

## Costo computazionale

- Tempo di esecuzione
- Necessità di memoria

## Il tasso di crescita del tempo che impieghiamo nell'algoritmo si divide in tre notazioni

- Worst case  $O()$ 
  - abbiamo  $f(n) = 3n$  e il suo  $O(f(n)) = O(n)$  perchè abbiamo una  $c \geq 3$  tale che ci ritroviamo con un valore maggiore da  $n_0$  in poi " $n \geq n_0$ "  
 $n \leq O(n)$  o  $n \leq c \cdot n$  per un  $n \geq n_0$  e con una  $c \geq 3$
- Best case  $\Omega()$ 
  - abbiamo  $f(n) = 3n$  e il suo  $\Omega(f(n)) = \Omega(n)$  perchè abbiamo una  $c \leq 3$  tale che ci ritroviamo con un valore inferiore a  $f(n)$  da  $n_0$  in poi " $n \geq n_0$ "  
 $n \geq \Omega(n)$  o  $n \geq c \cdot n$  per un  $n \geq n_0$  e con una  $c \leq 3$
- Average case  $\Theta()$ 
  - il caso medio ci permette di dimostrare che senza cambiare quello che c'è all'interno di  $\Theta$  (un "generico") possiamo aggiungere anche  $\Omega()$  e  $O()$  quindi sappiamo che esiste una  $c'$  tale che  $f(n) \leq c' \cdot n$  ed esiste una  $c''$  tale che  $f(n) \geq c'' \cdot n$  e questo vale per un certo punto  $n_0$  abbastanza grande.

## Proprietà e dimostrazione dell'algebra della notazione asintotica

### 1. $\forall k > 0$ se $f(n) \rightarrow O(g(n))$ allora anche $k \cdot f(n) \rightarrow O(g(n))$ .

Dim

- sappiamo che esistono delle  $c$  tali che  $f(n) \leq c \cdot g(n) \quad \forall n > n_0$   
allora  $k \cdot f(n) \leq k \cdot c \cdot g(n) \implies k \cdot f(n) \leq c' \cdot g(n) \quad k \cdot f(n) \leq k \cdot c \cdot g(n) \quad \text{e } c' = c \cdot k$

### 2. $\forall f(n), d(n) > 0$ se $f(n) \rightarrow g(n)$ e $d(n) \rightarrow h(n)$ allora

$$f(n) + d(n) \rightarrow (O(g(n) + h(n)) = O(\max(g(n), h(n))))$$

Dim

- sappiamo che abbiamo quattro costanti  $c', c'', n_0'$  e  $n_0''$  tali che  
 $f(n) \leq c' \cdot g(n) \quad \forall n > n_0'$  e  $d(n) \leq c'' \cdot h(n) \quad \forall n > n_0''$  ed allora sappiamo che  
 $f(n) + d(n) \leq c' \cdot g(n) + c'' \cdot h(n) \leq \max(c', c'') \cdot (g(n) + h(n)) \quad \forall n > \max(n_0', n_0'')$   
cioè  $f(n) + d(n) \leq c \cdot (g(n) + h(n)) \quad \forall n > n_0 \implies f(n) + d(n) \rightarrow O(g(n) + h(n))$   
e per le proprietà della notazione asintotica sappiamo che dobbiamo prendere solo il massimo tra la somma di funzioni quindi  
 $f(n) + d(n) \rightarrow O(\max(g(n), h(n)))$

### 3. $\forall f(n), d(n) > 0$ se $f(n) \rightarrow g(n)$ e $d(n) \rightarrow h(n)$ allora

$$f(n) \cdot d(n) \rightarrow O(g(n) \cdot h(n))$$

Dim

- sappiamo che abbiamo quattro costanti  $c', c'', n_0'$  e  $n_0''$  tali che  
 $f(n) \leq c' \cdot g(n) \quad \forall n > n_0'$  e  $d(n) \leq c'' \cdot h(n) \quad \forall n > n_0''$  ed allora sappiamo che  
 $f(n) \cdot d(n) \leq (c' \cdot c'') \cdot (g(n) \cdot h(n)) \quad \forall n \geq \max(n_0', n_0'')$  allora  
 $f(n) \cdot d(n) \rightarrow O(g(n) \cdot h(n))$  con la  $c = c' \cdot c''$  e  $n = \max(n_0', n_0'')$

