

Algoritmi e Strutture Dati

Leonardo Baldo

Contents

1	Introduzione	4
1.1	Induzione	4
1.2	Ricorrenza	4
2	Insertion Sort	5
2.1	Pseudocodice	5
2.2	Correttezza	5
3	Merge Sort	6
3.1	Pseudocodice	7
3.2	Complessità	7
4	Master Theorem	8
5	Heap Sort	9
5.1	Alberi	9
5.2	Alberi completi	9
5.3	Alberi quasi completi	9
5.4	Heap	9
5.5	Pseudocodice	10
5.6	Complessità	10
6	Quick Sort	11
6.1	Pseudocodice	11
6.2	Passaggi	11
7	Counting Sort	12
7.1	Funzionamento	12
7.2	Pseudocodice	12
7.3	Complessità	12
8	Radix Sort	13
8.1	Pseudocodice	13
8.2	Correttezza	13
8.3	Complessità	13
9	Hash Table	14
9.1	Esempio	15
9.2	Chaining	15
9.3	Caso medio	16
9.4	Funzioni di Hash	17
9.5	Open Addressing	18
9.6	Hashing Uniforme	19
10	Alberi Binari di Ricerca	20
10.1	Visita simmetrica (InOrder)	20
10.2	Ricerca (Search)	21
10.3	Ricerca di min e max	21
10.4	Successore	21
10.5	Inserimento	22
10.6	Cancellazione	23
11	RedBlack Tree	25
11.1	Rotation	26
11.2	Inserimento	26
11.3	Cancellazione	28

12 Divide et Impera	30
12.1 Memoizzazione	30
12.2 Ottimizzazione	31

1 Introduzione

Algoritmo: procedura che descrive tramite passi elementari come risolvere un problema (tramite un modello di computazione).

Uno stesso problema può essere risolto da diversi algoritmi. Di ogni algoritmo siamo interessati a conoscere:

- correttezza
- stabilità
- complessità

1.1 Induzione

L'induzione si struttura con:

- Un caso base $P(0)$.
- Un'ipotesi induttiva (se vale per $P(n)$ allora vale anche per $P(n + 1)$).

Matematicamente parlando significa che:

- Normalmente ho una formula, per esempio $n = 1$.
- Se vale per $n = 1$, provo con $n = 2$.
- Se funziona anche con $n = 2$, vuol dire che per $P(n)$ varrà anche $P(n + 1)$ e tutti i successivi.

Si ha anche un'ulteriore variante, l'induzione forte:

- U contiene 1 oppure 0.
- Se U contiene tutti i numeri minori di n allora contiene anche n .

La parola "forte" è legata al fatto che questa formulazione richiede delle ipotesi apparentemente più forti e stringenti per inferire che l'insieme U coincida con N per ammettere un numero nell'insieme è richiesto infatti che tutti i precedenti ne facciano parte (e non solo il numero precedente).

1.2 Ricorrenza

Relazione di ricorrenza è una formula ricorsiva che esprime il termine n -esimo di una successione in relazione ai precedenti. La relazione si dice di ordine r se il termine n -esimo è espresso in funzione al più dei termini $(n - 1), \dots, (n - r)$.

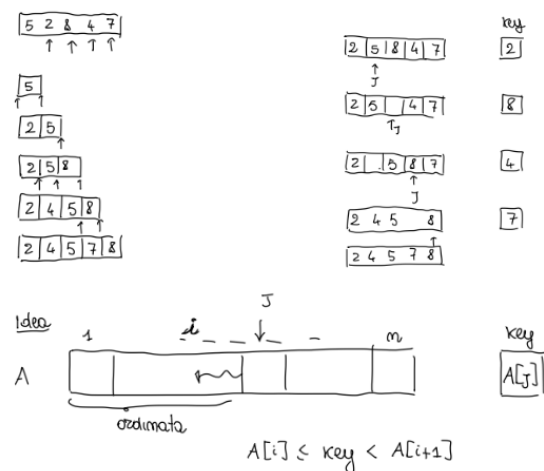
2 Insertion Sort

L'array viene virtualmente diviso in una parte ordinata e una non ordinata. I valori della parte non ordinata vengono prelevati e collocati nella posizione corretta della parte ordinata.

Caratteristiche dell'ordinamento per inserzione:

- Questo algoritmo è uno dei più semplici e di semplice implementazione.
- Fondamentalmente, l'ordinamento per inserzione è efficiente per piccoli valori di dati
- L'ordinamento per inserzione è di natura adattativa, cioè è adatto a insiemi di dati già parzialmente ordinati.

Quello che sostanzialmente fa è esaminare gli elementi a coppie e ordinarli gradualmente per come si presentano.



2.1 Pseudocodice

```

INSERTION-SORT(A)
1  n = A.length
2  for j = 2 to n
3      key = A[j]
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i - 1
8      A[i + 1] = key
    
```

2.2 Correttezza

Definiamo i seguenti passi per A da indice 1 ad indice $j - 1$

Inizializzazione: $j = 2, A[1, 1]$ ordinato

Mantenimento: inserisce $A[j]$ in $A[1 \dots j - 1]$ ordinato

Cancellazione: $j = n + 1$

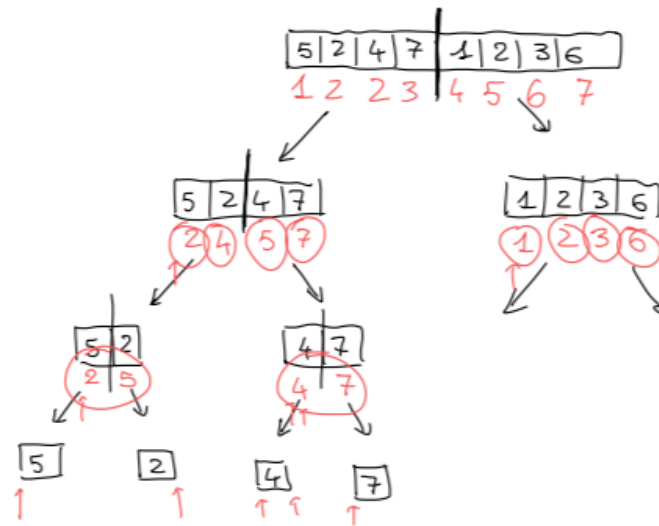
3 Merge Sort

Adotta l'approccio divide et impera, quindi:

- divide: prende il problema originale e lo divide in problemi più piccoli.
- impera: ricorsivamente, risolve i sottoproblemi; se abbastanza piccolo, si risolve subito.
- combina: le soluzioni di P_1, \dots, P_k si combinano in una soluzione di P .

