf{x} < \mathbf{A[m]}$ or $\mathbf{x} > \mathbf{A[m]}$? We are reducing the search space by 2 every time because if it's less, our **dx** becomes **m**, otherwise our **sx** becomes **m+1**.

At the first iteration the search space is n elements, at the second it is $\frac{n}{2}$, at the third one it is $\frac{n}{2^2}$ and so on.
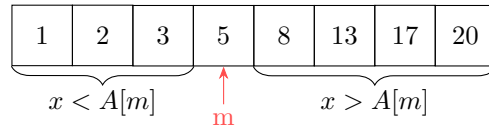
At the i° iteration it will be $\frac{n}{2^i}$.

During the last iteration the size of our array A is 1. So:

$\frac{n}{2^i} = 1 \Rightarrow n = 2^i \Rightarrow i = \log_2 n$

We have just said the amount of steps are $\log_2 n \Rightarrow O(\log n)$

Here below there's an array as example with the initial value of sx,m and dx.

1	2	3	5	8	13	17	20

sx   m   dx

1	2	3	5	8	13	17	20

$$\underbrace{\qquad\qquad}_{x < A[m]} \qquad \underset{m}{\uparrow} \qquad \underbrace{\qquad\qquad}_{x > A[m]}$$

I let you read the pseudocode here below.

Listing 1:*Iterative version of the binary search algorithm.*

```
Algorithm BinarySearch(Array A[0,..,n-1]) --> index
   sx <-- 0
   dx <-- n
   index <-- -1
   while sx < dx do
      m <-- (sx+dx) / 2
      if x < A[m] then
         dx <-- m
      else if x = A[m] then
         index <-- m
      else
         sx <-- m+1
```

2 Selection Sort

So we know that to do a biinary search, we need a sorted array.
There are many solutions to sort an array with more or less complexity in time and space.
The first algorithm we're gonna see is the selection sort. This sorting algorithm is really simple, essentially it slides the array and when it find the minimum value it put it at the beginning in the right place. After this step we consider that element sorted and so on until we have the entire array sorted.
Let's see here below the pseudocode and then a quick and easy complexity analysis.

Listing 2:*Selection sort algorithm.*

```
Algorithm SelectionSort(Array A[0,..,n-1])
   for k <-- 1 to n-2 do
      //finding the index m of minimum value between k and n-1
      m <-- k
      for j <-- k+1 to n-1 do
         if A[j] < A[m] then
            m <-- j
      swap A[m] with A[k]
```

The first for cycle ends at **n-2** because the last element is already sorted if we remember how it works the algorithm.
The second for cycle finds the minimum value starting from m position.
So, how many comparisons have I to do?
If we look at our array A, we can consider that at the k iteration we have **k** sorted elements on the left and **n-k-1** to sort on the right.

1	2	3	20	8	18	6	12

$\underbrace{\qquad\qquad}_{k\ sorted}$ $\uparrow$ $\underbrace{\qquad\qquad}_{n-k-1\ to\ sort}$
k
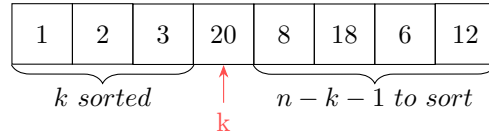
Basically we are doing this sum:

$$\sum_{k=0}^{n-2}(n-k-1)$$

This is equal to sum every $n \in \mathcal{N}$ from 0 to n-2, so:

$$\sum_{k=0}^{n-2}(n-k-1) = \sum_{i=1}^{n+1} i = \frac{n \cdot (n-1)}{2}$$

This is the total of comparisons I have to do in this algorithm. The complexity is thus $\Theta(n^2)$.
We can say also that this amount of comparisons is always done because we do not set any condition to do these comparisons.

3 Bubble sort

The second sorting algorithm we're gonna see is Bubble sort.
This works by sliding the array and compare the current element with the previous one. The bigger element is moved in the next position. This make the bigger element go forward until there's another bigger than it. It's called bubble because these big values go forward like the bubbles in the water.
Let's do a simulation here below.

5	7	6	1	13	2	12

let's do the first iteration.

5	6	1	7	2	12	13

We compared 5 with 7 $\Rightarrow$ already sorted.
Then 7 with 6 $\Rightarrow$ swap.

3

Then 7 with 1 $\Rightarrow$ swap.
Then 7 with 13 $\Rightarrow$ already sorted.
Then 13 with 2 $\Rightarrow$ swap.
Then 13 with 12 $\Rightarrow$ swap.
In the next iteration I'll slide again until the entire array is sorted.

Now we consider the pseudocode.

Listing 3:*Bubble sort algorithm.*

```
Algorithm BubbleSort(Array A[0,..,n-1])
  i <-- 1
  do
    swapped <-- false
    for j <--1 to n-i do
      if A[j] < A[j-1] then
        swap A[j] with A[j-1]
        swapped <-- true
    i <-- i + 1
  while swapped and i < n
```

The for cycle ends at **n-i** because the last **i** elements are already sorted as we discussed how it work bubble sort. The external do while cycle ends when we did an iteration without any swap of elements or we reached the **n-1** iteration. The **i < n** condition saves us the last iteration without any swap sliding the already sorted array.