### Verifica della notazione asintotica

Attraverso i limiti possiamo capire se la nostra notazione è corretta o sbagliata e questa si può verificare per i diversi casi che può verificare un limite  $\rightarrow +\infty$  di  $f(n) \rightarrow O/\Omega/\Theta(g(n))$

1.  $\lim_{n \rightarrow +\infty} f(n)/g(n) = n \in \mathbb{R}$  allora  $f(n) \rightarrow \Theta(g(n))$  ovviamente anche  $O$  &  $\Omega$
2.  $\lim_{n \rightarrow +\infty} f(n)/g(n) = 0$  allora  $g(n) \rightarrow +\infty$  più velocemente di  $f(n)$   $f(n) \rightarrow O(g(n))$
3.  $\lim_{n \rightarrow +\infty} f(n)/g(n) = 0$  allora  $f(n) \rightarrow +\infty$  più velocemente di  $g(n)$   $f(n) \rightarrow \Omega(g(n))$

### Sommatoria:

1.  $i=0 \rightarrow n$  di  $\sum i = n * ((n+1)/2) = \Theta(n^2)$
2.  $i=x \rightarrow n$  di  $\sum c^i = (c^{(n+1)} - c^x) / (c - 1) = \Theta(c^n)$

### Calcolo costo Computazionale

1. I for rispettano il range o il numero di elementi in un array/qualsiasi altra cosa.
2. I while hanno diverse modalità e condizioni come while  $n > 1$  (o qualsiasi numero) :
  - a.  $n // x$  con  $x$  un qualsiasi numero  $O(\log(x) n)$
  - b.  $n - x$  con  $x$  un qualsiasi numero  $O(n)$
  - c.  $i = 1$   $i * i \geq n$  con  $i += 1$  allora abbiamo  $O(n^{1/2})$
3. I for/while con for/while annidati, dobbiamo calcolare le informazioni e poi moltiplicare il tutto
4. In caso di un caso migliore e peggiore, prendiamo innanzitutto quello maggiore e quello che si ripete al crescere di  $n$  sennò poniamo  $c$  come il max.

### Ricorsione

La ricorsione è ispirata dalle funzioni ricorsive, e serve per risolvere un problema in un modo naturale come ad esempio trovare il fattoriale di  $n$  cioè  $n * (n-1)!$ , quest'ultimo avrà come caso base  $n == 0$  return 1.

#### Pro della ricorsione:

- la soluzione è più naturale comparata con la soluzione iterativa.

#### Contro della ricorsione:

- Utilizza molta memoria perché si deve ricordare i casi precedenti .
- Non è scalabile all'infinito a causa del grande utilizzo della memoria.

## Calcolo costo computazionale per la ricorsione

### 1. Metodo Iterativo:

- Sostituiamo all'interno della  $f(n)$  fino a quando non arriviamo a  $T(1)$
- $T(n) = T(n-1) + \Theta(1)$  con  $T(1) = \Theta(1)$
- $T(n) = [T(n-2) + \Theta(1)] + \Theta(1) \dots T(n) = [T(n-3) + \Theta(1)] + \Theta(1) + \Theta(1) \dots$
- da qui possiamo dire che dobbiamo trovare un  $k$  tale che  $n-k = 1$  e con un paio di semplificazioni arriviamo che  $k = n-1$  e questo è il limite della sommatoria ed infine ad ogni  $T(n)$  abbiamo un  $+\Theta(1)$  quindi la nostra sommatoria sarà  $i=0 \rightarrow n-1 \sum +\Theta(1)$  che sarebbe  $\Theta(n) - \Theta(1)$

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

$$T(n) = 2[2T\left(\frac{n}{4}\right) + \Theta\left(\frac{n}{2}\right)] + \Theta(n)$$

$$T(n) = 2[2[2T\left(\frac{n}{8}\right) + \Theta\left(\frac{n}{4}\right)] + \Theta\left(\frac{n}{2}\right)] + \Theta(n)$$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} \frac{n}{2^i} \cdot 2^i \quad k = \log_2 n$$

