

ALGORITMI E STRUTTURE DATI

- ALGORITMI E STRUTTURE DATI
 - Esame:
 - Cos'è un algoritmo?
 - Problema dell'ordinamento:
 - Algoritmi che vedremo:
 - InsertionSort:
 - correttezza:
 - Correttezza InsertionSort:
 - Dimostrazione (per induzione):
 - Dimostrazione correttezza:
 - Valutazione complessità dell'algoritmo:
 - La stabilità:
 - MergeSort:
 - Caso base:
 - Passo induttivo:
 - InsertSort:
 - Stime di somme
 - ALBERI BINARI
 - Vari casi:
 - Esempio:
 - Cancellazione:

Esame:

Parte scritta orale e laboratorio (lab va fatto prima dell'orale) (probabilmente l'orale è obbligatorio)

5 appelli:

- 2 compitini + orale (fine febb e inizio metà giugno, orale giugno o luglio)
- II app con scritto in luglio
- III met/fine luglio
- IV scritto settembre
- V scritto gennaio/febbraio

Per l'orale serve aver preso almeno 16 nello scritto.

Nei compitini nella media viene arrotondata per eccesso

Fare male il primo compitino non preclude il secondo

Tutte le altre informazioni sono nella pagina e-learning (ancora da fare) Si trovano:

- Le regole d'esame
- Alcuni esami scritti passati
- Qualche esercizio risolto
- Quiz di autovalutazione (vengono pubblicati dopo ogni argomenti, molto più semplici degli esami)
- Qualche link utile a siti per fare simulazioni con alcune strutture dati
- Riferimenti bibliografici
- Tutte le registrazioni e trascrizioni durante le lezioni dell'anno durante la pandemia

Teams viene usato solo per le emergenze

L'altro professore usa di più teams

Il ricevimento è su appuntamento tramite mail (è sospeso i 7 giorni lavorativi precedenti ad uno scritto).

Il libro di testo che si segue è "cormen leiserson Rivest (...) Introduction to Algorithms"

3 requisiti per l'esame

- Programmazione del primo anno (soprattutto per la ricorsione)
- Matematica discreta e analisi (non c'è un blocco, ma è molto consigliato)

Il laboratorio erano esercizi obbligatori da consegnare e un progetto (forse il progetto verrà cambiato) è obbligatorio lavorare in gruppo

Cos'è un algoritmo?

Una sequenza ordinata di operazioni \rightarrow modelli di calcolo (che tipo di macchina si utilizza) Si deve ragionare su correttezza e complessità e si deve trovare un problema da risolvere. Per risolvere un problema si deve:

avere una funzione che dice cosa ho in input e cosa ho in output.

P: input \rightarrow output

A: input \rightarrow output

Le due funzioni devono coincidere e quindi devo dimostrare che quello che ho scritto è quello che deve fare.

Nel corso creeremo dei problemi, creeremo degli algoritmi per risolverli e verrà valutato il tempo per risolverli.

Il modello di calcolo che useremo sarà pseudocodice (molto facile da tradurre in un linguaggio qualsiasi)

Problema dell'ordinamento:

problema: dato un vettore A contenente n numeri interi, determinare una permutazione π degli indici di A tale che

$$\forall i, j, i < j \implies A[\pi[i]] < A[\pi[j]]$$

A vettore Struttura dati statica (dimensione fissa) composta da elementi omogenei, è indicizzato (accesso diretto)

Cos'è una permutazione? (uno scambio tra gli indici – una funzione iniettiva) Un vettore A di lunghezza n ha un indice che va da 1 a n In input avremo il vettore di interi.

Gli input in questo problema sono tutti i possibili vettori e la dimensione dell'input è la lunghezza del vettore.

Per calcolare la complessità-tempo (A, M): $\forall \text{ input} \rightarrow \mathbb{R}^+$ Questo processo sarebbe particolarmente complicato, quindi si farà riferimento al caso peggiore, al caso medio e al caso migliore.