Let's analyse the comparisons. Inside the main for cycle we do **n-i** comparisons where **i** starts at **1** to **n-1**. We're doing this sum:

$$\sum_{i=1}^{n-1}(n-i) = \sum_{k=1}^{n-1} k = \frac{n \cdot (n+1)}{2} = \Theta(n^2)$$

Again, the total amount of comparisons is in the order of n^2, but there's a difference between selection sort and bubble sort. That's because here we have this amount of comparisons only in the worst case(reverse sorted array), but in the best case(array already sorted) we can easily see that the amount of comparisons is in the order of $\Theta(n)$ because we just slide one time the array and then the algorithm ends doing **n-1** comparisons.

4   Merge sort

Let's consider the idea of having two already sorted arrays.
How can we build an array that contains all the elements sorted?
The first option we can think is to do a bubble sort and surely we can do it in n^2 steps.
Anyway there's a clever way to do this and it's called **merge**.

5	6	10	12		2	9	15	20

2	5	6	9	10	12	15	20

4.1 merge

The merge procedure works as this:

at every step I compare the first two numbers and the mininum go in the final array. When one of the sorted array ends, I append the remaining numbers of the other array without compare anything.

How much it costs? Well, every compare will put an element in the right place, so in the worst case I compare all the elements and the remaining is only one element in the other array. So the merge procedure has complexity in the order of $O(n)$ because I do **n-1** comparisons in the worst case.

Here below I drop the pseudocode of a simple version of the merge procedure, which it isn't perfectly optimized in memory.

Listing 4:*Merge algorithm.*

```
Algorithm Merge(Array B[0,..,lb-1], Array C[0,..,lc-1]) --> Array
  Be X[o,..,lb+lc-1] an array
  i1,i2,k <-- 0
  while i1 < lb and i2 < lc do
    if B[i1] <= C[i2] then
       X[k] <-- B[i1]
       i1 <-- i1+1
    else
       X[k] <-- C[i2]
       i2 <-- i2+1
    k <-- k +1
  //The remaining elements are in B
  if i1 < lb then
    for j <- i1 to lb-1 do
       X[k] <-- B[j]
       k <-- k+1
  else
    //The remaining elements are in C
    for j <--i2 to lc-1 do
       X[k] <-- C[j]
       k <-- k+1
  return x
```

Let's remember two corollaries we will use later:

$$\sum_{i=0}^{k-1} 2^i = 2^k - 1$$

If you convert in binary, you can see that all the power of two are 1 with all zeros behind. If you sum those all you obtain 2^k.
Now the second corollary:
$\forall n \in \mathcal{N} \; \exists$ a power of two $N \; : \; n \leq N \leq 2n$

4.2   Using merge inside Merge sort

Let's define Merge sort recursively:
If $n \leq 1$ then A is already sorted, I do nothing.
Otherwise:

1. Divide A by two equal parts.

2. Sort them separately.

3. Merge them in a unique array(merge algorithm)

I'll show here below the pseudocode of the Merge sort, which use our merge algorithm written before. So this algorithm suffers too of the missing optimization in memory, but it will be good enough to do our complexity analysis.

Listing 5:*Merge algorithm.*

```
Algorithm Mergesort(Array A[0,..,n-1)
  If n > 1 then
    m <-- n / 2
    B <-- A[o,..,m-1]
    C <-- A[m,..,n-1]
    Mergesort(B)
    Mergesort(C)
  A <-- merge(B,C)
```

4.3 Amount of comparisons

The amount of comparisons made are:

1. Comparisons of Mergesort(B).

2. Comparisons of Mergesort(C).

3. Comparisons of merge(B,C).

The equation resulting from this observation is here below and we're going to solve it.

$$\begin{cases} 0 & if\ n \leq 1 \\ C\left(\left\lfloor \dfrac{n}{2} \right\rfloor\right) + C\left(\left\lceil \dfrac{n}{2} \right\rceil\right) + C_{merge}(n) & otherwise \end{cases}$$

Let's consider our **n** is even. The equation becomes:

$$\begin{cases} 0 & if\ n \leq 1 \\ 2C\left(\dfrac{n}{2}\right) + C_{merge}(n) & otherwise \end{cases}$$

By solving this equation we're going to demonstrate the amount of comparisons Mergesort do until we have a sorted array in the end. We'll solve by substitution.