$$T(n) = n \cdot \Theta(1) + \sum_{i=0}^{\log_2 n - 1} n$$

$$T(n) = \Theta(n) + n + n + \dots + n$$

$$T(n) = \Theta(\log_2 n \cdot n) = (\log_2 n^2)$$

## 2. Metodo di Sostituzione:

- È consigliata al fine di dimostrare la veridicità di un costo computazionale.
- $T(n) = T(n-1) + c \quad T(1) = d$
- Il nostro Principale obiettivo è quello di verificare il  $\Omega$  e  $O$ , al fine di poter dire che  $T(n) \leq k \cdot n$  ( $O$ ) e  $T(n) \geq h \cdot n$  ( $\Omega$ ) e in tutto questo avremmo un  $T(1) = d$  tale che per ( $O$ )  $d \leq k$  e per ( $\Omega$ )  $d \geq h$ .
- Andiamo a vederlo per  $O(n)$  della  $T(n)$  sopra, sappiamo che la  $k \geq d$ , e che  $T(n) = k \cdot (n-1) + c$  e che  $T(n) \leq k \cdot n$ , quindi  $k \cdot n - k + c \leq k \cdot n \implies k \geq c$ .  
quindi sappiamo che  $T(n) = O(n)$  se e solo se  $[k \geq c]$  e  $[k \geq d]$ .
- Questo vale anche per  $\Omega(n)$  ma con  $k \cdot n \leq T(n)$ .

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \quad T(1) = O(1)$$

$$T(n) = O(n \cdot \log n) \implies T(n) \leq k \cdot (n \cdot \log n)$$

$$T(n) = 2T\left(\frac{n}{2}\right) + c \cdot n \quad T(1) = d$$

① Caso Base  $n=1$

$$T(1) \leq k \cdot 1 \cdot \log_2 1 \stackrel{0}{=} d \leq 0 \quad \text{FALSO}$$

$d > 0$  per  
DEFINIZIONE

$$O(n \cdot \log n) = k \cdot n \log n + h \cdot n$$

$$d \leq h \quad \text{OK!}$$

② Passo induttivo ASSUMIAMO CHE  $\forall m < n \quad T(m) = O(n \log n)$

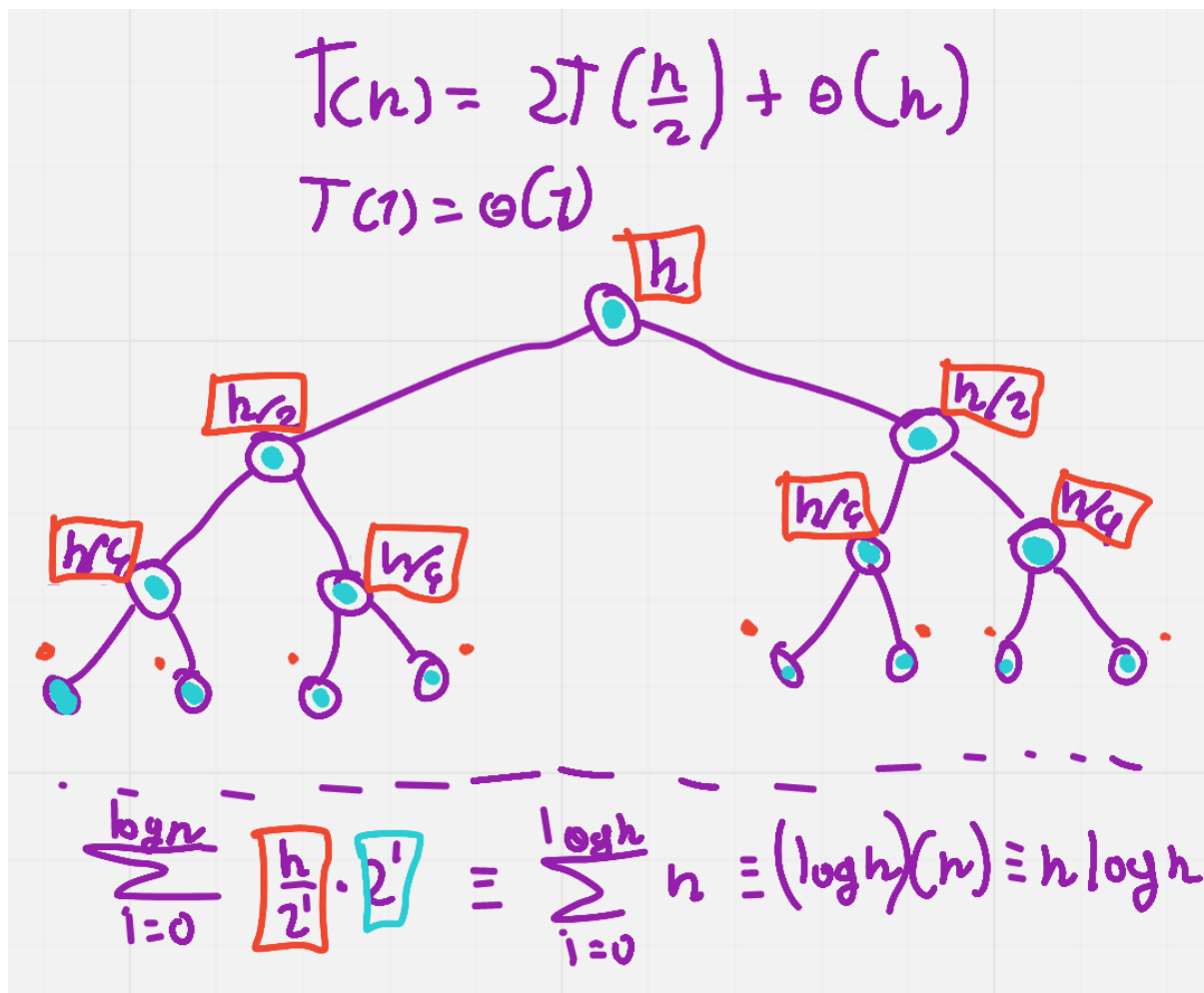
$$T(n) \leq 2\left(k \frac{n}{2} \log \frac{n}{2} + h \frac{n}{2}\right) + c n \equiv k n (\log n - \log_2 2) + h n + c n \quad \text{CINQUE FATTORI}$$