$T_p(A, M): \mathbb{N}$ (di dimensione n) $\rightarrow \mathbb{R}^+$

Tempo nel caso peggiore: $T_p(A, M)(5)$ quanto tempo richiede nel caso peggiore l'algoritmo A su un generico vettore di dimensione 5 sulla macchina M.

Calcoleremo anche la complessità nel caso migliore.

Ogni operazione di base ha un costo costante c (Che cambia di macchina in macchina), si ragiona in termini asintotici, in modo che non si specifichi questo tempo c , e si prende la funzione che tende $+\infty$

Algoritmi che vedremo:

Su cambi e confronti: **InsertionSort**, **MergeSort**, **HeapSort**, **QuickSort** di cui valuteremo:

- complessità, scambi e confronti
- algoritmi con ipotesi aggiuntive nell'input

Vedremo anche:

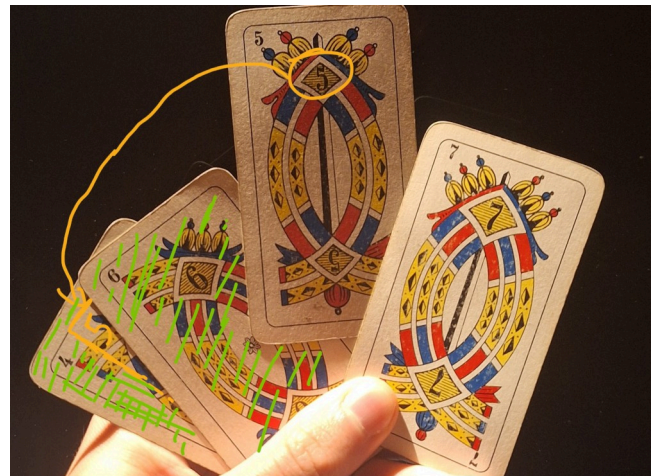
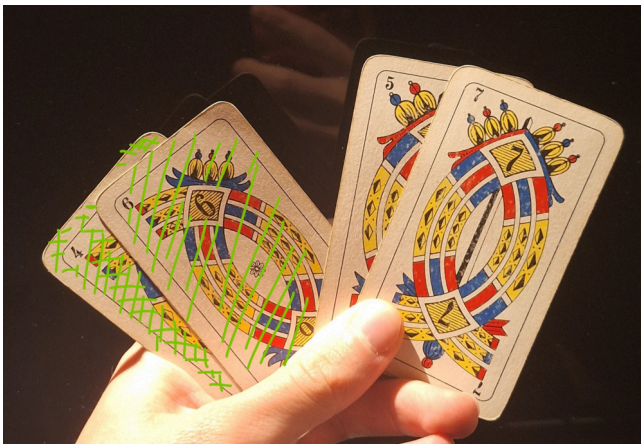
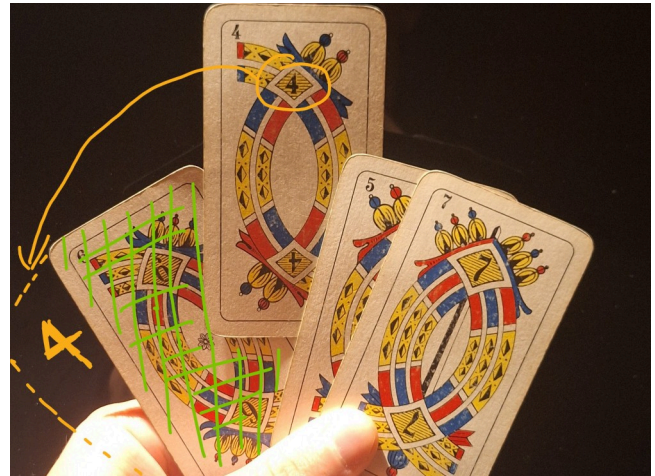
- algoritmi sul calcolo della mediana e altre statistiche
- liste, pile, code, code con priorità,
- tabelle di Hesh
- Alberi binari di ricerca
- Alberi bilanciati
- B-alberi
- Insiemi disgiunti
- Grafi (non pesati e pesati)