$$C(n) = 2C\left(\frac{n}{2}\right) + n - 1 =$$

$$2\left[2C\left(\frac{n}{2^2}\right) + \frac{n}{2} - 1\right] + n - 1 =$$

$$2^2 \cdot C\left(\frac{n}{2^2}\right) + n - 2 + n - 1 =$$

$$2^2 \cdot \left[2C\left(\frac{n}{2^3}\right) + \frac{n}{2^2} - 1\right] + n - 2n - 1 =$$

$$2^3 C\left(\frac{n}{2^3}\right) + n - 2^3 + n - 2^1 + n - 2^0 =$$

$$2^3 C\left(\frac{n}{2^3}\right) + 3n - \sum_{i=0}^{3-1} 2^i =$$

$$2^k C\left(\frac{n}{2^k}\right) + kn - \sum_{i=0}^{k-1} 2^i =$$

We know that $\sum_{i=0}^{k-1} 2^i = 2^k - 1$

Now let's manipulate to the base case:

$\dfrac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2(n)$ Let's substitute now:

$C(n) = n \cdot C(1) + n \cdot \log_2 n - n + 1$

We know that $C(1) = 0$, so the amount of comparisons is in the order of

$$\Theta(n \cdot \log(n))$$

What about the recursion stack of memory?

If we still consider n as even, then the equation is:

$$H(n) = 1 + H\left(\frac{n}{2}\right)$$

If we do similar manipulation as made a moment ago we can reach this result:

$$H(n) = \Theta(\log(n))$$

5   Quick sort

The recursive definition of Quick sort is pretty similar to the Merge sort one. This is because both algorithm are using the Divide et impera technique.

The difference between these two sortin algorithm is quite simple. Inside Merge-sort the complex part was the final merge, instead here we're going to see that the complex part will be the first one of dividing the problem in more problem of minor size.

| 44 | 55 | 12 | 42 | 94 | 9 | 18 | 64 |

We choose a pivot element, 42 for example. if $n \leq pivot$ then those elements go on the left of pivot, otherwise they go on the right.

9	12	18	42	44	55	64	94

Pivot

How to do this? We use an algorithm we will call partition. We define two index **sx** and **dx**. sx starts at the beginning of array while dx starts from the end.

1. dx starts sliding until it finds an element $\leq$ of pivot.

2. sx then starts sliding until it finfd an element $>$ of pivot.

3. Then swap sx with dx.

4. If sx $<$ dx, then swap dx with the pivot.

Let's give a look to the pseudocode of partition here below.

Listing 6:*Partition algorithm.*

```
Algorithm Partition(Array A[0,..,n-1],index i,index f) --> index
  pivot <-- A[i]
  sx <-- i
  dx <-- f
  while sx < dx do
    do
        dx <-- dx-1
    while A[dx] > pivot
    do
        sx <-- sx+1
    while sx < dx and A[sx] <= pivot
    if sx < dx then
        swap A[sx] with A[dx]
  swap A[i] with A[dx]
  return dx
```

The amount of comparisons is in the order of $O(n)$.
Now let's see the pseudocode of Quicksort that use partition.

Listing 7:*Quicksort algorithm.*

```
Algorithm Quicksort(Array A[0,..,n-1])
  Quicksort(A,0,n)
```

Listing 8:*Quicksort procedure.*

```
Procedure Quicksort(Array A[0,..,n-1],index i,index f)
```
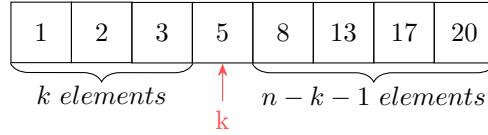
9

```
if f - i > 1 then
   m <-- partition(A,i,f)
   Quicksort(A,i,m)
   Quicksort(A,m+1,f)
```

We have now to calculate the amount of comparisons we do in this algorithm. To do this let's consider this aspect:

$k\ elements$ k $n - k - 1\ elements$

Basically, we have $C(k)$ on the left and $C(n - k - 1)$ on the right.
We know also that $C_{part}(n) = n$
So the total amount of comparisons is made of $n + C(k) + C(n - k - 1)$

5.1 Worst case

It's easily to see that the worst case happen when $k = 0$ or $k = n - 1$.

$$\begin{cases} 0 & if\ n \leq 1 \\ n + max\,(C_w(k) + C_w(n - k - 1) : k = 0, .., n - 1) & otherwise \end{cases}$$

$$C_w(n) = n + C_w(n - 1) = n + n - 1 + C_w(n - 2) = \\ n + n - 1 + n - 2 + C_w(n - 3) = \ldots =$$

$$n + n - 1 + n - 2 + \ldots + 2 + C(1) = \sum_{i=1}^{n} i = \frac{n \cdot (n + 1)}{2} \Rightarrow \Theta(n^2)$$

Well well well, we have built this complex algorithm but in the worst case we just found that the amount of comparisons is in the order of n^2 such as a Selection sort or a Bubble sort.
There's something to say about this anyway, because this case is very rare and you have to find it manually to make it happens for real.