$$\underline{k n \log n} - k n + \underline{h n} + c n \leq \underline{k n \log n} + \underline{h n} \equiv c n \leq k n$$

$$c \leq k \quad \text{OK!}$$

### 3. Metodo dell'Albero

- È molto simile al Metodo Iterativo, soltanto che qui cerchiamo di dare anche una rappresentazione grafica al fine di capire come si sviluppa la ricorsione sotto forma d'albero.
- $T(n) = T(n-1) + \Theta(1)$
- disegniamo il costo per  $T(n) = \Theta(1)$  e lo colleghiamo tramite un nodo a  $T(n-1)$  e gli diamo  $T(n-1) = \Theta(1)$  e così via alla fine il avremmo un albero con dei valori ai quali in alcuni casi gli potremmo assegnare delle variabili, e dobbiamo scoprire quanto ci mette  $T(n) \rightarrow T(1)$  in questo caso  $n-1$  Time quindi avremo  $i=0 \rightarrow n-1 \sum \Theta(1)$  che sarebbe  $\Theta(n) - \Theta(1)$
- Ora vediamo  $T(n) = 2T(n/2) + \Theta(n)^2$  con  $T(1) = \Theta(1)$
- In cima abbiamo  $\Theta(n)^2$ , al 2° livello  $= 2*(n/2)^2$ , al 3° livello  $= 4*(n/4)^2$ , da qui vediamo una ripetizione di  $2^i * (n/2^i)^2 == n^2 / 2^i$ , e per vedere dove dobbiamo fermarci sappiamo che  $n/2^k = 1$  quindi  $n = 2^k$  1 quindi  $k = \log n$  e abbiamo  $i=0 \rightarrow \log n \sum n^2/2^i$  che sarebbe  
(siccome non dipende da  $i$ )  $n^2 * \sum 1/2^i$  che sarebbe  $\Theta(n^2) * \text{un num finito}$ .



#### 4. Metodo Principale:

- È molto semplice ma abbiamo delle condizioni da rispettare come:
  - $T(n) = a \cdot T(n/b) + f(n)$  e  $T(1) = \Theta(1)$ .
- $S = n^{\log_b(a)}$ .
- Se  $f(n) = O(n^{\log_b(a) - \epsilon})$  per qualche costante  $\epsilon$  tale che  $S = n^{\log_b(a) - \epsilon}$  allora  $T(n) = O(n^{\log_b(a)})$
- Se  $f(n) = \Theta(n^{\log_b(a)})$  allora  $T(n) = \Theta(n^{\log_b(a)} \log n)$
- Se  $f(n) = \Omega(n^{\log_b(a) + \epsilon})$  per qualche costante  $\epsilon$  tale che  $S = n^{\log_b(a) + \epsilon}$  e se  $a \cdot f(n/b) \leq c \cdot f(n)$  per qualche costante  $c < 1$  allora  $T(n) = \Theta(f(n))$

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \quad T(1) = \Theta(1)$$

$$a=2 \quad b>2 \quad f(n) = n$$

$$n^{\log_2 2} = n^1 = f(n)$$

2 caso

$$f(n) = \Theta(n^{\log_2 2})$$

$\Theta(n \log n)$

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

### Linked List (liste puntate)

Le linked list sono formate da due valori principali, uno rappresenta il valore del puntatore e un altro il valore nella memoria del prossimo puntatore, in questo modo riusciamo ad avere una complessità spaziale costante a differenza degli array.