InsertionSort:

Idea:

I numeri che vengono ordinati vengono anche chiamati **keys** (chiavi)

Il metodo è simile a quello intuitivo con cui si ordina un mazzo di carte:




```

InsertionSort(a) {
  // j parte da 2 perché il primo elemento (a[1]) è già "ordinato" per
  // definizione
  for j <- 2 to a.length {

    key <- a[j]      // Salviamo il VALORE da inserire
    i <- j - 1       // Iniziamo a confrontare con l'elemento a sinistra

    // Sposta gli elementi di a[1..j-1] che sono più grandi di key
    // di una posizione verso destra
    while (i > 0 and a[i] > key) {
      a[i + 1] <- a[i]
      i <- i - 1
    }

    // Inserisce key nella posizione corretta
    a[i + 1] <- key
  }
}

```

Dopo aver verificato che il codice funziona con un caso “semplice”, dobbiamo controllare la correttezza e la complessità.

Per controllare la correttezza, dobbiamo definire la funzione con dei passi di computazione, che vanno dall’input all’output.

correttezza:

P: input \rightarrow output

A: input \rightarrow output (sequenza di istruzioni che termina, \forall input)

Come primissima cosa va dimostrato che termina sempre, gli algoritmi ci permettono di affrontare argomenti decidibili (che si può capire la risposta).

Correttezza InsertionSort:

\forall vettore A InsertionSort (A) termina e al termine A è ordinato in ordine crescente.

$\forall A, b \in [1, A - length] a < b \implies A[a] \leq A[b]$

(e A contiene gli stessi elementi che contiene inizialmente)

Fare attenzione che la correttezza va controllata all’interno del codice, non dell’idea iniziale

La correttezza è il teorema, mentre l’invariante (in questo caso il ciclo for) è come se fosse un lemma ed è il primo elemento che va controllato per la dimostrazione. Nella invariante va codificato matematicamente tutte le proprietà che credo siano vere tutte le volte che passo per

la riga di codice.

All'inizio della J-esima iterazione del ciclo for $A[1, \dots, j-1]$ è ordinato

Dimostrazione (per induzione):

Base:

$j=2$ $A[1, \dots, j-1] = A[1, \dots, 1]$ un vettore che contiene un solo elemento è un vettore ordinato.

Passo induttivo:

ipotesi induttiva: invariante vera all'inizio dell'iterazione j $A[1, \dots, j-1]$ è ordinato

Tesi:

invariante vera all'inizio dell'invariante $j+1$

All'inizio dell'iterazione $j+1$ $A[1, \dots, j]$ è ordinato

Devo quindi analizzare il codice e vedere cosa succede durante la j -esima iterazione del "for"

Un nuovo valore (chiamato x) viene salvato in $key=x$

Quindi $i=x-1$

Di conseguenza i si ferma su un valore $y < x$, tutti gli altri elementi vengono copiati a "destra" di una posizione

All'uscita del while:

(ordinato) (ordinato) (ordinato) x y h (ordinato) (ordinato)

Dimostrazione correttezza:

InsertionSort (A) termina.

Il while termina sempre perché all'interno del while l'indice i viene decrementato e una delle guardie del while richiede che $i > 0$. Il ciclo for termina perché j non è stata modificata all'interno del for. J viene aumentata di 1 ad ogni iterazione del for e la condizione richiede che $j \leq A.length$

InsertionSort (A) termina quando la testa (o condizione) del ciclo for viene eseguita con $j = A.length + 1$ e il test del for fallisce all'inizio dell'iterazione $A.length+1$

Sfruttando l'invariante:

$A[1, \dots, (A.length + 1)]$ è ordinato e corrisponde ad A

Valutazione complessità dell'algoritmo:

Calcolo caso peggiore:

$Tp : N \implies R_+$ // $Tp(n)$: tempo impiegato nel caso peggiore su un vettore di lunghezza n .