5.2 Best case

The best case happens when the partition is balanced. We just obtain similar result as we did for Mergesort.

$$\begin{cases} 0 & if\ n \leq 1 \\ n + min\,(C_w(k) + C_w(n - k - 1) : k = 0, .., n - 1) & otherwise \end{cases}$$

$$C_b(n) \simeq n + 2C_b\left(\frac{n}{2}\right) \Rightarrow C_b(n) \simeq n \cdot \log_2(n)$$

5.3 Average case

The average case is a little bit more difficult to analyze, but don't worry we'll make it too.

$$\begin{cases} 0 & if \ n \leq 1 \\ \dfrac{\sum_{k=0}^{n-1} C_n + C(k) + C(n-k-1)}{n} & otherwise \end{cases}$$

Let's manipulate this bad fraction.

$$C(n) = \frac{1}{n}\sum_{k=0}^{n-1} n + \frac{1}{n}\sum_{k=0}^{n-1} C(k) + \frac{1}{n}\sum_{k=0}^{n-1} C(n-k-1)$$

The first summation is equal to n.
The second and the third are symmetrical because the second sum from 0 to n-1 and the third do the reverse thing. So:

$$C(n) = n + \frac{2}{n}\sum_{i=0}^{n-1} C(i) = n + \frac{2}{n}\sum_{i=2}^{n-1} C(i)$$

$C(0) = C(1) = 0 \Rightarrow$ we can start from $i = 2$ Let's update the equation after these steps.

$$\begin{cases} 0 & if \ n \leq 1 \\ n + \dfrac{2}{n}\sum_{i=2}^{n-1} C(i) & otherwise \end{cases}$$

Now we'll demonstrate by induction that $C(n) \leq 2n \cdot ln(n) \ by \ n \geq 1$
Base step: $n = 1 \Rightarrow C(1) = 0 \Rightarrow 2ln(1) = 0 \Rightarrow OK$ Induction step:

$$C(n) = n + \frac{2}{n}\sum_{i=2}^{n-1} C(i) \leq n + \frac{2}{n}\sum_{i=2}^{n-1} 2i \cdot ln(i) = n + \frac{4}{n}\sum_{i=2}^{n-1} i \cdot ln(i)$$

We'll make use of an integral to get the correct value of this summation and refactor this disequation.

$$\sum_{i=1}^{n-1} i \cdot ln(i) \leq \int_{2}^{n} x \cdot ln(x)dx$$

We can solve this integral by parts.

$$\int_2^n x \cdot ln(x)dx = \frac{x^2}{2} \cdot ln(x) - \int_2^n \frac{1}{x} \cdot \frac{x^2}{2}dx = \frac{x^2}{2} \cdot ln(x) - \frac{x^2}{4}$$

$$\Rightarrow \left[\frac{x^2}{2} \cdot ln(x) - \frac{x^2}{4}\right]_2^n = \frac{n^2}{2} \cdot ln(n) - \frac{n^2}{4} - 2ln(2) + 1$$

$2ln(2) + 1 \simeq 0$

So:

$$\sum_{i=1}^{n-1} i \cdot ln(i) \leq \frac{n^2}{2} \cdot ln(n) - \frac{n^2}{4}$$

Then we return to the original disequation and we update it:

$$n + \frac{4}{n}\sum_{i=2}^{n-1} i \cdot ln(i) \leq n + \frac{4}{n} \cdot \left(\frac{n^2}{2} \cdot ln(n) - \frac{n^2}{4}\right)$$

$= n + 2n \cdot ln(n) - n = 2n \cdot ln(n) \Rightarrow Demonstrated.$

So we have demonstrated that in average case:
$C(n) \leq 2n \cdot ln(n) \simeq 1.39n \cdot ln(n)$

Regarding the use of the stack for recursion, we can say that such as merge sort this version is less optimized about memory because we have two recursions call in queue.
In average and best case we have a complexity in space in the order of $\Theta(log(n))$ but when the array is already sorted or almost sorted the amount of stack memory is in the order of $\Theta(n)$.

6 Binary Trees

A binary tree is a data structure which in a basic form has a special node called root and two nodes below attached called left child and right child. Below every child can be other child attached and so on.
Here below there's an example of binary tree made of integer numbers.
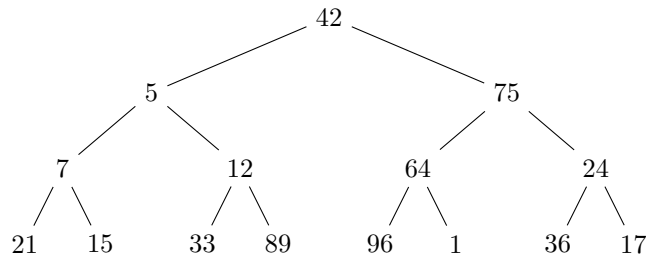
A binary tree has different features linked to the structure is made of. There are three different way to visit all nodes recursively of a binary tree by using the depth-first search.

1. **preorder**: first the root, then the left subtree and the right subtree.

2. **inorder**: first the left subtree, then the root and the right subtree.

3. **postorder**: first the left subtree, then the right subtree and the root in the end.