#### - Caratteristiche delle linked List

Le linked list sono abbastanza complesse da utilizzare visto che al fine di ritornare all'inizio dobbiamo utilizzare la ricorsione o salvarci da qualche parte il puntatore iniziale, infine trova, elimina e inserisce elementi in  $O(n)$ , apparte gli inserimenti alla fine della linked list che avviene in  $\Theta(n)$

### - Double Linked List (Liste doppiamente puntate)

A differenza delle normali Linked List, quest'ultime oltre a puntare al prossimo elemento, puntano anche al precedente, ovviamente il primo elemento ha come precedente None, quest'ultima cosa è molto utile nel caso dell'eliminazione di elementi/puntatori, ad esempio abbiamo  $\text{None} \leftrightarrow 1 \leftrightarrow 2 \leftrightarrow 3 \leftrightarrow 4 \rightarrow \text{None}$  se abbiamo il puntatore dell'intero 2, basterà dire che il next di 1 sarà 2.next e dire che il precedente di 3 sarà il precedente del 2, tutto questo in modo costante.

### Stack (pila)

Lo stack è rappresentato come una lista puntata, ma lui si ricorda sempre dell'ultimo puntatore, semplicemente riportando sempre l'ultimo puntatore(quello che ha come next None), lo stack è utile perché riusciamo a estrarre l'ultimo elemento in tempo costante, e l'inserimento viene fatto in tempo costante, perché possiamo inserire gli elemento sono in ultima posizione.

Nel complesso lo stack è poco utilizzato nei problemi informatici, ma dove è veramente utile fa la differenza, lascerò qua sotto un codice in python che ci permette di visitare un albero in modo iterativo grazie allo stack.

```
[
def visitTree(root):
    stack = [] #utilizziamo la lista di python come stack utilizzando append() e pop()
    counterOfElements = 0 #numero di elementi nello stack
    print(root.val)
    while (counterOfElements != 0 or root != None):
        if root == None or root.left == None:
            root = stack.pop()
            counterOfElements -= 1
        else:
            root = root.left
            if root.right != None:
                counterOfElements += 1
                stack.append(root.right)
]
```

### HashSet

Gli HashSet sono delle strutture dati che fanno uso degli array e formule matematiche, la costruzione di un HashSet può avvenire in diversi modi, io descriverò, uno visto in privato, quindi la prima cosa è costruire un array lunga 4 e ogni volta che quest'ultima ha occupati i  $\frac{3}{4}$  della sua capienza massima, ricreiamo una nuova array lunga il doppio, la cosa più negativa è che dobbiamo reinserire tutti gli elementi di nuovo, questo fa sì che il costo computazionale sia pari a  $O(n)$ , ma questo vuol dire allo stesso tempo che riusciamo a estrarre ed eliminare elementi in  $O(1)$ , quindi ci sono vantaggi e svantaggi nell'HashSet, una possibile soluzione sarebbe di creare direttamente un array lunga  $n$  al fine di non ricalcolare ogni volta l'array lungo  $2^h$ , bene ora vediamo come vengono inseriti questi elementi, in

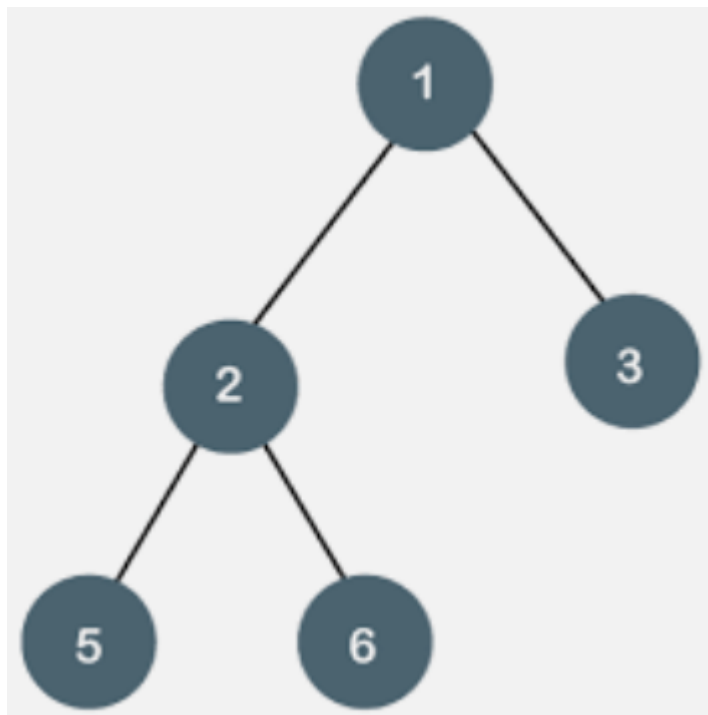


pratica è stata creata questa funzione  $f(x)$  che prende come input un elemento e lo trasforma in un numero, da qui facciamo il % di python che fa  $f(\text{input}) \% \text{sizeArray} = k$ , questo  $k$  rappresenta la posizione dell'array dove verrà inserito l'input, ovviamente la  $f(x)$  è la più randomica possibile sui moduli, ma allo stesso tempo sappiamo che riusciamo a ottenere da  $f(\text{input}) = l$ ,  $h(l) = \text{input}$ , e questa cosa è la caratteristica principale dell'hashSet, ovviamente nel caso peggiore impiegheremo  $O(n)$  nel trovare un elemento. Un modo che peggiora diminuisce il caso peggiore sarebbe considerare l'array della posizione  $k$  come un albero binario di ricerca con l'implementazione di alberi black and red, al fine di avere un albero binario di ricerca equilibrato, e al fine di inserire e ricercare elementi in un max di  $O(\log_2 n)$ .

## Alberi binari

Gli alberi binari sono di fatto degli elementi che hanno un valore e possono puntare a due diversi elementi anche chiamati left, right, per esempio la radice dell'albero avrà una parte left e una parte right (non escludiamo possano essere uguale a None), e a loro volta se diversi da None, avranno una parte left e right, e così via.

Ci sono diverse possibili visite per gli alberi, lascerò qui sono il codice in python



inorder Visit:

[5, 2, 6, 1, 3]

preorder Visit:

[1, 2, 5, 6, 3]

postorder Visit:

[5, 6, 2, 3, 1]

```

if root.left == None and root.right == None:
    print(root.val)
    pass

if root.left != None:
    inorderVisit(root.left)

print(root.val)
  
```

```

[
def inorderVisit(root):
  
```

```

        if root.right != None:
            inorderVisit(root.right)

        pass

def preorderVisit(root):
    if root.left == None and root.right == None:
        print(root.val)
        pass

    print(root.val)
    if root.left != None:
        preorderVisit(root.left)
    if root.right != None:
        preorderVisit(root.right)

    pass

def postorderVisit(root):
    if root.left == None and root.right == None:
        print(root.val)
        pass

    if root.left != None:
        postorderVisit(root.left)
    if root.right != None:
        postorderVisit(root.right)

    print(root.val)
    pass
]

```

### Alberi Binari di ricerca

Gli alberi binari di ricerca sono fatti al fine di ricercare un elemento in un tempo inferiore a  $O(n)$ , noi non riusciamo a rispettare completamente questa richiesta con un semplice albero di ricerca, però prima vediamo la struttura di questo albero, la radice può essere un numero qualunque presente ad esempio nel nostro array, ad esempio 35, e lo inseriamo come radice del nostro albero, poi prendiamo 45, e per inserirlo in questo albero dobbiamo fare i controlli sulle radici, ad esempio 45 è maggiore di 35 quindi va messo nella parte destra di 35 e questi controlli vanno fatti fino a quando non ci sarà un left o right uguale a None, and esempio se come figlio destro di 35 avessimo avuto 48, avremmo dovuto spostare 45 nella parte sinistra di 48, quest albero ricorda di fatto il quicksort, e con una visita inorder avremo

gli elementi in ordine crescente, come il quicksort ha il problema del tempo computazionale uguale a  $O(n^2)$  nel caso peggiore, fortunatamente però lo scopo del nostro albero non è quello di ordinare gli elementi, ma di inserirli e ricercare elementi in tempo  $O(\log^2 n)$ , questo è possibile con gli alberi red and black, che ci permettono di equilibrare il nostro albero in tempo al massimo  $O(\log n)$ , quindi di conseguenza una ricerca e un inserimento molto efficaci, questo tipo di struttura dati serve per supportare algoritmi o strutture dati come l'hashset al fine di migliorare il loro caso medio/peggiore, quindi è molto difficile avere ritrovarsi a risolvere problemi con queste strutture dati.