Viene utilizzato il criterio di **costo uniforme** (assumo che ogni istruzione di base del nostro linguaggio di programmazione abbia un costo costante = c)

Le **istruzioni di base** sono: operazioni aritmetiche, assegnamenti, incrementi, test booleani, ecc.. questa variabile è indipendente dalla complessità delle singole operazioni svolte (un esempio $1 + 1$ viene considerato come un c , come sarebbe $5432 * 232332$), per questo motivo questo principio non sempre dà un risultato corretto, ma per questo corso è sufficiente per i casi che vedremo (per casi complessi si dovrebbe usare un sistema di calcolo logaritmico – tiene conto il numero di bit utilizzati)

Anche in caso di lunghezza n fissa, il costo può variare per quanto è disordinato un vettore.

Esempio:

$[2 \ 1 \ 3 \ 3] = [15 \ 10 \ 20 \ 12] = c$ // *sono disordinati nello stesso modo, quindi hanno la stessa complessità*

$[1 \ 2 \ 3 \ 4] \neq [3 \ 4 \ 2 \ 1] \neq c$ // *Il caso peggiore è n ordinato in ordine opposto il caso migliore è un vettore già ordinato*

Per il caso migliore: $[2, \dots, n-1] = ((n+1)-2)+1 = n$ (numero di volte che il codice viene eseguito)
 $C \cdot n$

$C(n-1)$ (\forall linea di codice)

Il while ha 2 controlli booleani:

- $c \implies c \cdot T_j$ (j = numero di volte che il while viene eseguito)
- La riga sotto il while: $\sum_{j=2}^n (c \cdot (T_j - 1))$

sommando tutto:

$$Tp(n) = c \cdot n + 3c(n-1) \sum_{j=2}^n (c \cdot T_j) + 2 \sum_{j=2}^n (c \cdot (T_j - 1))$$

Sostituendo con numeri:

Nel caso più **fortunato**: $T_j = [1]$

Nel caso **peggiore**: $i=[0, \dots, j-1] \implies$ quindi viene eseguito j volte

Ergo:

$$Tp(n) = c \cdot n + 3c(n-1) + \sum_{j=2}^n (c \cdot T_j) + 2 \sum_{j=2}^n (c \cdot (T_j - 1))$$

ricordando la somma di gauss: $\sum_{k=1}^m k = \frac{m(m+1)}{2} = \Theta(m^2)$

Nel caso migliore: $T_j = 1$

$TO(n) = \Theta(n)$ // dovranno essere distinti gli algoritmi nel caso siano inplace o non inplace (se usano spazio ausiliario o no)

La stabilità:

Un algoritmo di ordinamento è stabile se preserva l'ordine iniziale.

| | | | | | | |
|-------|-------|-------|-----------|-------|-------|-------|
| (...) | (...) | (...) | 5a | (...) | (...) | (...) |
| (...) | (...) | (...) | (...) | 5a | 5 | (...) |

Un esempio è quando abbiamo un insieme di nomi e cognomi (ordinati per cognome), e vogliamo ordinarli anche per nome, è importante che l'ordinamento del cognome rimanga persistente e successivamente venga applicato l'ordinamento per il nome nel caso in cui due cognomi inizino con la stessa lettera.

Nel nostro algoritmo il while è stato scritto con:

```
(...)
while (i > 0 && a[i] > key){
    (...)
```

E va quindi analizzato per determinare se sia o meno stabile

MergeSort:

usa una procedura Divide et Impera

- Divido in sottoproblemi
- Risolvo i sottoproblemi (con ricorsione)
- Combino le soluzioni di sottoproblemi per una soluzione del problema generale

Caso base:

Se il vettore ha lunghezza 1 è già ordinato

Passo induttivo:

due sottoproblemi risolti (esempio: tutti i numeri pari ordinati e tutti i numeri dispari ordinati)

Idea:

Abbiamo un vettore da ordinare, lo dividiamo in due metà (ricorsivamente) fino ad arrivare a vettori di lunghezza 1 (caso base), e poi uniamo i vettori ordinati (merge).