6.1 Binary search tree

There are several types of binary trees. A first type we'll give a look is called **binary search tree**.
This one has a little rule regarding the insertion of new nodes. For every node, the node on the left must be smaller and the node on the right must be bigger. Here below I'll show an example of a binary search tree.

```
              42
            /    \
          5        75
         / \      /  \
        7   12   64   44
```
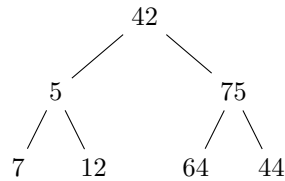
This structure gain some advantages because if I want for example find the maximum value I need to go only right until I find the value.
You can demonstrate that the amount of steps in time that **insertion**,**search** and **delete** operations do, depend only from the height of the tree.
So we have to compare the amount of nodes in a tree with the respective height. If we have to insert some nodes in a binary search tree and all of them are sorted, we can say that the height of the tree is equal to $h = n - 1$ where n is the amount of nodes and h the height.

```
          42
         /
       38
         \
         35
        /
      22
```
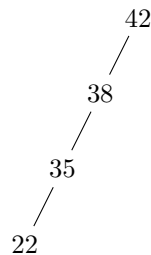
We know now that the mininum number of nodes in a tree of height h is equal to $h + 1$.
Now we demonstrate that the maximum number of nodes in a tree is equal to $2^{h+1} - 1$. We'll demonstrate by induction.

$$h = 0 \Rightarrow 2^{0+1} - 1 = 2 - 1 = 1 \Rightarrow OK$$
$$h \geq 0 \Rightarrow nodes(T) = 1 + nodes(left) + nodes(right)$$
We can consider that both subtree are just a tree of height h-1, so
by using the induction hypothesis we continue:
$$nodes(T) = 1 + 2^h - 1 + 2^h - 1 = 2^{h+1} - 1$$
So after this demonstration we can infer that:
$$h + 1 \leq n \leq 2^{h+1} - 1 \Rightarrow \log_2(n+1) \leq h \leq n - 1$$

6.2 Perfectly balanced tree

Well after what we said, we know that our focus need to be to balance the height of a tree because if a tree is well filled of nodes balanced on height, we can do operations in the order of $\Theta(\log(n))$.
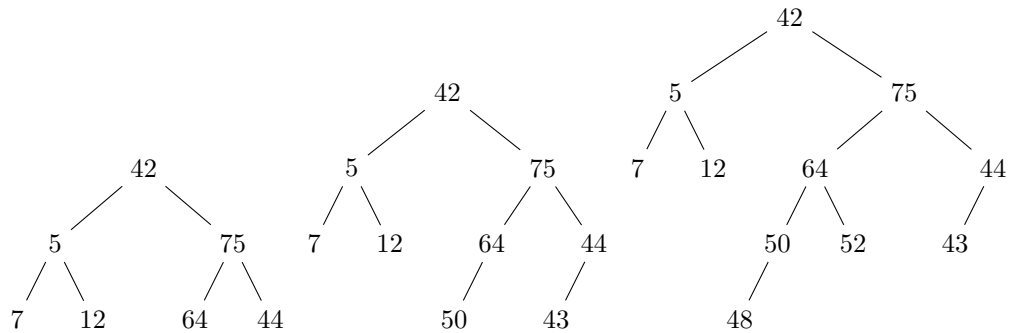This kind of tree has the following rule for insertion of new nodes:
For every node, the difference of the amount of nodes between the left subtree and the right subtree in absolute value can be at most 1. Here below a couple of example, the last one it isn't perfectly balanced.
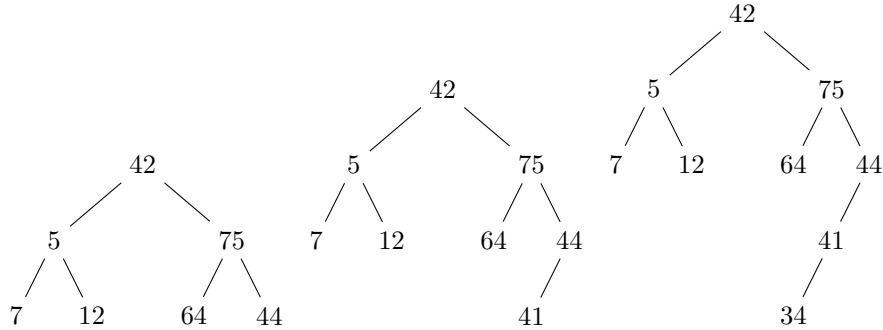
The height is in the order of $\Theta(\log(n))$ but if you consider the balance condition, you can easily see that in some case an insertion can cost the restructuring of the entire tree, which costs linear time $\Theta(n)$.

6.3 AVL tree

This kind of tree has a different balance condition, which is the following:
for every node, the difference of the height between the left subtree and the right subtree can be at most 1. Here below a couple of example, the last one it isn't avl.

```
                                                          42
                                                        /    \
                          42                          5        75
                        /    \                       / \      /   \
            42         5      75          7    12   64    44
          /    \      / \    /   \                  / \    \
        5      75    7  12  64    44              50  52   43
       / \     / \          /      \               |
      7  12   64 44        50      43             48
```

You can prove that the height of AVL tree is always in the order of $\Theta(\log(n))$.
So **insertion, search and delete** can be done in $\Theta(\log(n))$.
Whenever an insertion make the tree unbalanced, there's a method to re-balance the tree by making a rotation of the involved nodes.