Quindi, mergesort adotta i seguenti passi:

- dividere l'array in due parti
- ordina i sottoarray
- fonde i sottoarray ordinati



3.1 Pseudocodice

```
MERGE-SORT(A, p, r)
1  if p < r
2    q = (p+r)/2
3    MERGE-SORT(A, p, q)
4    MERGE-SORT(A, q+1, r)
5    MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
1  n1 = q - p + 1
2  n2 = r - q
3  for i = 1 to n1
4    L[i] = A[p+i-1]
5  for j = 1 to n2
6    R[j] = A[q+j]
7  L[n1+1] = R[n2+1] = infinity
8  i = j = 1
9  for k = p to r
10   if L[i] <= R[j]
11     A[k] = L[i]
12     i = i + 1
13   else // L[i] > R[j]
14     A[k] = R[j]
15     j = j + 1
```

3.2 Complessità

$A[p, \dots, k-1]$ contiene $L[1, \dots, i-1]$ e $R[i, \dots, j-1]$ $A[p, \dots, k-1] \leq L[1, \dots, n_1-1]$ e $R[i, \dots, n_2-1]$ è ordinato

Inizializzazione: $k = p$ e $A[p, \dots, k-1] = A[p, p-1]$

Mantenimento: Divisione in due metà progressive dell'array

Conclusione: $K = r + 1$ e $A[p, \dots, r]$ è ordinato e contiene i più piccoli elementi $L[1, \dots, n_1 + 1]$ e $R[1, \dots, n_2 + 1]$

La dimostrazione del perché sia corretto viene fatta per induzione: se vale per il primo elemento, vale per tutti i casi successivi.

4 Master Theorem

Normalmente, dato un problema con size n lo dividiamo in sottoproblemi con dimensione n/b ed $f(n)$ costo della "combina" di entrambe le cose. Otteniamo come ricorrenza (con $a \geq 1, b > 1$):

$$T(n) = aT(n/b) + f(n)$$

Esso stabilisce che, avendo a che fare con le ricorrenze di questo tipo, avremmo tre casi:

- Se il costo della risoluzione dei sotto-problemi ad ogni livello aumenta di un certo fattore, il valore di $f(n)$ diventerà polinomialmente più piccolo di $n^{\log_b a}$. Pertanto, la complessità temporale è opprressa dal costo dell'ultimo livello, vale a dire $n^{\log_b a}$.

$$f(n) = O(n^{\log_b a - \epsilon}) \quad (\epsilon > 0)$$

- Se il costo per risolvere il sotto-problema ad ogni livello è quasi uguale, allora il valore di $f(n)$ sarà $n^{\log_b a}$. Pertanto, la complessità temporale sarà $f(n)$ volte il numero totale di livelli $n^{\log_b a} \cdot \log(n)$.

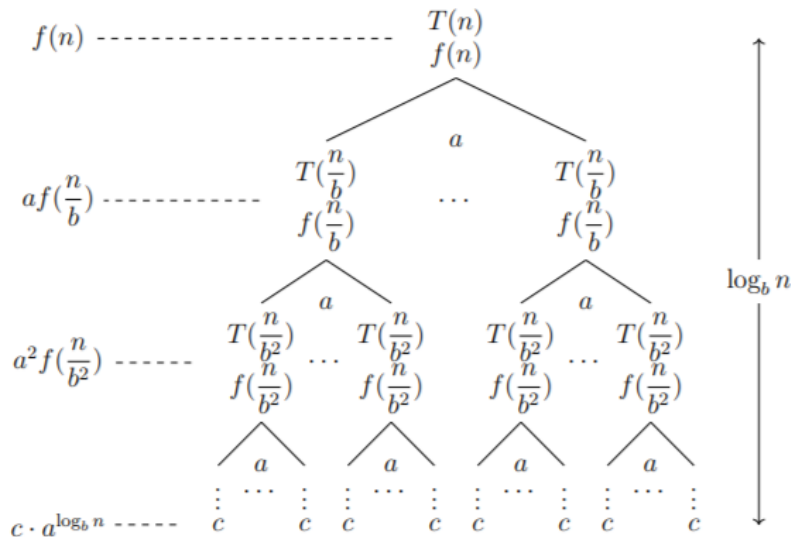
$$f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \cdot \log n)$$

- Se il costo della risoluzione dei sottoproblemi ad ogni livello diminuisce di un certo fattore, il valore di $f(n)$ diventerà polinomialmente più grande di $n^{\log_b a}$. Pertanto, la complessità temporale è opprressa dal costo di $f(n)$.

$$\left. \begin{array}{l} f(n) = \Omega(n^{\log_b a + \epsilon}) \quad (\epsilon > 0) \\ \exists 0 < k < 1 : a \cdot f(n/b) \leq k \cdot f(n) \end{array} \right\} \Rightarrow T(n) = \Theta(f(n))$$

L'idea è che la funzione si ripeta come rapporto ricorsivamente, questo corrisponde alla divisione in due parti e ancora in due parti, ecc.

$$T(n) = f(n) + a \cdot T\left(\frac{n}{b}\right) = f(n) + af\left(\frac{n}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + \dots + a^{\log_b n} f\left(\frac{n}{b^{\log_b n}}\right) = n^{\log_b a}$$



Confronta tra di loro $f(n)$ e $n^{\log_b a}$:

- se vince $f(n) \Rightarrow \Theta(f(n))$
- se pareggiano $\Rightarrow \Theta(n^{\log_b a} \cdot \log n)$
- se vince $n^{\log_b a} \Rightarrow \Theta(n^{\log_b a})$

5 Heap Sort

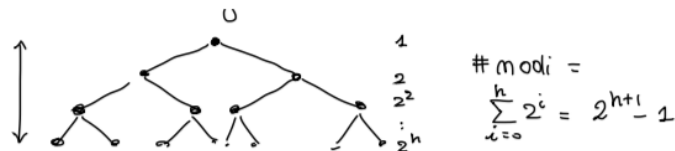
5.1 Alberi

- binario: ogni nodo ha al massimo due figli.
- altezza: distanza dalla radice alla foglia più distante.
- ordinato: ogni nodo ha un figlio minore o uguale a se stesso.

5.2 Alberi completi

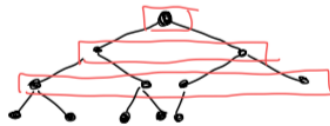
Albero binario completo: ogni nodo non foglia ha due figli e ogni cammino radice-foglia ha la stessa lunghezza.

$$\# \text{ nodi} = \sum_{i=0}^k 2^i - 1 \quad (h = \text{altezza})$$



5.3 Alberi quasi completi

Albero quasi completo: ogni livello è completo, tranne eventualmente l'ultimo con foglie tutte a sx.

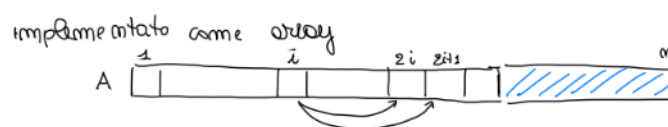


5.4 Heap

Heap: albero binario ordinato quasi completo, ma implementato come se fosse un array.