(...) **P** (...) **r** (...) **q** (...)

$$r = \frac{q+p}{2}$$

Determiniamo quindi 2 sottoproblemi e 2 sottovettori da riordinare:

Una volta ottenuti i due vettori ordinati, dobbiamo unirli in un unico vettore ordinato (merge - un nuovo DLC di LLCS).

```
MergeSort(A, p, q) {
  if (p < q) {
    // Calcola il punto medio (divisione intera)
    r <- floor((p + q) / 2)

    // Ordina ricorsivamente la prima metà
    MergeSort(A, p, r)

    // Ordina ricorsivamente la seconda metà
    MergeSort(A, r + 1, q)

    // Fonde le due metà ordinate
    Merge(A, p, r, q)
    // qui andrebbe definita la funzione Merge
  }
}
```

Il caso base sussiste qualora il caso base sia $q \leq p$, dove il programma termina perché il vettore è stato ordinato

Esempio:

10(1) 20 5 2(4)

```
MergeSort (A, 1, 4){
  r<-2
  Mergesort(A,1,2) // viene diviso e ne abbiamo due sotto vettori
  r<-1
  Mergesort(A,1,1) // caso base quindi termina (1=1)
  Mergesort(A,2,2) // caso base come sopra
  Viene fatta quindi la chiamata
  Merge (A, 1, 1, 2) (copia prima il più piccolo, e poi il più grande, crea
```

```
quindi un vettore ordinato)
}
```

Mi trovo quindi con $A[10,20]$ che viene unito al resto (che viene ordinato nello stesso modo)

```
MergeSort(A,3,4){
    r<-3
    Mergesort(A,3,3) // caso base
    Mergesort(A,4,4) // caso base
    Merge(A,3,3,4) // unisce i due vettori su un vettore ausiliario B, mettendoci
    prima il numero minore, poi il numero maggiore
}
```

Merge ora prende i vari vettori ordinati, e tramite LLCS copia il più piccolo di ogni vettore, uno dopo l'altro in maniera ricorsiva, fino a quando non terminano entrambi i vettori

Correttezza Merge:

Merge(A,p,r,q) termina e se all'inizio $A[p\text{---}r]$ e $A[r+1\text{.....}q]$ sono ordinati, allora al termine $A[p,\text{.....},q]$ è ordinato

Correttezza MergeSort:

MergeSort(A,p,q) termina sempre con $A[p\text{....}q]$ ordinato

Dimostrazione correttezza MergeSort:

Per induzione sul numero di ricorsione:

Base:

0 chiamate ricorsive: $p \geq q$ $A[p\text{....}q]$ ha lunghezza ≤ 1 ed è quindi ordinato

Passo induttivo:

Mergesort(A,p,q) termina con $A[p\text{...}q]$ ordinato se vengono effettuate al più m chiamate

Mergesort(A,p',q') termina con $A[p'\text{.....}q']$ ordinato se vengono eseguite $m+1$ chiamate

Mergesort(...)

$$m + 1 > 0 \Rightarrow p < q \Rightarrow p \leq r < q$$

quindi MergeSort farà al di più $m+1$ chiamate e quindi termina l'ipotesi induttiva con la correttezza di merge

$$\log_e n \in O(n)$$

Prop:

$$\lim_{x \rightarrow \infty} \frac{g(n)}{f(n)} = g(n) \in O(f(n))$$

$$\forall E > 0 \exists M \forall n \geq M \frac{g(n)}{f(n)} \leq E f(n)$$

$$\lim_{x \rightarrow +\infty} \frac{g(n)}{f(n)} = b \implies g(n) \in O = (f(n))(b \neq \infty)$$