6.4   B-tree(Bayer)

This kind of tree is useful to reduce the amount of access to the storage memory, which is $\simeq 100.000$ slower than RAM memory. Below the definition of structure:
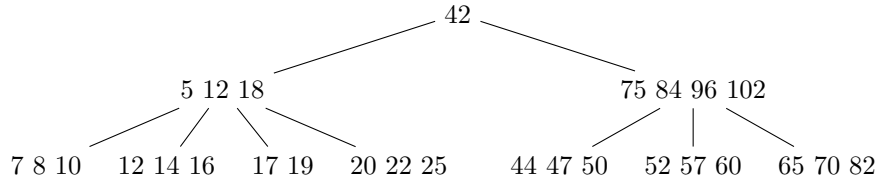A tree of t order is a B-tree T that:

- Every node has at most 2t children

- every internal node that it isn't the root has at least t children.

- The root has at least 2 children.

- All the leafs are at the same depth.

Definition of content of the tree:

- Every leaf contains k sorted keys:
  $a_1 \leq a_2 \leq ... \leq a_k$

- $t - 1 \leq k \leq 2t - 1$

- every internal node with k+1 children and subtree $T_0, .., T_k$ contains k sorted keys $a_1 \leq a_2 \leq ... \leq a_k$ that:
  $\forall \; key \; C_i \; of \; T_i = 0, .., k$ results that:
  $C_0 \leq a_1 \leq a_2 \leq \ldots \leq C_{k-1} \leq a_k \leq C_k$

An example with t=3:

```
                                    42
          5 12 18                              75 84 96 102
 7 8 10  12 14 16  17 19  20 22 25    44 47 50   52 57 60   65 70 82
```

Every node is just a sorted array containing an amount of keys $\leq 2t$. So if I would like to search for a node, I can so a binary search using $\Theta(\log(n))$ steps and then continue on the respective subtree if not found.

So the total amount of steps is in the order of $\Theta(h \cdot \log(n))$ where h is the height of the tree.

You can prove that the **h** is independent from the t value and the total amount of time spent is in the order of $\Theta(\log(n))$.

Now let's consider our original purpose in the use of this kind of tree.

If I have $10^6 = 1.000.000$ keys, if I use an AVL tree to do a search, the number of access I will do is $\log_2(10^6) \simeq 20$.

If I use a Bayer tree of order t=100 instead the amount of accesses to storage I will do is $\log_{100}(10^6) \simeq 3$.

As you can see the improvement is significant.

6.5   Heap

An heap is an almost complete binary tree. It means that the tree is full from the root to the penultimate level. On the last level we use the convention of put the leaf on the left when it is possible.

You can prove that an almost complete binary tree has the height in the order of $\Theta(\log(n))$.

If we consider a max-heap, every key contained in every node is always $\geq$ of the keys contained in the children. Here below an example of max-heap.

```
            42
      12          40
    7     5    35    37
  4   3
```
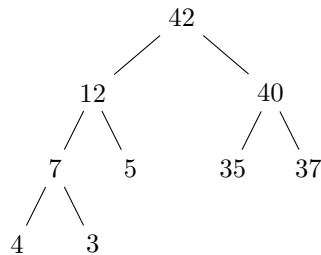
How to build an heap? There's a procedure called Heapify which use time in the order of $\Theta(\log(n))$. It starts from the bottom by fixing every subtree until the entire tree is an heap. We won't see the pseudocode of Heapify, but here below we're gonna see instead the one of createHeap.

```
Procedure createHeap(Tree T)
  h <-- height of T
  for p <-- h down to 0 do
    foreach node x of depth p do
      Heapify(subtree of root x)
```

We have to do call for every node $\Rightarrow$ n calls.
Heapify here costs $\Theta(h')$ comparisons where h' is the height of the subtree in the current iteration, $h' \leq h$.
By knowing this we can do a quick analysis and say that createHeap costs an amount of $\Theta(n \cdot h) = \Theta(n \cdot \log(n))$ comparisons because we know that $h \simeq \log_2(n)$.

7   Heap sort

We have studied all of these kind of trees just to reach this achievement. We can now use the last type, the heap, for sorting purpose.
The algorithm is called Heap sort and uses the concept of heap to sorting an array without using memory excluding the array in input.
The heap is "already" inside the array and we're gonna see how.

26	11	19	10	5	6	2

Starting from this array, we can build a tree by putting all the elements in "breadth". If you will do an inorder visit, you will get the order of the original array.
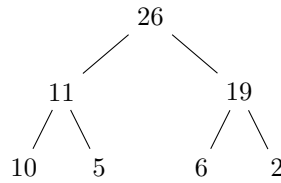
We are showing this tree for simplicity, but in the real case you can just use this rule:
The left child of a node is the one of index $2i + 1$, while the right one it is the one of index $2i + 2$.
How to sort? We start by removing the root and putting at the end of the array. After that, we call Heapify and then we remove again the root until our array is completely sorted.
The amount of comparisons is in the order of $\Theta(n \cdot \log(n))$ because we have to call n times the Heapify which use time in the order of $\Theta(\log(n))$
This algorithm is efficient such as Merge sort and Quick sort for comparisons, but it is the only one that doesn't use additional memory.

7.1 Can we go below $\Theta(n \cdot \log(n))$?

Well, if you wanna sort using comparisons, the short answer is **no**. Let's see why:

There are several sorting algorithm, which each of them use a different strategy to sort the elements.

If you build a decision tree of the comparisons an algorithm do, you will get a different tree for every algorithm.

Now, by knowing that permutation of n elements is equal to $n!$, when we visit this tree to reach the leafs we do an amount of comparisons in the order of $\log_2(n!)$.

By using the Stirling approximation we have that $n! \simeq \sqrt{2\pi \cdot n} \cdot \left(\dfrac{n}{e}\right)^n$

So: $\log(n!) \simeq \log\left(\sqrt{2\pi \cdot n} \cdot \left(\dfrac{n}{e}\right)^n\right)$ Let's do some manipulation and we get:

$\log_2\left(\sqrt{2\pi}\right) + \dfrac{1}{2} \cdot \log_2(n) + n \cdot \log_2(n) - n \cdot \log_2(e) \Rightarrow \Theta(n \cdot log(n))$ So we have proved that we can't do better if we do comparisons.

Anyway There are sorting algorithm that doesn't use comparisons to sort such as Bucket sort and Radix sort. They cost in time in the order of $\Theta(n)$, but we won't analyze them here.

8 Graph

A graph is a mathematical structure that describe relations and connections between couple of objects.

It is defined as $G = (V, E)$ where V is the set of nodes and $E \subseteq V x V$ is the set of links between nodes.

A graph can be oriented or not-oriented. The first one count the direction of the link, the second dosen't.

$(x, y) \in E \Rightarrow$ the link is incident on x and y. If the graph is oriented we say that this link is going out from x and it's entering in y, also y is adjacent to x.

The degree of a node is defined as: $\delta(v)$ the number of incident links on v.

*V=n and *E=m

$$\sum_{v \in V} \delta(v) = 2m$$

if the graph is not-oriented.

If oriented:

$\delta_{in}(v)$ is the degree "in"

$\delta_{out}(v)$ is the degree "out"

$\delta(v) = \delta_{in}(v) + \delta_{out}(v)$

$$\sum_{v \in V} \delta_{in}(v) = \sum_{v \in V} \delta_{out}(v) = m$$

Every link contribute to the degree "in" of a node and to the degree "out" of another node.

A path from x to y with $x, y \in V$ is:

A sequence $v_0, v_1, .., v_k \in V$ that:

$V_0 = x \quad v_k = y$ and $(v_{i-1}, v_i) \in E \quad i = 1, .., k$ The length of the path is equal to the number of links.

- y is reachable from x if $\exists$ a path from x to y.

- Simple path: there aren't repetitive nodes.

- Cycle: a path from x to x(closed path).

- Chain: it is equal to the path, but it doesn't matter the direction of the links.

- Circuit: it is a chain with $v_0 = v_k$.

- Connected graph: $\forall v_1, v_2 \in V \exists$chain.

- Strongly connected graph: $\forall v_1, v_2 \in V \exists$ path.

- hamiltoniam circuit: it is a circuit that go through every node once.

- Eulerian circuit: it is a circuit that go through every link once. $\exists$ an Eulerian circuit $\Leftrightarrow \forall v \in V \delta(v)$ is even.

8.1   Tree seen as a Graph

A tree as a graph is defined as following:

An not-oriented graph, connected and without any cycles.

A forest is a set of trees.

$G = (V, E) \; tree \Rightarrow *V = *E - 1$ Let's prove it by induction on n:

$n = *V$

$n = 1 \Rightarrow 1$ node and 0 links. OK

$n > 1 \Rightarrow$ I choose a node x and I remove it.

I obtain these graphs $G_1 = (V_1, E_1), .., G_k(V_k, E_k)$

If $*V_i < n$ then by using induction hypothesis $\Rightarrow *E_i = *V_i - 1$

$*V = *V_1 + \ldots + *V_k + 1$ This $+ 1$ is the x removed.

$*E = *E_1 + \ldots + *E_k + k$ This $+$ k are the links incident on x removed.

I use once again the induction hypothesis $\Rightarrow *E = *V_1 - 1 + *V_2 - 1 * \ldots + *V_k - 1 + k = *V - 1$ We have proved that a graph is a tree if containe the links -1 of the original graph.

9   Spanning Tree

A spanning tree is defined as follows:

$G = (V, E)$ not-oriented and connected, then a spanning tree is $G' = (V', E')$ with $V' = V$ and $E' \subseteq E$.