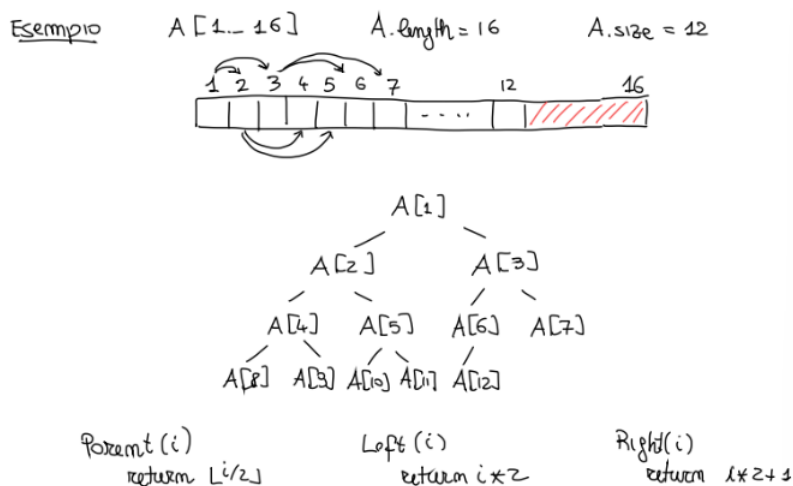
Caratteristiche:

- $A[1]$ è radice
- Un generico nodo sta in posizione $A[i]$.
- Un nodo $A[2i]$ è il figlio sinistro del nodo $A[i]$
- Un nodo definito come nodo parent/genitore sta in posizione $A[i/2]$
- Lo spazio occupato effettivamente è $A.size$, la lunghezza dell'array è $A.length$.



Max Heap Il Max Heap è un Heap dove:

- $\forall A[i], A[i] \geq$ nodi discendenti (quindi, $A.Left[i], A.Right[i]$)
- $\forall A[i], A[i] \leq$ nodi antenati (quindi, $A[i] \leq A.Parent[i]$)



5.5 Pseudocodice

```

MAX-HEAPIFY(A)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heapsize and A[l] > A[i]
4      max = l
5  else
6      max = i
7  if r ≤ A.heapsize and A[r] > A[max]
8      max = r
9  if max ≠ i
10     A[i] ↔ A[max]
11     MAX-HEAPIFY(A, max)
    
```

5.6 Complessità

Dipende dall'albero, il quale avrà n elementi possibili:

- la complessità $O(h)$, con h altezza del sottoalbero, alternativamente $O(\log(n))$.

$$\left. \begin{array}{l} n \geq 2^{h-1} + 1 = 2^h \\ h \leq \log_2 n \end{array} \right\} \Rightarrow O(h) \cong O(\log n)$$

- se invece fosse visto come ricorrenza, si può dimostrare con il Master Theorem

6 Quick Sort

L'algoritmo di ordinamento più utilizzato e generalmente più efficiente con $O(n^2)$ come caso pessimo ed $O(n \log n)$ come caso medio e migliore è proprio quicksort. Viene definito algoritmo "in-place", cioè un algoritmo che non ha bisogno di spazio extra e produce un output nella stessa memoria che contiene i dati.

Anche questo algoritmo è basato sul paradigma divide-et-impera, infatti, per ordinare $A[p, r]$:

- Divide:
 - sceglie un pivot x in $A[p, r]$
 - partiziona in $A[p, q - 1] \leq x$ e $A[q + 1, r] \geq x$
- Impera:
 - ricorre su $A[p, q - 1]$ e su $A[q + 1, r]$

6.1 Pseudocodice

```
QUICKSORT(A, p, r)
1  q = PARTITION(A, p, r)
2  QUICKSORT(A, p, q-1)
3  QUICKSORT(A, q+1, r)
```

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p - 1
3  for j = p to r - 1
4      if A[j] <= x
5          i = i + 1
6          A[i] <=> A[j]
7  A[i + 1] <=> A[r]
8  return i + 1
```

6.2 Passaggi

Per ordinare un array, seguite i passaggi seguenti:

1. Si sceglie come pivot un qualsiasi valore di indice dell'array.
2. Quindi si partiziona l'array in base al pivot.
3. Quindi si esegue un Quicksort ricorsivo della partizione di sinistra.
4. Successivamente, si esegue un Quicksort ricorsivo della partizione corretta.

Diamo un'occhiata più da vicino alla partizione di questo algoritmo:

1. Si sceglie un pivot qualsiasi, ad esempio il valore più alto dell'indice.
2. Si prenderanno due variabili per puntare a sinistra e a destra dell'elenco, escludendo il pivot.
3. La sinistra punterà all'indice più basso e la destra all'indice più alto.
4. Ora si spostano a destra tutti gli elementi maggiori del pivot.
5. Quindi si sposteranno tutti gli elementi più piccoli del perno nella partizione di sinistra.

7 Counting Sort

Il Counting Sort è una tecnica di ordinamento stabile, utilizzata per ordinare gli oggetti in base alle chiavi che sono piccoli numeri. Conta il numero di chiavi i cui valori sono uguali. Questa tecnica di ordinamento è efficiente quando la differenza tra le diverse chiavi non è così grande, altrimenti può aumentare la complessità dello spazio.

7.1 Funzionamento

1. Scopri l'elemento *max* dall'array specificato.

<i>max</i>	8	4	2	2	8	3	3	1
------------	---	---	---	---	---	---	---	---

2. Inizializzare una matrice di lunghezza *max* + 1 con tutti gli elementi 0. Questa matrice viene utilizzata per memorizzare il conteggio degli elementi nella matrice.

0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8

3. Memorizza il conteggio di ciascun elemento nel rispettivo indice nell'array count.

0	1	2	2	1	0	0	0	1
0	1	2	3	4	5	6	7	8

4. Memorizzare la somma cumulativa degli elementi dell'array count. Aiuta a posizionare gli elementi nell'indice corretto dell'array ordinato.

0	1	3	5	6	6	6	6	7
0	1	2	3	4	5	6	7	8

5. Individuare l'indice di ogni elemento della matrice originale nell'array count. Questo dà il conteggio cumulativo.

1	2	2	3	3	4	8
0	1	2	3	4	5	6

7.2 Pseudocodice

```

COUNTING-SORT(A, B, k)
1  C[0..k] = 0
2  for j = 1 to A.length
3      C[A[j]] = C[A[j]] + 1
4  for i = 1 to k
5      C[i] = C[i-1] + C[i]
6  for j = A.length to 1
7      B[C[A[j]]] = A[j]
8      C[A[j]] = C[A[j]] - 1

```

7.3 Complessità

$C[0..k] = 0$ $\Theta(k)$
 for j = 1 to A.length $\Theta(n)$
 ...
 for i = 1 to k $\Theta(k)$
 ...
 for j = A.length to 1 $\Theta(n)$
 ...

Somma $\Theta(n + k)$ con $k = \Theta(1) \Rightarrow \Theta(n)$

8 Radix Sort

Il Radix Sort è un algoritmo di ordinamento che ordina gli elementi raggruppando prima le singole cifre tutte nella stessa attuale posizione. Quindi, ordina gli elementi in base al loro ordine crescente/decescente. L'idea è quella di ordinare cifra per cifra con un algoritmo stabile e risolvere i problemi di memoria di Counting Sort.

8.1 Pseudocodice

```
Radix-SORT(A, d)
1  for j = 1 to d
2      COUNTING-SORT(A, j)
```

8.2 Correttezza

Invariante: A^{j-1} ordinato

Inizializzazione: $j = 1, A^{j-1} = A^0$

Mantenimento: A^{j-1} ordinato e rodino rispetto a j

8.3 Complessità

- d volte Counting-Sort $\Theta(n + b) \Rightarrow \Theta(d(n + b)) = \Theta(n)$
- d cifre $\Theta(1)$
- b base $\Theta(n)$

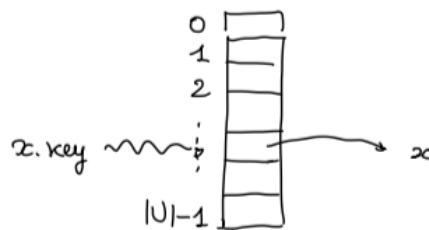
$$\left. \begin{array}{l} m \text{ bit} \Rightarrow m = O(\log n) \\ r \text{ bit per cifra} \Rightarrow r = O(\log_2 n) \\ \text{base } 2^r \end{array} \right\} \Rightarrow \Theta\left(\frac{m}{r}(m + 2^r)\right) = \Theta\left(\frac{m}{\log n}(m + 2^{\log n})\right) = \Theta\left(\frac{m}{\log n}n\right) = \Theta(n)$$

9 Hash Table

La Hash Table è una struttura di dati che memorizza i dati in modo associativo. In una tabella, i dati sono memorizzati in un formato array, dove ogni valore di dati ha un proprio valore di indice univoco. L'accesso ai dati diventa molto veloce se si conosce l'indice del dato desiderato.

In questo modo, diventa una struttura di dati in cui le operazioni di inserimento e ricerca sono molto veloci, indipendentemente dalla dimensione dei dati. La Hash Table utilizza un array come supporto di memorizzazione e usa la tecnica dell'Hash per generare un indice in cui un elemento deve essere inserito o da cui deve essere individuato.

In particolare, si considera U come insieme (universo) della chiavi, visto come $I = 0, 1, \dots, |U| - 1$. La Hash Table T viene vista come array $T[0, \dots, |U| - 1]$, in cui un elemento x è inserito in $T[x.key]$.



Problemi:

- Non è possibile avere oggetti con la stessa chiave
- OK se il numero $|U|$ è piccolo

```
INSERT(T, x)
1   T[x.key] = x            $\Theta(1)$ 
```

```
DELETE(T, x)
1   T[x.key] = NULL        $\Theta(1)$ 
```

```
SEARCH(k)
1   return T[k]            $\Theta(1)$ 
```

9.1 Esempio

Problema: consideriamo che la key sia di 8 caratteri (8 bit per rappresentare un carattere). Risulta molto costosa in termini di memoria la tabella hash.

Obiettivo: usare quantità di memoria proporzionale al numero di elementi da memorizzare.

Idea: creazione di una tabella T di dimensione $m \ll |U|$

$$h : U \rightarrow \{0, 1, \dots, m-1\}$$

Se $n > m \Rightarrow \exists x_1, x_2 : h(x_1.key) = h(x_2.key) \Rightarrow$ conflitto

Dove:

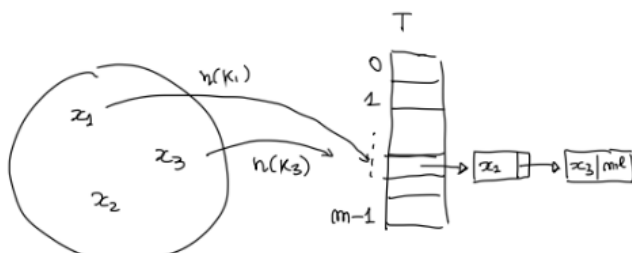
- $n = \#$ elementi memorizzati nella tabella T
- $m = \#$ celle

La collisione verifica con $n =$ numero di elementi memorizzati se $m \ll |U|$ se $n > m$.

Il principio è il pigeonhole principle, quindi nella tabella hash, ogni valore trova una corrispondenza. Il pigeonhole principle è una delle idee più semplici ma più utili della matematica e può aiutarci in questo caso. Una versione di base dice che se $(N + 1)$ piccioni occupano N buche, allora in qualche buca devono esserci almeno 2 piccioni. Quindi, se 5 piccioni occupano 4 buche, deve esserci una buca con almeno 2 piccioni.

9.2 Chaining

Il Chaining propone come soluzione quella di mettere sulla tabella liste dinamiche di elementi, invece che singoli elementi, in modo che in caso si incorra in una cella già occupata dopo un hashing, l'elemento venga inserito in coda (o in testa) alla lista. Nell'approccio di concatenamento, la tabella hash è una matrice di elenchi collegati, cioè ogni indice ha un proprio elenco collegato. Tutte le coppie chiave-valore mappate allo stesso indice saranno memorizzate nell'elenco collegato di quell'indice.



$T[0, \dots, m-1]$ contiene liste

$T[j] =$ lista degli elementi x t.c. $h(x.key) = j$

```

INSERT(T, x)
1   T[x.key] = x            $\Theta(1)$ 

```

```

DELETE(T, x)
1   T[h(x.key)] = NULL      $\Theta(1)$ 

```

```

SEARCH(k)
1   return T[h(k)]          $\Theta(n)$ 

```

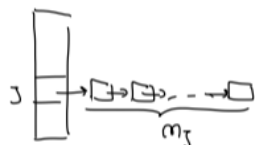
L'operazione peggiore è quella di ricerca, dove alla peggio la complessità è lineare.

9.3 Caso medio

Partenza:

- $m = \#$ celle tabella
- $n = \#$ elementi inseriti
- $\alpha = \frac{n}{m}$ che può essere minore, maggiore oppure uguale ad 1

Ogni elemento x in input ha la stessa probabilità di $\frac{1}{m}$ di essere "indirizzato" in una qualunque delle m celle. Esso è indicato come $n_j =$ lunghezza lista in $T[j]$.



$$E[n_j] = \sum_{i=1}^m \frac{1}{m} \cdot 1 = \frac{m}{m} = \alpha$$

Costo medio di Search: dato dalla ricerca di una chiave non presente k

calcolo $h(k) = j$
 accedo alla lista $T[j]$ $\left. \vphantom{\begin{array}{l} \text{calcolo } h(k) = j \\ \text{accedo alla lista } T[j] \end{array}} \right\} \Rightarrow O(1)$
 scorro la lista $T[j]$ fino in fondo (n_j elementi) $\Rightarrow n_j$ $\left. \vphantom{\begin{array}{l} \text{accedo alla lista } T[j] \\ \text{scorro la lista } T[j] \text{ fino in fondo} \end{array}} \right\} \Rightarrow \Theta(1 + \alpha)$

Ricerca di una chiave: presente

calcolo $h(k) = j$
 accedo alla lista $T[j]$ $\left. \vphantom{\begin{array}{l} \text{calcolo } h(k) = j \\ \text{accedo alla lista } T[j] \end{array}} \right\} \Rightarrow O(1)$
 scorro la lista $T[j]$ fino in a trovare l'elemento con chiave k $\left. \vphantom{\begin{array}{l} \text{accedo alla lista } T[j] \\ \text{scorro la lista } T[j] \text{ fino in a trovare l'elemento} \end{array}} \right\} \Rightarrow 1 + \frac{n_j}{2} \Rightarrow 1 + \frac{\alpha}{2}$

Più precisamente: cerco x con chiave $x.key = k$.

9.4 Funzioni di Hash

Se le chiavi fossero numero reali $k \in [0, 1) \Rightarrow h(k) = \lfloor mx \rfloor$

L'ipotesi di Hash uniforme semplice dipende dalle probabilità con cui vengono estratti gli elementi da inserire; probabilità che in genere non sono note. Le funzioni hash che descriveremo assumono che le chiavi siano degli interi non negativi.

Metodo della divisione:

$$\left. \begin{array}{l} U = \{0, 1, \dots, |U| - 1\} \\ h(k) \in \{0, 1, \dots, m - 1\} \end{array} \right\} \Rightarrow h(k) = k \bmod m$$

La scelta di m è critica:

- $m = 2^p$
- $m = 2^p - 1$

$h(k) = h(k')$ se k si ottiene "permutando" k' .

Metodo della moltiplicazione:

- se $x \in [0, 1) \Rightarrow h(x) = \lfloor (mx) \rfloor$
- chiave $k \in 0, 1, \dots, |U| - 1$
- dato k
 - fisso costante $A \in (0, 1)$ ($0 < A < 1$)
 - $k * A = k * A \bmod 1$
- $h(k) = \lfloor (m(k * A \bmod 1)) \rfloor$

pro:

- m non critico
- A non critico

contro:

- $m = 2^p$
- $w = \#$ bit particolare
- $A = \frac{q}{2^w}$ ($0 < q < 2$)
- $m(k * A \bmod 1) = m(k \frac{q}{2^w} \bmod 1)$

9.5 Open Addressing

Idea: memorizzo gli elementi dell'insieme dinamico solo nello spazio della tabella.

- funzione di Hash $h(k, i)$
- k chiave
- $i \neq$ tentativo

La funzione Hash specifica l'ordine degli slot da sondare (provare) per una chiave (per inserire/ricercare/cancellare).

- provo con $h(k, 0)$
- se capito in una cella occupata, provo con $h(k, 1)$
- poi con $h(k, 2)$ e così via, fino a che non trovo una cella libera
- per esplorare tutta la tabella: $h(k, 0), h(k, 1), \dots, h(k, m - 1) \quad \forall k \in U$

Operazioni possibili:

```
INSERT(T, x)
1  i = 0
2  repeat
3      j = h(x.key, i)
4      if (T[j] = NULL) or (T[j] = DEL)
5          T[j] = x
6          return j
7      i = i + 1
8  until i = m
9  error
```

```
SEARCH(T, k)
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j].key = k
5          return j
6      i = i + 1
7  until (i = m) or (T[j] = NULL)
8  return NOT FOUND
```

```
DELETE(T, j)
1  T[j] = DEL
```

L'Open Addressing risulta una soluzione inefficiente in caso avvengano molte cancellazioni. Similmente, giusto per citarla, è possibile anche attuare una randomizzazione dell'input, per avere una distribuzione più uniforme delle chiavi nelle liste e non dipendente dall'input. La complessità sarà sempre $\Theta(1 + \alpha)$

9.6 Hashing Uniforme

Ogni elemento determina con la stessa probabilità una qualunque delle $m!$ sequenze di ispezione.

In questo contesto le funzioni di Hash sono:

- **Ispezione Lineare:** fisso $h(k)$ funzione di hash e $h'(k+1) = (h(k) + i) \bmod m$
 - vantaggi: semplice; poche permutazioni (m dipende solo da $h'(k)$)
 - svantaggi: addensamento primario (addensamenti di celle occupate)
- **Ispezione Quadratica:**
 - dato $h(k)$ funzione di hash $h'(k, i) = (h(k) + c_1 i + c_2 i^2) \bmod m$ con c_1, c_2 opportune e $c_2 > 0$
 - addensamento secondario (addensamento quadratico di celle occupate)
 - $\#$ sequenze di ispezione dipende solo da $h(k) \bmod m \Rightarrow m$ possibilità
- **Doppio Hashing:** siano $h_1(k), h_2(k)$ funzioni di Hash di un solo argomento
 $\Rightarrow h(k, i) = (h_1(k) + i h_2(k)) \bmod m$

Costo

Il costo della funzione Search con hashing uniforme è riassunta come: $0 \leq \alpha = \frac{n}{m} \leq 1$

- Ricerca di una chiave non presente:
 - $\frac{1}{1-\alpha}$ se $\alpha < 1$
 - m se $\alpha = 1$
- Ricerca di una chiave presente:
 - $\frac{1}{\alpha} \log\left(\frac{1}{1-\alpha}\right)$ $\alpha < 1$
 - $1 + \log m$ $\alpha = 1$

10 Alberi Binari di Ricerca

Albero binario: struttura di dati ad albero composta da nodi, ognuno dei quali ha al massimo due figli, chiamati nodo destro e nodo sinistro. L'albero inizia con un singolo nodo, noto come radice.

Matematicamente:

- nodi x ed elemento/valore/chiave $x.key$
- operazioni hanno costo $O(h)$ quando non bilanciato, $O(\log(n))$ se bilanciato

Definizione induttiva:

- \emptyset è un albero
- se r è un nodo, T_1 e T_2 alberi $\Rightarrow r(T_1, T_2)$ è un albero
- ogni nodo x ha i seguenti campi:
 - $x.p$
 - $x.key$
 - $x.left$
 - $x.right$

Operazioni possibili:

- Visita simmetrica (InOrder)
- Ricerca (Search)
- Ricerca di min e max
- Successore
- Inserimento
- Cancellazione

10.1 Visita simmetrica (InOrder)

Elencare gli elementi del sottoalbero radicato in un nodo x in ordine di chiave crescente.

```
IN-ORDER(x)
1  if x != NULL
2      IN-ORDER(x.left)
3      print(x)
4      IN-ORDER(x.right)
```

La complessità di tale operazione è lineare, dato da una visita di tutto l'albero.

$$T(n) = \begin{cases} c & (n = 0) \\ T(k) + T(n - k - 1) + d & (n > 0, k < n) \end{cases}$$

Stima di complessità: $T(n) = (c + d)n + c$

10.2 Ricerca (Search)

Data k chiave, cerca nel sottoalbero radicato nel nodo x un nodo con chiave k .

```
SEARCH(x, k)
1  if (x == NULL) or (x.key == k)
2      return x
3  else if (k < x.key)
4      return SEARCH(x.left, k)
5  else
6      return SEARCH(x.right, k)
```

La complessità, nel caso peggiore, è la ricerca che continua fino ad una foglia ed il cammino radice-foglia è quello massimo. Complessità: $O(h)$.

```
SEARCH(x, k)
1  while (x != NULL) and (x.key != k)
2      if (k < x.key)
3          x = x.left
4      else
5          x = x.right
6  return x
```

Se dobbiamo esaminare tutto l'albero, nel caso peggiore avremo una complessità $\Theta(n)$ sulla base di una relazione di ricorrenza $T(n) = c + T(k) + T(n - k - 1)$.

10.3 Ricerca di min e max

Ricerca continuando ad andare verso sx oppure continuando ad andare verso dx. Per entrambi, la complessità è data dall'altezza dell'albero, in generale $O(h)$.