Proprietà:

$$g_1(n) \in O(f_1(n)) \vee g_2(n) \in O(f_2(n)) \implies g_1(n) + g_2(n) \in O(f_1(n) + f_2(n))$$

$$f(n) \in O(f(n))$$

$$g(n) \in O(f(n)) \vee f(n) \in O(h(n)) \implies g(n) \in O(h(n))$$

si usa O come limitazione superiore al costo (il costo è al più)

si usa Ω come limitazione inferiore al costo (il costo è almeno)

$$G(n) \in O(f(n)) \iff f(n) \in \Omega(g(n))$$

InsertSort:

$$f(n) = \sum_{i=1}^n i \cdot k \quad k \geq 1 \text{ costante}$$

$$k \in \mathbb{N}$$

$$f(n) = \sum_{i=1}^n i^3 = 1^3 + 2^3 + \dots + n^3 = \Theta(n^4)$$

Stime di somme

$$\sum_{i=1}^n \log i = \Theta(n \log n)$$

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

$$\sum_{i=0}^n g^i = g^0 + \dots + g^n \implies \text{converge}(\frac{1}{1-g}) \wedge \text{diverge} + \infty$$

Iterazione:

$$T(n) = T(n-1) + b = (T(n-2) + b) + b =$$

$$(T(n-3) + b) + b + b = \dots =$$

$$(T(n-i) + i \cdot b) = \dots$$

$$(T(1) + (n-1) \cdot b) = a + (n-1) \cdot b = (\Theta)(n)$$

albero di ricorsione:

| Numero istanza | Dimensione | Costo | Costo liv |
|----------------|------------|-------|-----------|
| 1 | n | b | b |
| 1 | n-1 | b | b |
| 1 | n-2 | b | b |
| ... | ... | ... | ... |
| 1 | 1 | a | a |

$$b(n-1) + a = (\Theta)(n)$$

$$T(n) \geq \sum_{i=0}^{n=1} \cdot b \cdot n^2 \cdot \left(\left(\frac{1}{3}\right)^1 + \left(\frac{2}{3}\right)^2\right)^i + 2^{\log \frac{2}{3} n} \cdot a$$

↓

$$bn^2 + n^{\log \frac{3}{2} 2 \cdot a} = (\Theta)(n^2)$$

↓

$$T(n) \in \Omega(n^2)$$

SOSTITUZIONE VARIABILE:

Non si usa con:

$T(n)$ = (una serie di t con numeri minori di 1 – delle frazioni di n) con un costo $f(n)$ (a parte le chiamate ricorsive) – in questo caso si sua l'albero delle chiamate ricorsive

Si usa con:

$$T(n) = \begin{cases} T(g(n)) + T(k(n)) + \dots + \Theta(f(n)) \\ g(n), k(n), \dots \end{cases} \quad \text{non sono lineari}$$

esempio:

$$T(n) = \begin{cases} \Theta(1) & n \leq 2 \\ 2T(\sqrt{n}) + \Theta(\log n) & n > 2 \end{cases}$$

ALBERI BINARI

L'altezza dell'albero è la lunghezza del più lungo cammino + foglia

Con n nodi l'altezza massima dell'albero è $n - 1$ (h appartiene $O(n)$) Quindi h appartiene $\Omega(\log n)$

TBST Se InOrder(T) Stampa in ordine crescente Ricerca (max min chiave prod) Insertion

Per ricerca massimo e minimo: Minimo: quando continuo a scendere a sinistra fino a quando non trovo più nodi

```
BTS-Min(T){
    x<-T.root
    while (x != NIL && x.left != NIL){
        x<-x.left
    }
    return x
} // nel caso peggiore ha complessità (teta)(h), h = altezza albero
// nel caso l'albero sia completamente bilanciato: (teta)(n), n= numero di
nodi
// in termini di spazio: è In place
```

Ricerca di una chiave k:

```
BST-Search(x,k){    // % cerca K nel sottoalbero radicato in x
    If(x= NIL || x.key = k)    // mettere x=NIL prima così in caso vero non
implode
        Return x
    }else{
        If(x.key > k){
            Return BST.Search(x.left,k)
        }else{
            Return BST-Search(x.right,k)
        }
    } // nel caso peggiore: come prima (teta) (h || n), NON è in Place
perché ric.
```