A mininum spanning tree is $T = (V, E_t)$ that:

we define in before the weight function $\omega : E \Rightarrow \mathcal{R}$

$$\omega(G') = \sum_{e \in E'} \omega(e)$$

- T is a tree with $E_t \subseteq E$

- $\forall\ T' = (V, E'_t)\ with\ E'_t \subseteq E \omega(T) \leq \omega(T')$

10   Greedy

The greedy technic is used to solve optimization problems that can require exponential time.

We have the P problem and a set C of candidates.

The goal require to find $S^* \subseteq C$ optimal solution.

I start from the $\emptyset$ and step by step I build the feasible solution $S \subseteq C$.

At every step I expand this partial solution already obtained.

The algorithm ends when it is not possible to expand anymore the solution obtained.

Rules of expansion:

- **Feasible solution**: The partial solution must satisfy the constraints.

- **Local optimal solution**: Between the candidates, I choose always the better in that moment.

- **Impossible to go back**: every choice is definitive and it is not possible to go back and think twice of what we have done.

10.1   Backpack problem

This is a common problem that can be solved by a greedy algorithm.
We have a backpack of height **h** and **k** objects $c_1, \ldots, c_k$ all of them with height $h_1, \ldots, h_k$.
The goal is choosing how many objects put in backpack without exceeding the height of the backpack.
Feasible solution: Any $S \subseteq (c_1, \ldots, c_k)$ that:
$f(S) \leq h$ with

$$f(S) = \sum_{c_i \in S} h_i$$

Optimal solution: Any $S^* \subseteq (c_1, \ldots, c_k)$ that: $f(S^*) \geq f(S) \forall S$

In this problem, the greedy algorithm doesn't find always the correct solution. There are a few cases where the solution found isn't the real optimal one.

11   Kruskal algorithm

This algorithm use the greedy strategy to find the minimum spanning tree starting from a weighted connected graph.
Kruskal starts by sorting the weight of all links. After that I have a forest consisting of all nodes alone without links.
I slide the list of links and I add the current link to the graph if and only if it doesn't create a cycle between nodes already linked.
Here below I let you read the pseudocode:

Listing 10:*Kruskal algorithm.*

```
Algorithm Kruskal(Graph G=(V,E,w)) --> tree
  Sort E using a sorting algorithm
  T <--(V,empty)
  foreach v in V do
    makeset(v)
  foreach (x,y) in E sorted do
    Tx <-- find(x)
    Ty <-- find(y)
    if Tx != Ty then
      union(Tx,Ty)
      add to T the link(x,y)
  return T
```

This algorithm is implemented by using the concept of partition with union/find algorithms.

These algorithms helps us knowing when two nodes are already linked or not.
The implementation of the graph is a simple link list.
Complexity:

- Suppose heapsort for sorting $\Rightarrow \Theta(n \cdot \log(n))$

- first foreach does n makeset which cost $O(1)$ each other $\Rightarrow O(n)$

- second foreach is doing m iterations:

  1. it does 2m find which cost $\Theta(log(n))$ each one by using QuickUnion balanced $\Rightarrow \Theta(m \cdot \log(n))$

  2. it does also n-1 union because a tree has n-1 links starting from a graph. Each union costs $O(1) \Rightarrow \Theta(n)$

If we consider that the graph is connected we have that:
$n - 1 \leq m \leq n^2$

So the total amount of time spent is in the order of $\Theta(m \cdot log(n^2)) + \Theta(m \cdot log(n)) \Rightarrow \Theta(m \cdot log(n))$

You can prove that Kruskal finds the minimum spanning tree always even if greedy strategy doesn't find the optimal solution in all problems.

12   Minimum path problems

$G = (V, E)$ oriented with $\omega : E \Rightarrow \mathcal{R}$
$\quad \Pi = < v_0, \ldots, v_k >$ a path from v_0 to v_k

$$\omega(\Pi) = \sum_{i=1}^{k} \omega(v_i)$$

weight of the path
$\Pi_{x,y}^*$ is the minimum path from x to y with $x, y \in V$ if:

- $\Pi_{x,y}^*$ is a path from x to y

- $\forall \Pi$ from x to y we have that: $\omega(\Pi_{x,y}^*) \leq \omega(\Pi)$

A property of minimum path:
If $\Pi_{x,y}^*$ is a minimum path from x to y that go through a node **v** then:

- the partial path from x to v is a minimum path from x to v.

- the partial path from v to y is a minimum path from v to y.

22

$$\Pi_{x,v}^* + \Pi_{v,y}^* = \Pi_{x,y}^*$$

This is called optimality principle.

There are three main type of mininum path problems:

1. Find the minimum path between two nodes. It doesn't exists an algorithm to solve this problem directly.

2. Find the minimum paths between all the couple of nodes. This is solved by **Floyd-Warshall** algorithm.

3. Find the minimum path between a node **s** and all the others. This is solved by **Bellman-Ford** and **Dijkstra** algorithms.

12.1   Floyd-Warshall algorithm

WIP

12.2   Bellman-Ford algorithm

WIP

12.3   Dijkstra algorithm

WIP