```
MIN(T)
1  x = T.root
2  if x == NULL
3      return NULL
4  else
5      while x.left != NULL
6          x = x.left
7      return x
```

```
MAX(T)
1  x = T.root
2  if x == NULL
3      return NULL
4  else
5      while x.right != NULL
6          x = x.right
7      return x
```

10.4 Successore

Si intende il nodo elencato dopo un nodo x passato come parametro in una visita simmetrica.

Dato $x \in ABR$:

- minimo tra i nodi più grandi di x
- nodo che segue x in una visita InOrder
- se x ha sottoalbero dx non vuoto \Rightarrow il successore è $\min(x.right)$
- se x non ha sottoalbero dx \Rightarrow il successore è il più vicino antenato di x è nel sottoalbero sx

```

SUCCESSOR(x)
1  if x.right != NULL
2      return MIN(x.right)
3  else
4      y = x.p
5      while (y != NULL) and (x == y.right)
6          x = y
7          y = y.p
8      return y

```

Complessità: $O(h)$.

10.5 Inserimento

La funzione Insert viene utilizzata per aggiungere un nuovo elemento in un albero di ricerca binario in una posizione appropriata.

La funzione Insert deve essere progettata in modo tale da non violare la proprietà dell'albero di ricerca binario a ogni valore.

1. Allocare la memoria per l'albero.
2. Impostare la parte dei dati sul valore e impostare il puntatore sx e dx dell'albero su NULL.
3. Se l'elemento da inserire sarà il primo elemento dell'albero, i puntatori sx e dx di questo nodo punteranno a NULL.
4. In caso contrario, controlla se l'elemento è inferiore all'elemento radice dell'albero.
5. Se è vero, esegue ricorsivamente l'operazione con la sx della radice.
6. Se è falso, allora si esegue questa operazione ricorsivamente con il sottoalbero dx della radice.

```

INSERT(T, z)
1  x = T.root
2  y = NULL
3  while x != NULL
4      y = x
5      if z.key < x.key
6          x = x.left
7      else
8          x = x.right
9  z.p = y
10 if y == NULL
11     T.root = z
12 else
13     if z.key < y.key
14         y.left = z
15     else
16         y.right = z

```

Complessità: $O(h)$.

10.6 Cancellazione

Distinguiamo due casi:

- z ha al più un figlio
- z ha due figli \Rightarrow pericoloso: rischio che si copino i dati di y in z .

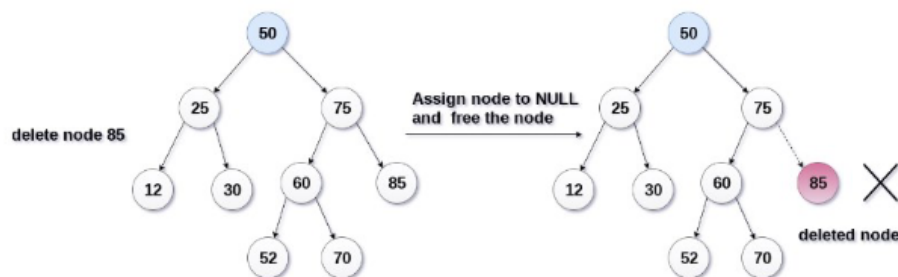
Creiamo una funzione `Transplant`, che cerca di sostituire il nodo u e il suo corrispondente sottoalbero con un altro nodo v e il suo sottoalbero. Suppongo che p stia per il genitore di un nodo.

```
TRANSPLANT( $T, u, v$ )
1  if  $u.p == \text{NULL}$ 
2     $T.\text{root} = v$ 
3  else
4    if  $u == u.p.\text{left}$ 
5       $u.p.\text{left} = v$ 
6    else
7       $u.p.\text{right} = v$ 
8  if  $v \neq \text{NULL}$ 
9     $v.p = u.p$ 
```

Esistono tre situazioni di eliminazione di un nodo dall'albero di ricerca binario:

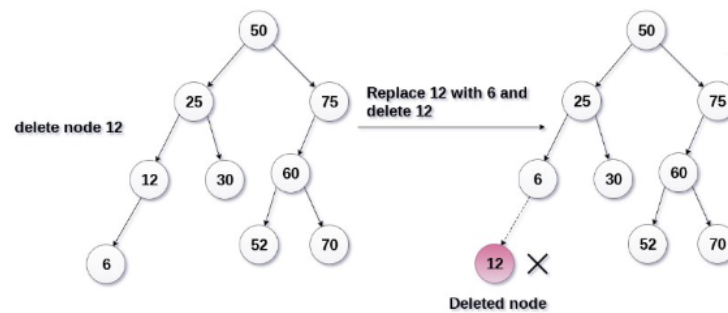
- Il nodo da eliminare è un nodo foglia: è il caso più semplice, in questo caso si sostituisce il nodo foglia con `NULL` e si libera semplicemente lo spazio allocato.

Nell'immagine seguente, stiamo eliminando il nodo 85, poiché si tratta di un nodo foglia; quindi, il nodo verrà sostituito con `NULL` e lo spazio allocato verrà liberato.



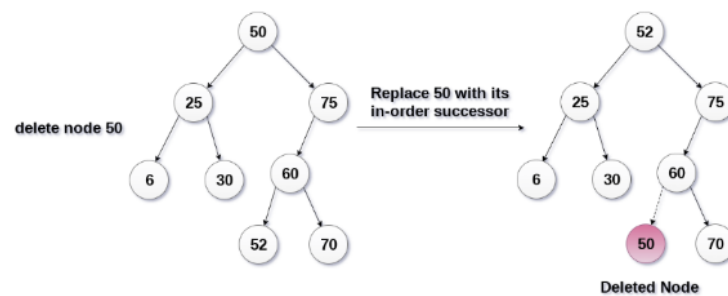
- Il nodo da eliminare ha un solo figlio: si sostituisce il nodo con il suo nodofiglio e si elimina il nodo figlio, che ora contiene il valore da eliminare. È sufficiente sostituirlo con `NULL` e liberare lo spazio allocato.

Nell'immagine seguente, il nodo 12 deve essere eliminato. Ha solo un figlio. Il nodo verrà sostituito con il suo nodo figlio e il nodo 12 sostituito (che ora è un nodo foglia) verrà semplicemente eliminato.



- Il nodo che deve essere cancellato ha due nodi figli: il nodo da eliminare viene sostituito con il suo successore o predecessore in ordine ricorsivo, finché il valore del nodo (da eliminare) non viene collocato sulla foglia dell'albero. Al termine della procedura, si sostituisce il nodo con NULL e si libera lo spazio allocato.

Nell'immagine seguente, il nodo da eliminare è il nodo 50, che è il nodo radice dell'albero. L'attraversamento in ordine sparso dell'albero: 6, 25, 30, 50, 52, 60, 70, 75. Sostituire 50 con il suo successore in ordine 52. Ora 50 verrà spostato alla foglia dell'albero, che verrà semplicemente eliminata.



```

DELETE(T, z)
1  if z.left == NULL
2      TRANSPLANT(T, z, z.right)
3  else if z.right == NULL
4      TRANSPLANT(T, z, z.left)
5  else
6      y = MIN(z.right)
7      if y.p != z
8          TRANSPLANT(T, y, y.right)
9          y.right = z.right
10         y.right.p = y
11         y.left = z.left
12         y.left.p = y
13     TRANSPLANT(T, z, y)

```

Complessità: $O(h)$

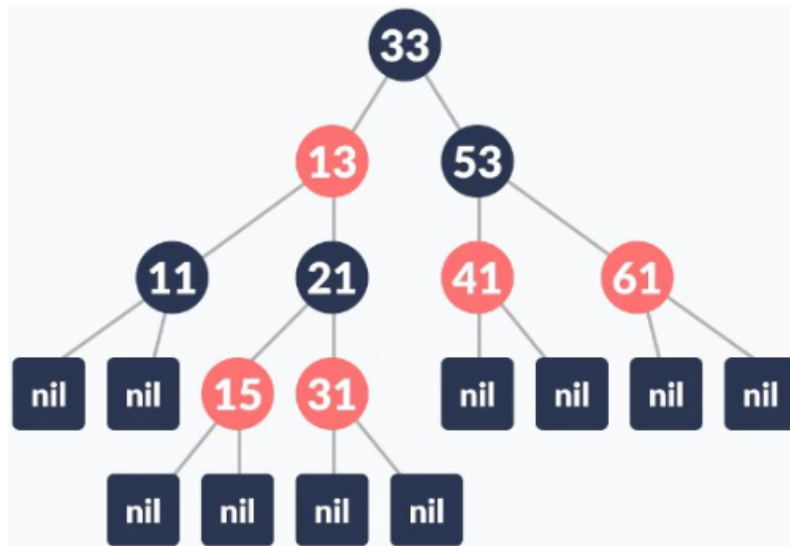
Attenzione se dobbiamo esaminare tutto l'albero, nel caso peggiore avremo una complessità $\Theta(n)$ sulla base di una relazione di ricorrenza $T(n) = c + T(k) + T(n - k - 1)$.

11 RedBlack Tree

I RB-Tree sono ABR i cui nodi hanno un bit extra riservato ad un campo colore $x.color$, che può essere: *red* per il rosso, *black* per il nero.

Un RB-Tree soddisfa le seguenti proprietà:

- Proprietà RB: ogni nodo è colorato, rosso o nero
- Proprietà della radice: la radice è nera
- Proprietà delle foglie: ogni foglia (NULL) è nera
- Proprietà del rosso: se un nodo rosso ha dei figli, questi sono sempre neri
- Proprietà Depth: per ogni nodo, qualsiasi percorso semplice da questo nodo a qualsiasi foglia discendente ha la stessa profondità nera (il numero di nodi neri)



Le operazioni degli RB-Tree saranno chiaramente simili a prima, in particolare:

- Search
- Max
- Min
- Successor
- Predecessor (se x ha 2 figli, il suo predecessore è il valore max nel suo sottoalbero sx e il suo successore il valore min nel suo sottoalbero dx. Se non ha un figlio a sx, il predecessore di un nodo è il suo primo antenato a sx)
- Insert (difficile mantenere la colorazione)
- Delete (difficile mantenere la colorazione)

La complessità delle operazioni è sempre data dall'altezza h dell'albero, come $h = O(\log(n))$ con $h \leq 2\log_2(n+1)$.

11.1 Rotation

Nell'operazione di *Rotation*, le posizioni dei nodi di un sottoalbero vengono scambiate. Esistono 2 tipi di rotazioni:

- Rotazione a sx: la disposizione dei nodi a dx viene trasformata in quella dei nodi a sx

```
Left(T, x)
1  y = x.right
2  x.right = y.left
3  x.right.p = x
4  Transplant(T, x, y)
5  y.left = x
6  x.p = y
```

- Rotazione a dx: la disposizione dei nodi a sx viene trasformata in quella del nodo a dx

```
Right(T, y)
1  x = y.left
2  y.left = x.right
3  if x.right != T.NULL
4      x.right.p = y
5  x.p = y.p
6  if y.p == T.NULL
7      T.root = x
8  else if y == y.p.right
9      y.p.right = x
10 else y.p.left = x
11     x.right = y
12     y.p = x
```

11.2 Inserimento

Vogliamo inserire z nell'albero T . L'idea è quella di porre $z.color = RED$ (migliore, in quanto andando a modificare il numero di nodi neri, cambia l'altezza nera, e la cosa è difficile da sistemare).

- Se violo la proprietà 2 (radice deve essere nera) $\Rightarrow z.color = BLACK$
- Se violo la proprietà 4 (se un nodo rosso ha dei figli, questi sono sempre neri) \Rightarrow risolvo localmente e sposto verso l'alto il problema

```
RB-INSERT(T, z)
1  INSERT(T, z)
2  z.color = RED
3  RB-INSERT-FIXUP(T, z)
```

Abbiamo dei problemi rispetto alle proprietà degli RB:

- $z.color = RED$
- $z.p.color = RED$

Analizziamo il macrocaso $z.p$ è figlio sinistro. Abbiamo due possibilità per y :

- $y.color = RED$

Inverto il colore di $z.p.p$ con quello dei figli \Rightarrow risolviamo localmente rimandiamo il problema verso l'alto

- $y.color = BLACK$

Possiamo distinguere due sottocasi:

- z figlio dx \Rightarrow applico $Left(T, z.p)$
- z figlio sx \Rightarrow scambio i colori di $z.p.p$ con $z.p \Rightarrow$ applico $Right(T, z.p.p)$

```

RB-INSERT-FIXUP(T, z)
1  while z.p.color == RED
2      if z.p == z.p.p.left
3          y = z.p.p.right
4          if y.color == RED
5              z.p.p.color = RED
6              z.p.color = BLACK
7              y.color = BLACK
8              z = z.p.p
9      else
10         if z == z.p.right
11             LEFT(T, z, p)
12             z = z.left
13             z.p.color = BLACK
14             z.p.p.color = RED
15             RIGHT(T, z.p.p)
16     else
17         ...
18     T.root.color = BLACK

```

Complessità $O(\log n) + \max 2$ rotazioni

Quale proprietà RB può essere violata quando si chiama RB-INSERT-FIXUP

- Proprietà 1: continua a valere
- Proprietà 2: violata quando z è la radice e z è rosso
- Proprietà 3: continua a valere
- Proprietà 4: violata quando sia z che $p[z]$ sono rossi
- Proprietà 5: continua a valere, perché coloriamo z di rosso

I 3 casi seguenti violeranno le proprietà dell'albero rosso-nero e i passaggi successivi mostrano come RB-INSERT-FIXUP funziona per ricorrere alle proprietà rosso-nere:

- caso 1: lo zio y di z è rosso
 1. colorare $p[z]$ e y di nero
 2. colorare $p[p[z]]$ di rosso
 3. spostate il puntatore z di due livelli su $p[p[z]]$
- caso 2: lo zio di z , y , è nero e z è un figlio dx
 1. eseguire la rotazione a sx su z
 2. passare al caso 3
- caso 3: lo zio di z , y , è nero e z è un figlio di sx
 1. colorare $p[z]$ di nero
 2. colorare $p[p[z]]$ di rosso
 3. eseguite la rotazione a dx su $p[p[z]]$

11.3 Cancellazione

Per la cancellazione $RB\text{-Delete}(T, z)$ distinguiamo 2 casi:

- z ha un figlio
- z ha due figli

Ci comportiamo allo stesso modo della $Delete(T, z)$ per un ABR, facendo però un ulteriore accorgimento:

- se $z.color = RED \Rightarrow$ non ho problemi
- se $z.color = BLACK \Rightarrow$ risolvo localmente e sposto verso l'alto il problema

Dunque, in seguito alla $Delete(T, z)$, il nodo x che ha preso il posto di z ne "assorbirà" il colore diventando doppiamente $BLACK$. Abbiamo due casi:

- x è figlio s_x (noi analizziamo solo questo, immaginiamo)
- x è figlio d_x

Abbiamo due possibilità per w :

- $w.color = RED$
 1. scambio i colori di w con $x.p$
 2. applico $Left(T, x, p)$
- $w.color = BLACK$

Possiamo distinguere 3 sottocasi:

- $w.left.color = BLACK$ e $w.right.color = BLACK$
 1. x cede un suo $BLACK$ a $x.p$ e w diventa per forza RED
 2. risolviamo localmente e rimandiamo il problema in alto
- $w.right.color = BLACK$
 1. scambio i colori di w con $w.left$
 2. applico $Right(T, w)$
- $w.right.color = RED$
 1. scambio i colori di $x.p$ con w e $w.right$ diventa $BLACK$
 2. applico $Left(T, x, p)$

Complessità $O(\log n) + \max 3$ rotazioni

```

RB-DELETE(T,z)
1  if z.left == T.NULL && z.right == T.NULL
2      then y = z
3      else y = TREE-SUCCESSOR(z)
4  if y.left != T.NULL
5      then x = y.left
6      else x = y.right
7  x.p = y.p
8  if y.p == T.NULL
9      then T.root = x
10     else if y == y.p.left
11         then y.p.left = x
12         else y.p.right = x
13  if y != z
14      then z.key = y.key
15  if y.color == BLACK
16      then RB-DELETE-FIXUP(T,x)
17  return y

```

```

RB-DELETE-FIXUP(T,x)
1  while x != T.root && x.color == BLACK
2      if x == x.p.left
3          then w = x.p.right
4              if w.color == RED
5                  then w.color = BLACK
6                     x.p.color = RED
7                     Left(T,x.p)
8              if w.left.color == BLACK && w.right.color == BLACK
9                  then w.color = RED
10                 x = x.p
11              else if w.right.color == BLACK
12                  then w.left.color = BLACK
13                     w.color = RED
14                     Right(T,w)
15                     w = x.p.right
16                     w.color = x.p.color
17                     x.p.color = BLACK
18                     w.right.color = BLACK
19                     Left(T,x.p)
20                     x = T.root
21              else (same with Left and Right inverted)
22  x.color = BLACK

```

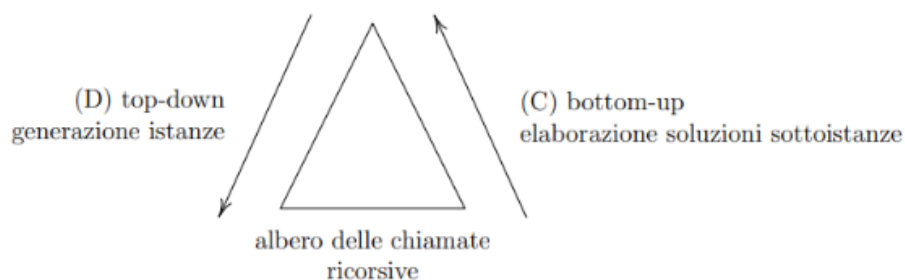
12 Divide et Impera

Il metodo Divide et Impera, non ha memoria, ogni volta che si calcola una soluzione, il processo richiede di ricalcolare la sottoistanza del problema che lo ha risolto, generando quindi un problema di sottoistanze ripetute.

Il Divide et Impera, possiede 2 fasi:

- **Top-down:** con cui vengono generate le istanze risolte divide il problema originale in un numero di sottoproblema (divide), risolvendo i problemi ricorsivamente (impera) e combinando le soluzioni dei sottoproblemi nella soluzione ai sottoproblemi originali (combine)
- **Bottom-up:** normalmente non ricorsiva ma iterativa
 - caratterizza la struttura delle soluzioni ottimali
 - definiscono ricorsivamente i valori delle soluzioni ottimali
 - calcola il valore delle soluzioni ottimali
 - costruisce una soluzione ottimale a partire dalle informazioni calcolate

Sinteticamente, si elaborano le soluzioni alle istanze presenti.



L'approccio della programmazione dinamica salta completamente l'approccio top-down, normalmente usando algoritmi iterativi per evitare di dover calcolare la soluzione. La programmazione Divide et Impera usa entrambi gli approcci e normalmente in modo ricorsivo.

12.1 Memoizzazione

È possibile mantenere i vantaggi del Top-down e quindi anche del Divide et Impera senza incorrere nel problema del ricalcolo delle soluzioni, usando il concetto di memoizzazione.

Un algoritmo memoizzato è costituito da 2 subroutine/procedure:

- **routine di inizializzazione:**
 - risolve direttamente i casi base
 - inizializza una struttura dati "tabella" che contiene le soluzioni ai casi base ed elementi per tutte le sottoistanze da calcolare inizializzate ad un valore di default
 - invoca la struttura ricorsiva
- **routine ricorsiva:**
 - segue il codice del Divide et Impera preceduto da un test sulla tabella per verificare se la soluzione è già stata calcolata e memorizzata
 - se sì, ritorna
 - altrimenti si calcola ricorsivamente e si memorizza nella struttura

12.2 Ottimizzazione

Ora, consideriamo i problemi di ottimizzazione, per cui si cerca di trovare una cosiddetta soluzione ottima, definita come s^* , definendo tutto come segue:

- I = insieme delle istanze
- S = insieme delle soluzioni
- $\forall i \in I, S(i) = \{s \in S : (i, s) \in \Pi\}$ = insieme delle soluzioni ammissibili
- $c : S \rightarrow \mathbb{R}$ = funzione di costo
- $i \in I, s^* \in S(i) : c(s^*) = \min / \max \{c(s) : s \in S(i)\}$

Paradigma generale sullo sviluppo di un algoritmo di programmazione dinamica. Caratteristiche dei problemi:

- struttura ricorsiva: la soluzione ottima si ottiene come funzione di soluzioni ottime di sottoistanze ("proprietà di sottostruttura ottima", quindi il fatto che ogni soluzione contenga già soluzioni precedenti)
- esistenza di sottoistanze ripetute ("overlapping subproblems"), teoricamente risolvibili in maniera ottima (altrimenti, conviene applicare Divide et Impera)
- spazio dei sottoproblemi piccolo: poche sottoistanze che possono contribuire a creare la soluzione al livello superiore

Ricetta:

- caratterizza la struttura di una soluzione ottima s^* in funzione di soluzioni ottime s_1^*, \dots, s_k^* di sottoistanze di taglia inferiore
- determinare una relazione di ricorrenza del tipo $c(s^*) = f(c(s_1^*), \dots, c(s_k^*))$
- calcola $C(S^*)$ utilizzando la ricorrenza ma impostando il calcolo in maniera bottom-up (iterativo) oppure memoizzando il codice Divide et Impera basato sulla definizione di $C(S^*)$
- (opzionale) mantieni informazioni strutturali aggiuntive che permettono di ricostruire S^*