Ricerca del Sucessore Dato T BST e dato x (app) T Il successore y di x è il nodo di T che contiene la più piccola chiave più grande della chiave di x (se esiste)

Se il sottoalbero dx di x è vuoto il più giovane antenato di uni x è parte del sottoalbero di x

```

BST-Successor(X){
  If(x.right != NIL){
    Return BST.minimum(x.right)
  }else{
    Y<-x parent
    While(y != NIL && x != y.left){
      X<-y
      Y<- x.parent
    }
  }return y
}

```

INSERIMENTO

T BST è un nuovo nodo con chiave k che non sta in T

```

BST-Insert(T, z) {      // % z.key non appartiene a T
  y <- NIL
  x <- T.root

  // Scende nell'albero per trovare la posizione corretta
  // x è il puntatore di scansione, y è il puntatore al padre (trailing
  pointer)
  while (x != NIL) {
    y <- x
    if (z.key < x.key) {
      x <- x.left
    } else {
      x <- x.right
    }
  }

  // Inserimento del nodo z
  z.parent <- y

  if (y == NIL) {
    T.root <- z      // L'albero era vuoto
  } else {
    if (z.key < y.key) {
      y.left <- z
    } else {
      y.right <- z
    }
  }
}

```

Rotazioni di alberi bilanciati (cercare di abbassare Altezza di albero):

Prep: se T è un BTS al termine dell'operazione di RightRotate(T,x) ho ancora un BST

T viene passato per gestire il caso in cui x sia la radice. Costo (teta)(1) modifico una quantità costante di posti vicino ad x

```

RighthRotate(T,x){
  P<- x.parent
  Y<- x.left
  If(p=NULL){
    y.parent<-NULL
    T.root<-y
  }else{
    y.parent<-p
    if(x=p.left){
      p.left<-y
    }else{
      p.right<-y
    }
  }
  Z<-y.right
  x.left<-z
  if(z != NIL){
    z.parent<-x
  }
  y.right<-x
  x.parent<-y
}

```

Serve una strategia per realizzare le rotazioni per mantenere: BTS bilanciati Per gli alberi rossoneri:

- le figlie sono NIL e black
- ogni nodo Red ha 2 figli black
- per ogni nodo x lungo ogni cammino x-foglie trovo la stessa quantità di nodi black

Vari casi:

Caso fortunato:

- X Red figlio di Red ha zio Black
- X e suo zio sono **opposti**

Caso quasi fortunato:

- X Red figlio di Red ha zio black
- X e suo zio NON sono **opposti**

Caso sfortunato:

- X Red figlio di Red ha zio Red

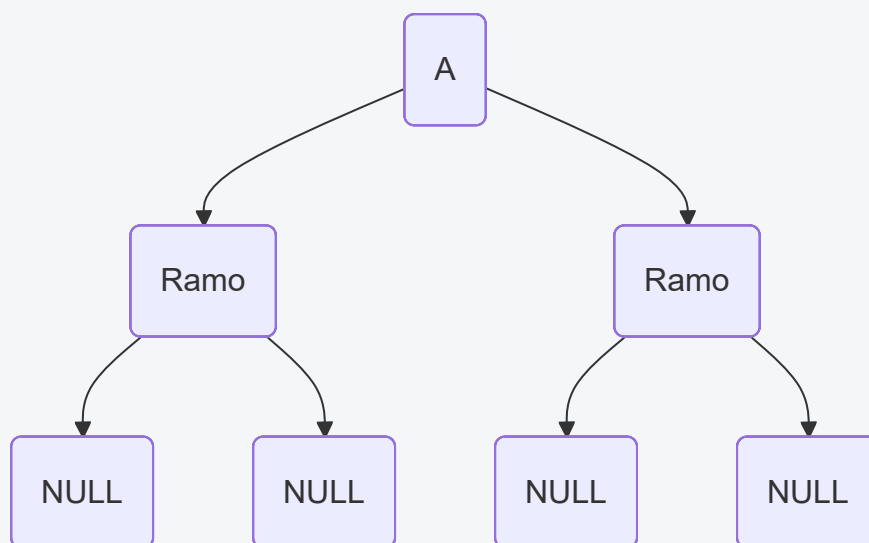
Esempio:**copia le cose dal telefono**

Complessità Dato T RBT contenente n chiavi inserire una nuova chiave che costo?

- secondo $\Theta(\log_n)$
- risalgo con 1, 2, 3 per risalire ($\Theta(1) \cdot O \log_n$)

Totale: $\Theta(\log_n)$

In ogni caso applico al di più 2 rotazioni

**Cancellazione:****Standard utilizzati:**

Red (rosso)

Black (Blue)

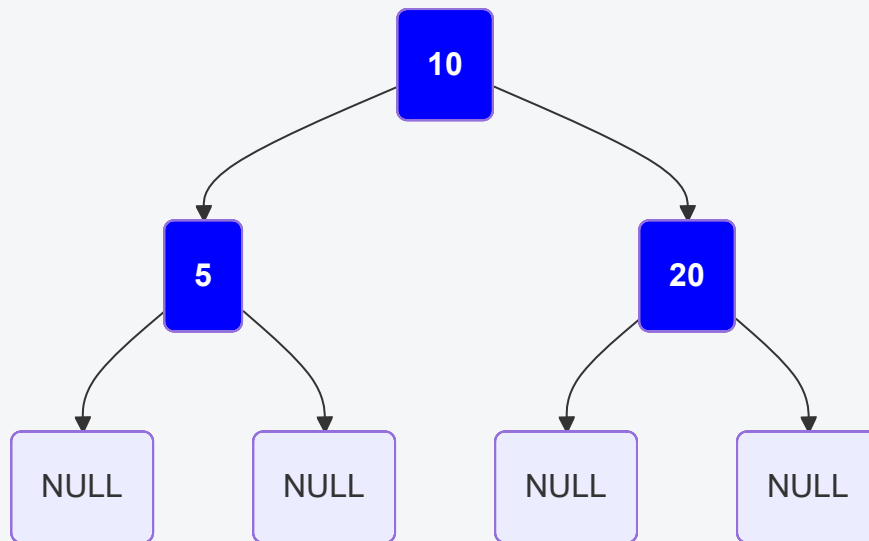
Non Importa

DB (dubble black(Nero))

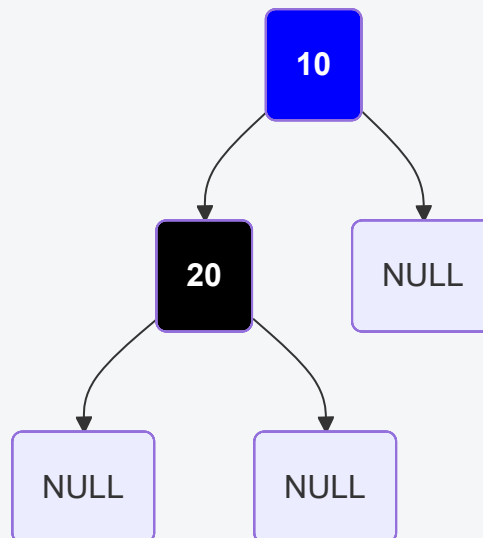
Mi concentro nel nodo che sparisce dall'albero:

Se al suo posto metto o NIL o if il suo unico figlio non NIL

Es:



per cancellare 5:



Dobbiamo tenere conto che 20 ora vale come 2 black

Se ad un certo punto della procedura il DB è RED lo ricoloro BLACK e ho finito

1: Caso fortunato: Le DB (dubble black) ha un nipote Red opposto

LeftRotate(T,p) → Ricalcolo n forse p e y

2: Caso quasi fortunato: Le DB ha un nipote Red **non** opposto

RigthRotate(T,y) → Caso fortunato (copia appunti)

3: Caso sfortunato:

