Algoritmi e Laboratorio

Damiano Trovato

Primo semestre, anno 2024/2025

Indice

1 Introduzione al corso **5**
- 1.1 Introduzione al corso di Algoritmi . . . 5
  - 1.1.1 Prova d'esame e sistema di valutazione . . . 5
  - 1.1.2 Codice Teams . . . 5
  - 1.1.3 Argomenti generali del corso . . . 5

2 Introduzione agli Algoritmi di Ordinamento **7**
- 2.1 Alcuni algoritmi di Ordinamento iterativi . . . 7
- 2.2 Caratteristiche degli algoritmi di sorting . . . 8

3 La Ricorsione **9**
- 3.1 Cos'è la Ricorsione . . . 9
- 3.2 Algoritmi, problemi e soluzioni ricorsive . . . 10
  - 3.2.1 Calcolo del fattoriale . . . 10
  - 3.2.2 Moltiplicazione . . . 10
  - 3.2.3 Calcolo n-esimo numero della serie di Fibonacci . . . 11
- 3.3 Problema dello zaino (o Knapsack problem) . . . 12
  - 3.3.1 Massimizzare il numero di oggetti . . . 12
  - 3.3.2 Massimizzare il peso degli oggetti . . . 12
  - 3.3.3 Massimizzare il valore degli oggetti . . . 14

4 Heap **15**
- 4.1 Introduzione all'heap . . . 15
  - 4.1.1 Proprietà dell'heap . . . 15
  - 4.1.2 Tipi di heap . . . 16
- 4.2 Funzioni nei Min-Heap (Max-Heap) . . . 16
  - 4.2.1 Valore del minimo . . . 17
  - 4.2.2 Heapify . . . 17
  - 4.2.3 Estrazione del minimo . . . 17
  - 4.2.4 Inserimento di un nuovo nodo . . . 18
  - 4.2.5 Decrease-key . . . 18
- 4.3 Astrazione vs Implementazione Heap . . . 19
  - 4.3.1 Heap come Array . . . 19
  - 4.3.2 Funzioni left, right, parent . . . 19

4.3.3 Heapify . . . 20
4.3.4 Insert . . . 20
4.3.5 Extract-min . . . 20
4.3.6 Costruire un heap . . . 21
4.4 Heapsort . . . 21

5 Algoritmi di Ordinamento a Tempo Lineare 22
5.1 Ordinamento basato sul confronto . . . 22
5.2 Counting Sort . . . 24
5.2.1 Problemi del Counting Sort . . . 25
5.2.2 Modifica del vettore C e stabilità . . . 25
5.2.3 Counting Sort in Loco . . . 25

6 Tabelle Hash 26
6.1 Introduzione alle Hash Table . . . 26
6.1.1 Cos'è un dizionario? . . . 26
6.1.2 Possibili implementazioni e complessità . . . 26
6.1.3 Tabelle a Indirizzamento Diretto . . . 27
6.2 Hashing . . . 28
6.2.1 Collisioni . . . 28
6.3 Hashing con Concatenazione . . . 28
6.3.1 Principio di Hashing Uniforme Semplice . . . 29
6.3.2 Implementare una funzione hash . . . 30
6.3.3 Metodo della moltiplicazione con operazioni bitwise . . . 31
6.4 Hashing a indirizzamento aperto . . . 32
6.4.1 Funzione di hashing per indirizzamento aperto . . . 32
6.4.2 Cancellazione . . . 32
6.4.3 Quando preferire l'indirizzamento aperto . . . 33
6.4.4 Pseudo-codice . . . 33
6.5 Funzioni hash per indirizzamento aperto . . . 34
6.5.1 Scansione lineare . . . 34
6.5.2 Scansione quadratica . . . 35
6.5.3 Hashing doppio . . . 35

7 Alberi rosso-neri 36
7.1 Cosa sono gli alberi rosso neri? . . . 36
7.2 Perché preferire un albero rosso-nero ad un BST? . . . 36
7.2.1 Ripasso: proprietà dei BST . . . 36
7.3 Proprietà degli alberi rosso-neri . . . 37
7.3.1 Rilassamento terza proprietà* . . . 37
7.3.2 Sulla quinta proprietà . . . 37
7.3.3 Altezza degli RB-Tree . . . 37
7.4 Operazioni di base su alberi rosso-neri . . . 38
7.4.1 Rotazioni . . . 38
7.4.2 Ricolorazione . . . 39
7.4.3 Inserimento . . . 39

7.4.4 RB-Insert-Fixup . . . 39
7.4.5 Rimozione negli RB-Trees . . . 40
7.4.6 RB-Delete-Fixup . . . 41

8 Programmazione dinamica 43
8.1 Introduzione ai problemi di ottimizzazione . . . 43
8.1.1 Formalizzazione problemi di ottimizzazione . . . 43
8.2 La programmazione dinamica . . . 44
8.2.1 Quali problemi possiamo risolvere? . . . 44
8.3 Rod-Cutting Problem . . . 45
8.3.1 Introduzione al Rod-Cutting problem . . . 45
8.3.2 Rod-Cutting ricorsivo top-down . . . 46
8.3.3 Rod-Cutting ricorsivo con memorizzazione . . . 47
8.3.4 Soluzione bottom-up con tabulazione . . . 48
8.4 Problema della moltiplicazione di matrici . . . 49
8.4.1 Come si effettua una moltiplicazione tra matrici? . . . 49
8.4.2 La complessità del prodotto tra matrici . . . 49
8.4.3 Programmazione dinamica - Prodotto tra matrici . . . 52

9 Programmazione greedy 54
9.1 Introduzione alla strategia greedy . . . 54
9.1.1 Cosa si intende per scelta greedy? . . . 54
9.2 Problema dello zaino - esempio . . . 55
9.2.1 Problema dello zaino . . . 55
9.2.2 Riformulazione del problema . . . 55
9.3 Problema della compressione - Huffman . . . 57
9.3.1 Definizioni . . . 57
9.3.2 Come scegliere i codici? . . . 57
9.3.3 Algoritmo Greedy - Codifica Huffman . . . 58
9.3.4 Pseudo-Codice . . . 58
9.3.5 Codice aggiuntivo . . . 58

10 Algoritmi sui Grafi 60
10.1 Introduzione agli algoritmi sui grafi . . . 60
10.1.1 Grafi orientati e non orientati . . . 60
10.1.2 Come rappresentare un grafo . . . 60
10.2 Gli algoritmi di visita sui grafi . . . 61
10.2.1 BFS . . . 61
10.2.2 Implementazione della BFS . . . 61
10.2.3 Pseudo-codice procedura BFS . . . 62
10.2.4 Complessità dell'algoritmo BFS . . . 62
10.2.5 Esempio di BFS applicata ad un grafo . . . 63
10.2.6 Calcolo delle distanze . . . 65
10.3 Struttura topologica di un grafo . . . 66
10.3.1 Ordinamento topologico . . . 66
10.4 Ricerca in profondità - DFS . . . 67

10.4.1 Introduzione alla DFS 67
10.4.2 Il filo di Arianna? 67
10.4.3 Alberi DFS 70
10.4.4 Riconoscere i tipi di arco nel grafo 71
10.4.5 Procedura algoritmo DFS 72
10.4.6 Ordinamento topologico 73
10.4.7 Calcolo delle componenti fortemente connesse 74

11 Varie ed eventuali **76**
11.1 "Varie ed eventuali" cosa? 76
11.2 Code con priorità 77
11.3 Hash tables scansione lineare 78
11.3.1 Esercizio tipo - 1 78

Capitolo 1

Introduzione al corso

1.1 Introduzione al corso di Algoritmi

Questo primo capitolo includerà tutto ciò che è stato detto nella lezione introduttiva del professore Faro al corso di Algoritmi e Laboratorio del corso M-Z dell'anno 2024/2025.

1.1.1 Prova d'esame e sistema di valutazione

Prova scritta e orale su entrambi i moduli. L'orale può essere opzionale con un voto maggiore di 22, abbassando di 5 il voto della prova scritta. La prova scritta durerà 3 ore, e la valutazione sarà in trentesimi.

1.1.2 Codice Teams

Codice Teams comune ad ambo i corsi: *******

1.1.3 Argomenti generali del corso

Modulo di Algoritmi:

- **Strutture dati di base**
- **Algoritmi fondamentali di base**
  Con un focus importante sulla ricorsione.
- **Tecniche avanzate di programmazione**
  Relative al design dell'algoritmo, in nessun linguaggio di programmazione specifico, imparando a scrivere degli algoritmi ex novo per la risoluzione di problemi specifici.
- **La creazione di algoritmi ricorsivi**
  La ricorsione offre delle soluzioni molto eleganti e semplici a problemi spesso molto complessi.

- **Struttura dati Heap**
  Struttura dati utilizzata per l'implementazione delle code con priorità e per l'heapsort.

- **Algoritmi di Ordinamento in Tempo Lineare**
  Come il counting sort, con pro e contro di vario tipo.

- **Hash Tables e Dizionari**
  Struttura dati al cui interno vengono effettuate principalmente operazioni di ricerca.

- **Alberi Binari di Ricerca Rosso Neri**
  Struttura dati basata sugli alberi binari con ulteriori vincoli che ne definiscono l'essere bilanciato. Ogni operazione avviene in tempo logaritmico, anche nel worst case.

- **Programmazione Dinamica**
  Paradigma o tecnica di programmazione usata per risolvere problemi di vario tipo. Basata sulla ricorsione, risolve problemi di ottimizzazione.
  [Domanda frequente: cos'è un problema di ottimizzazione? Un problema di ottimizzazione è un problema che presenta più soluzioni e un criterio univoco per trovare la soluzione migliore, detta soluzione ottima.]

- **Problemi sui Grafi**
  Algoritmi per la risoluzione di problemi sui grafi.

Capitolo 2

Introduzione agli Algoritmi di Ordinamento

2.1 Alcuni algoritmi di Ordinamento iterativi

- **Selection Sort:**
  Scorro l'array cercando l'elemento minimo (o massimo) ponendolo nel primo slot dell'array (nell'ultimo slot dell'array). Continuo iterativamente. Complessità nel caso pessimo $O(n^2)$ e nel caso migliore $O(n^2)$.

- **Bubble Sort:**
  Ordinamento a bolle. Verifico se l'ordine degli elementi, a due a due, è corretto o meno. Scambio gli elementi se non sono in ordine. Procedo iterativamente n volte su tutto l'array.
  Complessità nel caso pessimo $O(n^2)$ e nel caso migliore $O(n^2)$.

- **Insertion Sort:**
  Dividendo l'array in due parti, un sottoarray sinistro e un sottoarray destro, inserisco gli elementi del sottoarray destro nel sottoarray sinistro, in relazione agli altri elementi dell'array. Caso pessimo $O(n^2)$ o $O(nk)$ e nel caso migliore $O(n)$. k è la distanza massima tra la posizione di ogni valore dell'array di input e la sua posizione nella controparte ordinata. L'Insertion Sort ha performance diverse con input differenti. Un algoritmo di ordinamento che trae vantaggio dagli elementi già in ordine dell'array (come nel caso dell'Insertion Sort), si dirà **adattivo**.

2.2 Caratteristiche degli algoritmi di sorting

- **Adattività**:
  La performance dell'algoritmo cambia dipendentemente dall'input.
  In alcuni casi è possibile calcolarne la complessità in funzione di una variabile k (vedi l'insertion sort).
- **Funzionamento In Loco/In place:**
  Non è richiesta una memoria ausiliaria (il Merge Sort richiede una memoria ausiliaria. Il Quick Sort no).
- **Stabilità:**
  Un algoritmo stabile preserva l'ordinamento relativo tra gli elementi con la stessa chiave.

Caratteristiche degli algoritmi di ordinamento noti - Iterativi

Selection sort: in loco. $O(n^2)$
Bubble sort: in loco, stabile. $O(n^2)$
Insertion Sort: adattivo, in loco, stabile. Worst $O(n^2)$, Avrg $O(nk)$, Best $O(n)$

Caratteristiche degli algoritmi di ordinamento noti - Ricorsivi

Quick sort: in loco[1]. Worst $O(n^2)$, Avrg e Best $O(n \log n)$
Merge sort: stabile. $O(n \log n)$

Implementare un Selection Sort stabile

È possibile rendere un algoritmo Selection Sort sostituendo lo swap tra gli elementi dell'array, con una funzione che shifta tutti gli elementi e porta l'elemento selezionato alla fine dell'array. L'implementazione base del Selection Sort **non è stabile**.

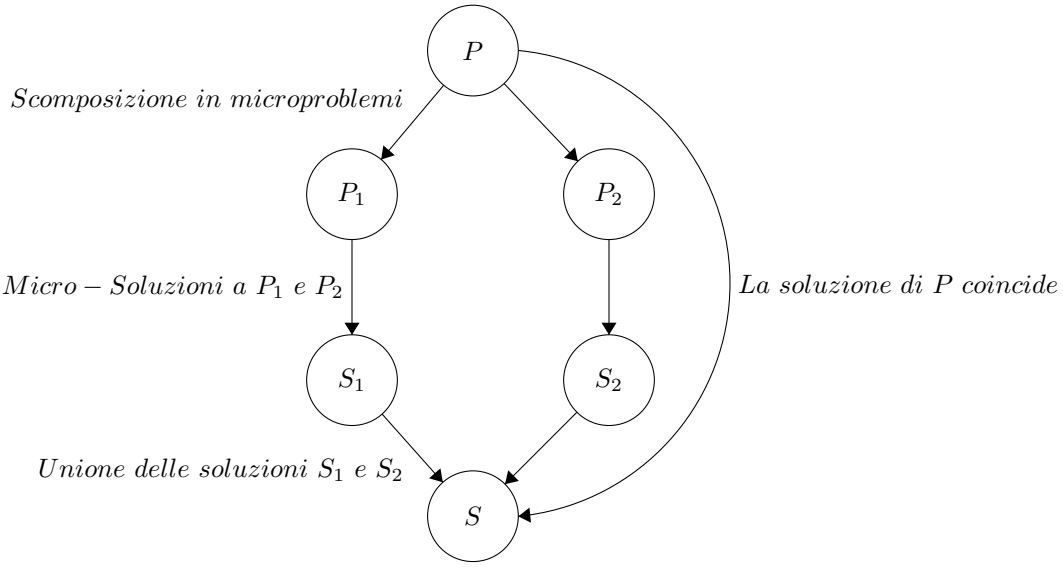
[1] Scopriremo presto che la ricorsione implica l'utilizzo di memoria ausiliaria per le chiamate ricorsive. È vero che il Quick sort opera direttamente sull'array di input, ricadendo quindi nella categoria degli in-place sorting algorithms, ma è importante sapere che la ricorsione implica sempre una richiesta di memoria aggiuntiva.

Capitolo 3

La Ricorsione

3.1 Cos'è la Ricorsione

La ricorsione è una tecnica di programmazione che permette di richiamare una funzione all'interno della definizione della funzione stessa.
Permette di approcciarsi a determinati problemi in maniera più elegante e intuitiva: problemi scomponibili in problemi più piccoli sono tra i principali candidati ad avere soluzioni ricorsive, sfruttando l'approccio *divide et impera*. Diventa necessario indivduare la struttura dei sottoproblemi e la soluzione al problema di dimensione minore, che chiameremo **caso base**. Un approccio di tipo ricorsivo lavorerà nel seguente modo:



3.2 Algoritmi, problemi e soluzioni ricorsive

Un approccio ricorsivo sarà spesso più consono ad uno iterativo: basti pensare al problema delle Torri di Hanoi o a molti dei problemi di ottimizzazione che vedremo più avanti. La loro interpretazione ricorsiva risulterà molto più intuitiva ed elegante rispetto alla controparte iterativa.
Tuttavia, la ricorsione richiede molte più risorse, se messa a confronto con l'iterazione: una richiesta di memoria che cresce ad ogni chiamata ricorsiva.
All'interno di questa sezione, affronteremo lo studio di alcuni algoritmi, problemi tipici e soluzioni di tipo ricorsivo.

3.2.1 Calcolo del fattoriale

Uno dei problemi ricorsivi più tipici è quello del calcolo del fattoriale:

$$f(n) = n! = 1 * 2 * 3 * 4 * ... * (n-1) * n$$

In forma ricorsiva, scriveremo:

$$f(n) = \begin{cases} 1 \text{ se } n = 1 \\ n * f(n-1) \end{cases}$$

oppure

$$n! = \begin{cases} 1 \text{ se } n = 1 \\ n * (n-1)! \end{cases}$$

La forma ricorsiva in pseudo-codice sarà:

```
1.  fattoriale(n)
2.      if n = 1 then return 1
3.      return fattoriale (n-1)*n
```

La forma iterativa:

```
1.  fattoriale(n)
2.      FOR p = 2 TO n DO
3.      p = p * i
4.      RETURN p
```

3.2.2 Moltiplicazione

Implementazione della moltiplicazione $x \cdot y$.

```
1.  Mul(x,y)
2.      if y = 1 then return x
3.      if y = 0 return 0
4.      return Mul(x,y-1)+x
```

3.2.3 Calcolo n-esimo numero della serie di Fibonacci

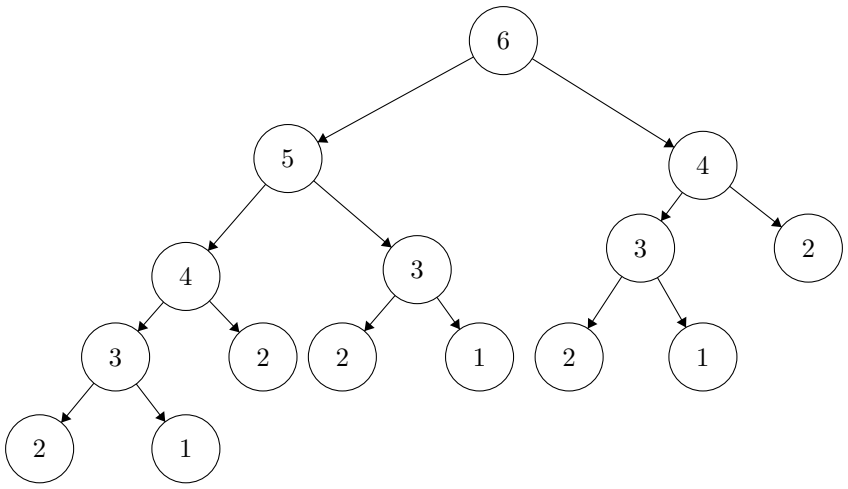
$$f(n) = \begin{cases} 1 \text{ se } n \leq 2 \\ f(n-1) + f(n-2) \text{ se } n > 2 \end{cases}$$

In pseudocodice:

```
1.  Fib(n)
2.      if n <= 2 then return 1
3.      if n = 0 return 0
4.      return Fib(n-1)+Fib(n-2)
```

Serie di Fibonacci - Ricorsione con memorizzazione

Il problema della soluzione ricorsiva del calcolo della serie di Fibonacci sta nel ricalcolo dello stesso problema molteplici volte.



Risolto il problema $f(6)$, avremmo calcolato 2 volte $f(4)$, 3 volte $f(3)$ e 5 volte $f(2)$. Per risolvere questo problema, è possibile implementare una ricorsione con memorizzazione usando un array. Sarà possibile conservare la soluzione ad $f(i)$ nell'i-esima locazione dell'array. In pseudocodice:

```
1.  Fib(n)
2.      if n <= 2 return F[n]
3.      if F[n-1] = NULL
4.      then F[n-1] = fib(n-1)
5.      if F[n-2] = NULL
6.      F[n-2] = Fib(n-2)
7.      return Fib[n-1]+Fib[n-2]
```

3.3 Problema dello zaino (o Knapsack problem)

Il **problema dello zaino** (o Knapsack problem) è un insieme di problemi di ottimizzazione tipici che possono essere risolti tramite un **approccio ricorsivo**. Dato uno zaino di portata k chilogrammi e una collezione A di n oggetti

$$A = \{a_1, a_2, a_3, a_4, ..., a_{n-1}, a_n\}$$

Indicheremo S l'insieme degli oggetti che decidiamo di inserire nello zaino, definito come:

$$S \subseteq A \sum_{i \in S} p(x) \leq k$$

Dove $p(x)$ associa all'oggetto x il proprio peso. Di conseguenza, la somma di tutti gli elementi dello zaino dovrà essere minore o uguale alla sua portata massima.

3.3.1 Massimizzare il numero di oggetti

$|S|$ è la cardinalità dell'insieme S. In questa versione del problema, vogliamo massimizzare il numero di oggetti, e quindi cercheremo, il massimo di $|S|$ possibile rispetto alla dimensione dello zaino, gli oggetti e il loro peso.
Partiamo da un array A **ordinato in maniera decrescente**.

$$Z(k, n) = \begin{cases} 0 \text{ se } n = 0 \\ 0 \text{ se } k < p(n) \\ 1 + Z(k - p(n), n - 1) \end{cases}$$

In alternativa

$$Z(k, n) = \begin{cases} 0 \text{ se } n = 0 \text{ o } k = 0 \\ -\infty \text{ se } k < 0 \\ \max(1 + Z(k - p(n), n - 1), Z(k, n - 1)) \end{cases}$$

In questo caso, stiamo scorrendo l'array di oggetti A dall'ultimo al primo elemento (per cui $n - 1$).
Come condizioni che interrompono la ricorsione, ovvero casi base, abbiamo la fine degli oggetti nella collezione A ($n = 0$) o il superare la capienza massima dello zaino ($k < p(n)$).
Per ogni elemento aggiunto allo zaino, incrementiamo il nostro risultato.

3.3.2 Massimizzare il peso degli oggetti

In questa istanza del problema, invece, cerchiamo il peso massimo ottenibile.

$$Z(k, n) = \begin{cases} 0 \text{ se } n = 0 \\ -\infty \text{ se } k < 0 \\ \max((p(n) + Z(k - p(n), n - 1)), Z(k, n - 1)) \end{cases}$$

L'algoritmo interrompe la ricorsione se finiscono gli oggetti.
Il peso viene posto a $-\infty$ se viene superata la portata.

Esempio di massimazione del peso

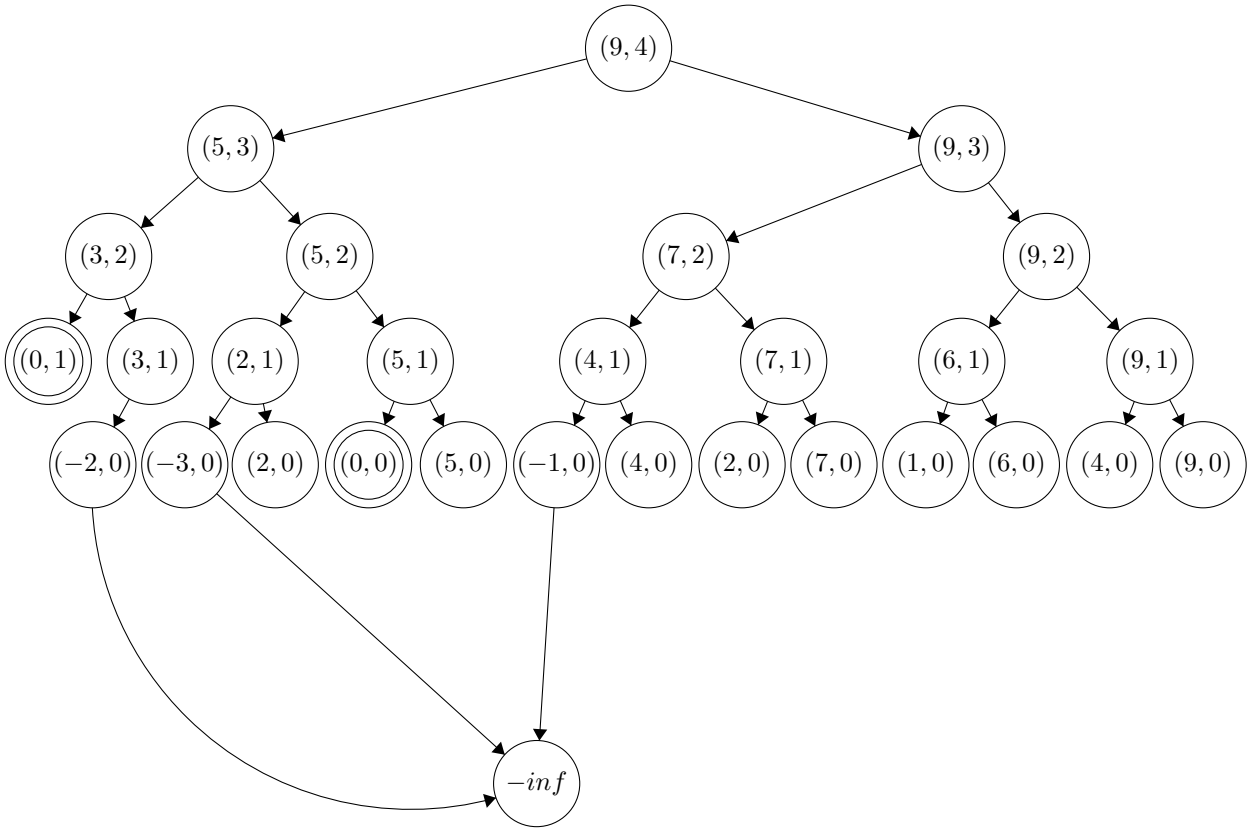
$$A = \{a_1, a_2, a_3, a_4\};\ k = 9$$

$$P(A) = [5, 3, 2, 4]$$

Riprendiamo l'algoritmo ricorsivo per risolvere questo specifico problema:

$$Z(k,n) = \begin{cases} 0 \text{ se } n = 0 \\ -\infty \text{ se } k < 0 \\ \max((p(n) + Z(k - p(n), n-1)), Z(k, n-1)) \end{cases}$$

Albero di ricorsione:



La soluzione al problema è 9.

3.3.3 Massimizzare il valore degli oggetti

Introduciamo un valore associato ad ogni oggetto, denotato con la funzione $v(i) = a_i$.
Stiamo cercando di trovare il valore complessivo massimo.
L'algoritmo sarà il seguente: $\sum_{i \in S} p(x) \leq k$
$\sum_{i \in S} v(x) = massima$

$$Z(k,n) = \begin{cases} 0 \text{ se } n = 0 \\ 0 \text{ se } k = 0 \\ -\infty \text{ se } k < 0 \\ \max(v(n) + Z(k - p(n), n-1), Z(k, n-1)) \end{cases}$$

Capitolo 4

Heap

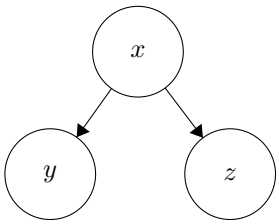
4.1 Introduzione all'heap

L'heap, in italiano "mucchio", è una struttura dati che può essere utilizzata per implementare una coda con priorità. La struttura dati in questione viene spesso astratta come un albero, ma vedremo successivamente che sarà possibile implementarla tramite array.

4.1.1 Proprietà dell'heap

L'heapv è un albero con le seguenti caratteristiche:

- **Binario.**
  Ogni nodo ha al più due figli.
- **Posizionale.**
  È possibile distinguere figlio sinistro e destro di ciascun nodo.
- **Completo.**
  In ogni livello i dell'albero, sono presenti 2^i nodi. Ricordiamo che il primo livello è sempre 0. Tuttavia, nel caso specifico dell'heap, l'ultimo livello può essere non completo, ma gli elementi devono essere distribuiti da sinistra verso destra.
- **Parzialmente ordinato.**
  La priorità di un nodo sarà sempre maggiore o uguale a quella dei suoi figli. I valori dei figli sono indipendenti tra loro.



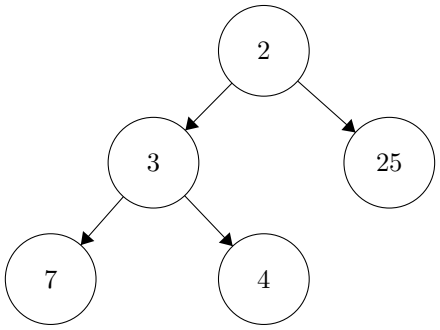
Dove $P(x) \geq P(y)$ e $P(x) \geq P(z)$. Si dirà ordinamento parziale in quanto non offrirà un'ordinamento uno a tutte le chiavi.

Inoltre, avrà sempre n nodi e altezza $h \in O(\log n)$

4.1.2 Tipi di heap

- **Min-Heap.**
  Priorità più alta a valori più piccoli.
- **Max-Heap.**
  Priorità più alta a valori più grandi

Esempio di min-heap:



4.2 Funzioni nei Min-Heap (Max-Heap)

- Valore del minimo (del massimo)
- Estrazione del minimo (del massimo)
- Inserimento
- Decrease-Key (Increase-Key)
- Delete
- Heapify (H, x)

4.2.1 Valore del minimo

Operazione in tempo costante $O(1)$, il minimo sarà sempre la radice.

4.2.2 Heapify

Operazione che, dato un albero in input e la sua radice, scambia la posizione delle chiavi in modo da far rispettare l'ordine parziale.

```
1.  HEAPIFY(H,x)
2.      y = left(x)
3.      z = right(x)
4.      min = x
5.      IF y != NULL AND key(y)<key(x)
6.          min = y
7.      IF y \neq NULL AND key(z)<key(min)
8.          min = z
9.      IF min != x
10.         swap(x, min)
11.         heapify(H,min)
```

La complessità sarà $O(\log n)$, ovvero uguale alla complessità con cui cresce l'albero.
Analizziamo nel dettaglio l'equazione di ricorrenza:

$$T(n) = aT(\frac{n}{b}) + f(n)$$

Nella peggiore delle ipotesi, ovvero nel caso di un albero sbilanciato, i due sottoalberi saranno di dimensione $(\frac{n}{3})$, dove n è il numero di nodi dell'albero. Inoltre, moltiplichiamo $(\frac{n}{3})$ per 2 in quanto il sottoalbero sinistro può avere al più il doppio dei nodi del sottoalbero destro.

$$T(n) = T(\frac{2}{3}n) + O(1)$$

4.2.3 Estrazione del minimo

Invece di rimuovere il nodo, inverto la chiave del nodo con l'ultimo nodo (da sinistra) dell'ultimo livello, per poi rimuovere il nodo vero e proprio contenente la chiave che vogliamo rimuovere, rendendo la procedura più semplice. Effettuo poi un Heapify.
Complessità $O(\log n)$

4.2.4 Inserimento di un nuovo nodo

```
1.  INSERT(H,k)
2.      x = new node(H)
3.      key(x) = k
4.      p = parent(x)
5.      while(p != NULL and key(p)>key(x)) do
6.          swap(x,p)
7.          x = p
8.          p = parent(x)
```

Complessità $O(\log n)$

4.2.5 Decrease-key

Nodo x, nuova chiave k (presumibilmente $P(x) \geq k$), albero H.

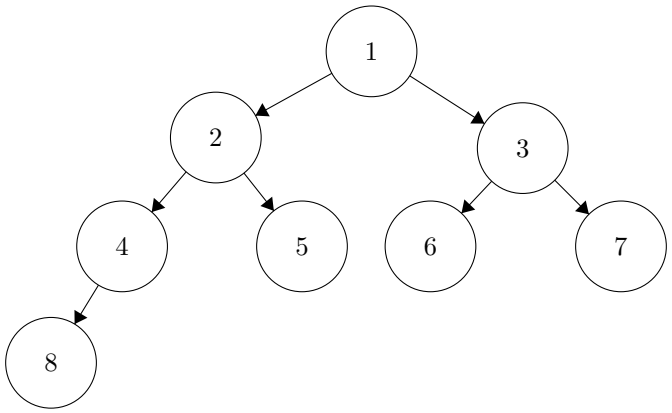
```
1.  DECREASE-KEY(H,x,k)
2.      key(x) = k
3.      p = parent(x)
4.      while(p != NULL and key(p)>key(x)) do
5.          swap(x,p)
6.          x = p
7.          p = parent(x)
```

Complessità $O(\log n)$

4.3 Astrazione vs Implementazione Heap

Descrivere le proprietà relative all'heap con l'ausilio di un albero, rende il tutto molto intuitivo.à 7L'astrazione dell'heap come albero è sicuramente consona e funzionale. Tuttavia, in termini di implementazione, le caratteristiche dell'heap ci permettono di poterlo implementarlo per mezzo di un array, aumentando radicalmente la velocità delle operazioni.

4.3.1 Heap come Array



$$H = \big[[1] \quad [2] \quad [3] \quad [4] \quad [5] \quad [6] \quad [7] \quad [8]\big]$$

4.3.2 Funzioni left, right, parent

```
1.  left(i)
2.      return 2i
```

```
1.  right(i)
2.      return 2i+1
```

```
1.  parent(i)
2.      return floor(i/2)
```

Implementando la moltiplicazione per 2 con un left shift, e la somma con 1 con un or logico, riduciamo il tempo di esecuzione delle operazioni:

```
1.  left(i)
2.      return (i<<1)
```

```
1.  right(i)
2.      return (i<<1)||1
```

```
1.  parent(i)
2.      return floor(i/2)
```

4.3.3 Heapify

```
1.  heapify(A, i):
2.      l = left(i)
3.      r = right(i)
4.      min = 1
5.      if (l <= heapsize and A[l] < A[min]):
6.          min = l
7.      if (r <= heapsize and A[r] < A[min]):
8.          min = r
9.      if (min != i):
10.         swap (A, i, min)
11.         heapify (A, min)
```

4.3.4 Insert

```
1.  insert(A, k):
2.      heapsize = heapsize + 1
3.      A[heapsize] = k
4.      i = heapsize
5.      p = parent(i)
6.      while(p != 0 and A[p]>a[i]):
7.          swap(A,i,p)
8.          i = p
9.          p = parent(i)
```

4.3.5 Extract-min

Una delle funzioni che rende l'heap il miglior candidato ad essere una coda con priorità.

```
1.  extract-min(A):
2.      if(heapsize) < 1:
3.          error "heap underflow"
4.      min = A[1]
5.      swap(A[1], A[heapsize])
6.      heapsize--
7.      heapify(A, 1)
```

4.3.6 Costruire un heap

Costruire un heap tramite albero binario richiede un tempo $O(n \log n)$ (n per scorrere la collezione di numeri che vogliamo posizionare nell'heap, $\log n$ come la crescita dell'heap). Tuttavia, l'heap è una struttura dati di tipo ricorsivo: è costituito da sotto-heap. Anche in un albero binario che non rispetta l'ordine che ogni heap deve rispettare, ogni foglia è da considerarsi un heap. Detto ciò, è possibile usare la funzione *heapify* dai parent delle foglie fino alla radice. Creiamo così la funzione *build-min-heap*:

```
1.  build-min-heap(A,n):
2.      for i = floor(n/2) down to 1:
3.          heapify(A,i)
```

Dove A è l'albero e n è il numero di elementi, Questa funzione, avrà complessità $O(n)$. Avendo definito due heapify diversi, uno per l'array, uno per gli alberi, allora la funzione *build-min-heap* sarà unica.

4.4 Heapsort

Implementare un selection sort che sfrutti l'heapify e le proprietà di ordinamento parziale dell'heap per trovare il massimo (o il minimo) dell'array, ci offre un algoritmo di complessità $O(n \log n)$, che chiameremo Heap Sort. Ordinamento crescente usando il max-heap:

```
1.  heapsort(A,n):
2.      build-max-heap(A,n)
3.      for i = 1 to n-1:
4.          extract-max(A)
```

Ordinamento decrescente usando il max-heap:

```
1.  heapsort(A,n):
2.      build-min-heap(A,n)
3.      for i = 1 to n-1:
4.          extract-min(A)
```

Capitolo 5

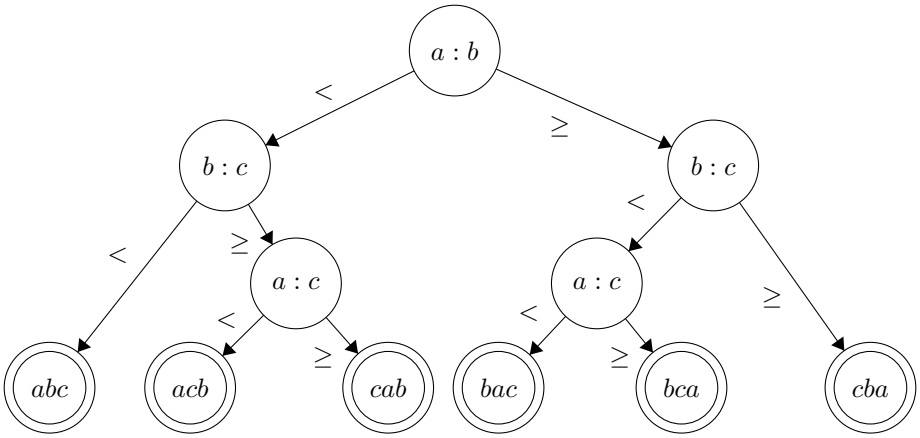
Algoritmi di Ordinamento a Tempo Lineare

5.1 Ordinamento basato sul confronto

Un algoritmo di ordinamento basato sul confronto funziona nel seguente modo:

1. Due elementi vengono confrontati, con lo scopo di ottenere un ordinamento relativo tra essi.

2. Ripeti il primo passaggio (iterativamente o ricorsivamente) ottenendo alla fine un'ordinamento totale.



Le foglie di questo albero sono tutte le possibili permutazioni dell'array $[a][b][c]$, e se il numero di permutazioni di n elementi è uguale a $n!$, allora diremo che l'altezza sarà:

$$h = O(log(n!))$$

Che approssimeremo, secondo l'approssimazione di Stirling a

$$h = O(n \log n)$$

Qualsiasi algoritmo faccia dei confronti, userà alberi di confronto analoghi a questo, indi per cui il problema dell'ordinamento tramite confronti, avrà come complessità minima $O(n \log n)$. Questa dimostrazione, valida esclusivamente per gli algoritmi di ordinamento basati sui confronti, non riguarderà i prossimi algoritmi di cui andremo a parlare.

5.2 Counting Sort

Array A di partenza:

$$A = \begin{bmatrix}[4] & [7] & [6] & [3] & [9] & [9] & [6]\end{bmatrix}$$

Array C di appoggio (dimensione maggiore o uguale all'elemento più grande):

$$C = \begin{bmatrix}[0] & [0] & [0] & [1] & [1] & [0] & [2] & [1] & [0] & [2]\end{bmatrix}$$

Dato un'array A di elementi $a_1, a_2, ..., a_n$, andremo a mettere nell'array C, nella casella $C[a_n]$ il numero di volte in cui appare l'elemento a_n. Andremo poi a ricostruire l'array A ordinato, basandoci su ciò che abbiamo scritto in C.

$$A' = \begin{bmatrix}[3] & [4] & [6] & [6] & [7] & [9] & [9]\end{bmatrix}$$

```
1.  counting_sort(A,n);
2.      k = max(A,n)
3.      C = new array(k + 1)
4.      for i = 0 to k do
5.          C[i] = 0
6.      for i = 0 to n - 1 do
7.          C[A[i]] = C[A[i]] + 1
8.      p = 0
9.      for i = 0 to k-1 do
10.         for j = 1 to C[i] do
11.             B[p] = i
12.             p = p + 1
```

Quando il valore degli elementi di C è molto alto, è spontaneo pensare che non valga la pena di avvalersi del Counting Sort;
Se la differenza tra il valore più piccolo e il più grande non è particolarmente alta, potremmo comunque usare il counting sort nel seguente modo:

1. Calcoliamo $max(A) - min(A) + 1$
2. Il numero appena calcolato sarà uguale alla dimensione di C necessaria.
3. Scorriamo l'array A da ordinare e incrementiamo a ogni iterazione di i, $C[A[i] - min]$
4. Ricostruaimo l'array in funzione di ciò che è scritto in C.

Possiamo anche lavorare con numeri decimali (moltiplicando per potenze di 10), negativi (sommando il numero più piccolo rendendolo lo 0), stringhe (se andiamo a convertire i caratteri in valori ASCII), strutture dati (rappresentabili tramite numero).

5.2.1 Problemi del Counting Sort

Data la natura dell'algoritmo in questione, non è possibile ordinare dati che non sono riconducibili a dei numeri. Ciò che andremo a fare inserendo i valori in A', è ricopiare le chiavi di confronto. Ricopiando le chiavi, andremo a perdere possibili dati aggiuntivi.

5.2.2 Modifica del vettore C e stabilità

È possibile andare a modificare il comportamento di C, propagando la somma degli elementi inseriti in C come nel seguente esempio:

$$C = \big[[0] \quad [0] \quad [0] \quad [1] \quad [2] \quad [2] \quad [4] \quad [5] \quad [5] \quad [7]\big]$$

In questo modo, andremo a memorizzare un'informazione differente, ma utile per velocizzare il bucket sort, che nella fase di ricostruzione dell'array, procederà nel seguente modo:

1. Leggiamo l'elemento a_n di A.
2. Accediamo alla casella $C[a_n]$ e ne leggiamo il valore.
3. Il valore coinciderà con il numero di elementi più piccoli di a_n nell'array ordinato, e quindi faremo: $A'[C[a_n] + 1] = a_n$

Inoltre, scorriamo l'array A da destra a sinistra per preservare l'ordine relativo degli elementi uguali.

```
1.  counting_sort(A,n);
2       h = min(A,n)
3.      k = max(A,n)
4.      C = new array(k - h + 1)
5.      for i = 0 to k do
6.          C[i] = 0
7.      for i = 0 to n - 1 do
8.          C[A[i]-h] = C[A[i]-h] + 1
9.      for i = 0 to n - 1 do
10.          C[i] = C[i] + C[i-1]
11.      for i = n-1 to 0 do
12.         B[C[A[i]-min]-1] = A[i]
13.         C[A[i]-min] =  C[A[i]-min] - 1
```

5.2.3 Counting Sort in Loco

È possibile implementare una sorta di Counting Sort in loco, ma esso non sarà stabile. È inoltre inevitabile al creazione del vettore C. Basterà infatti operare con degli swap dei vari elementi dell'array piuttosto che copiando gli elementi su un nuovo ipotetico array A'

Capitolo 6

Tabelle Hash

6.1 Introduzione alle Hash Table

Introdurremo il concetto di Tabella Hash partendo da un problema di implementazione: dobbiamo implementare un dizionario. Qual è la soluzione migliore, tenendo bene a mente le funzioni che vogliamo implementare?

6.1.1 Cos'è un dizionario?

Un dizionario è un insieme su cui possiamo effettuare operazioni di inserimento, cancellazione e soprattutto ricerca. La ricerca può portare a due esiti differenti:

- Ricerca con successo:
  ricerca di un elemento presente nella tabella
- Ricerca senza successo:
  ricerca di un elemento non-presente nella tabella.

6.1.2 Possibili implementazioni e complessità

- Array non-ordinato:
  Inserimento: $O(1)$
  Cancellazione: $O(1)$
  Ricerca: $O(n)$
- Array ordinato:
  Inserimento: $O(n)$
  Cancellazione: $O(n)$
  Ricerca: $O(\log n)$ *Ricerca Binaria*
- Lista non-ordinata:
  Inserimento: $O(1)$
  Cancellazione: $O(1)$
  Ricerca: $O(n)$

- Lista ordinata:
  Inserimento: $O(1)$
  Cancellazione: $O(1)$
  Ricerca: $O(n)$
- BST:
  Inserimento: $O(\log n)$
  Cancellazione: $O(\log n)$
  Ricerca: $O(\log n)$
- Tabella a Indirizzamento Diretto:
  Inserimento: $O(1)$
  Cancellazione: $O(1)$
  Ricerca: $O(1)$

6.1.3 Tabelle a Indirizzamento Diretto

Una tabella a indirizzamento diretto è una soluzione molto conveniente, nei casi in cui l'insieme universo U è un insieme relativamente piccolo.
Una tabella a indirizzamento diretto avrà uno slot per ogni elemento $x \in U$. Ad ogni elemento è associata una chiave k, ovvero un indice della tabella a indirizzamento diretto.
In alcuni casi, potremmo addirittura usare il dato stesso come chiave! Così facendo, ci basterà contenere un solo dato all'interno della tabella, che indichi la presenza o meno dell'elemento $T[x.key] = T[x]$ (a cui, per comodità, ci riferiremo solo con k) nel nostro dizionario!
Di seguito, gli pseudo-codici di inserimento, rimozione e ricerca nelle T.I.D.
Tutte le operazioni in questione vantano complessità $O(1)$.

Inserimento:

```
1.    Insert(T,k)
2.      T[k] = 1
```

Rimozione:

```
1.    Insert(T,k)
2.      T[k] = 0
```

Ricerca:

```
1.    Search(T,k)
2.      if T[k] = 1
3.           return true
4.      else return false
```

I contro

Apparentemente, le Tabelle a indirizzamento diretto sono perfette; tuttavia, esistono dei casi in cui non sono veramente consone, e in cui anzi, non possono essere proprio utilizzate:

- **Insieme U molto grande rispetto a K**
  Se l'insieme universo U è molto grande, ma l'insieme K delle chiavi da memorizzare è piccolo, la memoria da allocare sarà quasi del tutto sprecata.
- **L'insieme universo è infinito.**
  Un calcolatore non ha memoria infinita, e una tabella a indirizzamento diretto ha bisogno di un numero finito di indici.

È possibile tuttavia andare a introdurre un intermedio tra l'insieme U e una nuova tabella: la funzione di Hashing.

6.2 Hashing

Il ruolo della funzione hashing è quello di associare ad un elemento k dell'insieme $S \subseteq U$, un indice della tabella Hash.

$$h : k \to T[h(k)]$$

Le spazio richiesto da una tabella Hash sarà ridotto a $\Theta(|K|)$. Sarà uguale quindi al numero di chiavi da memorizzare, molto di meno rispetto a $\Theta(|U|)$. Invece, la ricerca manterrà complessità $O(1)$, ma solo nel caso medio.

6.2.1 Collisioni

La funzione $h(k)$, avendo solitamente un codominio di cardinalità inferiore al dominio, non è una funzione biunivoca. Per il pigeonhole principle, se gli elementi da inserire nella Hash Table sono più degli indirizzi, almeno due chiavi finiranno allo stesso indirizzo.
Quando due chiavi finiscono nella stessa cella, abbiamo un fenomeno chiamato "collisione".

6.3 Hashing con Concatenazione

Facciamo in modo che la nostra Tabella Hash contenga dei puntatori ad una lista concatenata. Il problema delle collisioni viene risolto, ma non avremo più una tabella a valori binari. Andiamo quindi a modificare le nostre tre operazioni principali e a studiarne la complessità:

Inserimento:

```
1.    Insert(T,k)
2.      List-Insert(T[h(k)], k)
```

E avrà $O(1)$, come l'inserimento in lista.

Rimozione:

```
1.    Insert(T,k)
2.      List-Delete(T[h(k)], k)
```

E avrà $O(1)$, come la cancellazione dalla lista.

Ricerca:

```
1.    Search(T,k)
2.      return List-Search(T[h(k)], k)
```

Il worst case sarà quello in cui ogni elemento è inserito nella stessa posizione della tabella. $O(n)$, come la ricerca in una lista di n elementi. Andiamo a studiare tuttavia il caso medio, introducendo il concetto di uniformità.

6.3.1 Principio di Hashing Uniforme Semplice

Se la probabilità che un elemento venga inserito in una locazione di memoria è uguale tra tutti gli elementi, diremo che la funzione di Hashing è uniforme.
Con $|T| = m$, ovvero m numero di locazioni all'interno della tabella hash, la probabilità che un elemento venga inserito in una determinata locazione sarà pari a $\frac{1}{m}$. Il fattore di carico, denotato con α, indicherà la lunghezza media di una lista all'interno di una tabella Hash con concatenazione, ed è dato dal rapporto:

$$\frac{n}{m} = \alpha$$

In questo modo, otterremo un valore tramite cui possiamo indicare la complessità della ricerca, nel caso medio, ovvero $O(\alpha)$.

6.3.2 Implementare una funzione hash

Sono molti i metodi utilizzati per calcolare l'hashing di una data chiave k.

Metodo della divisione

Con $k \in U$, $h : U \to \{0, 1, ..m-1\}$ e $|T| = m$

$$h(k) = k \; mod \; m$$

Esempio:
Con 10 slot nella tabella, associerò T[9] al numero 169, T[0] al numero 10, T[3] a 1673, ma anche a 3, 63 e 5003. Il posizionamento di valori molto differenti, dipendendo in questo caso da una piccola porzione dell'informazione (in questo caso, la cifra delle unità). Non offre una distribuzione ottimale dei valori. Per questo motivo, con questo metodo, andiamo a evitare tabelle hash di dimensione $m = 2^p$, dove p è un qualsiasi intero positivo. Perché?
Nel caso di numeri positivi rappresentati in binario, all'interno di una tabella hash di dimensione 2^p, l'hashing dipenderà esclusivamente dalle prime p cifre binarie. Se l'hashing dipende solo da una piccola parte del dato, non offre un hashing uniforme, ed è quindi sconsigliato.

Metodo della moltiplicazione

Con $k \in U$, $h : U \to \{0, 1, ..m-1\}$ e $0 < A < 1$

$$h(k) = \lfloor (k \cdot A \; mod \; 1)m \rfloor$$

Spieghiamo il principio di di questo metodo:

1. $k \cdot A$ ci ritorna un valore compreso tra 0 e k.
2. Il resto della divisione per 1 con un numero reale, ritornerà il valore della parte decimale, ottenendo un valore compreso tra 0 e 1, 1 escluso.
3. Moltiplicando per m, otteniamo valori compresi tra 0 e m, m escluso.
4. La funzione $floor$ ci permette di passare da valori continui a valori discreti, e quindi $h : U \to \{0, 1, ..m-1\}$.

Risulta quindi più conveniente il secondo metodo.

6.3.3 Metodo della moltiplicazione con operazioni bitwise

Per rendere meno dipendiosa la funzione del metodo della moltiplicazione, uso delle operazioni bitwise.
Sia w la dimensione di una word. A è compreso nell'intervallo $(0, 1)$. Possiamo sicuramente rappresentare A come un intero se spostiamo a sinistra i bit di A di w posti.
Approfondisci questa parte sul libro.

6.4 Hashing a indirizzamento aperto

Un'altro metodo che permette di risolvere le collisioni, è l'indirizzamento aperto. Supponiamo di avere una tabella con un numero m di slot. Se cercando di effettuare un inserimento nella tabella, la posizione calcolata dalla funzione di hashing risulta già occupata, verrà calcolato e ispezionato un altro slot.
Ciò avverrà fino a quando non si troverà uno slot libero (contenente $NULL$ o D, ne parleremo dopo), o fino a quando non si saranno controllati tutti gli m slot della tabella.
In quel caso, l'inserimento non avrà successo, a causa della tabella piena.

6.4.1 Funzione di hashing per indirizzamento aperto

La funzione di hashing all'interno di una tabella con indirizzamento aperto, sarà così definita:

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

Definiamo "sequenza di ispezione" di una chiave k, la sequenza di indici che viene generata da tutte le funzioni di hashing calcolate su k, ovvero:

$$\langle h(k, 0),\ h(k-1), \ldots,\ h(k, m-1)\rangle$$

E sarà sempre uguale ad una permutazione di $\langle 0, 1, \ldots, m-1\rangle$. La sequenza di ispezione conterrà tutti gli slot della tabella.

6.4.2 Cancellazione

La funzione di cancellazione, all'interno di una tabella hash che risolve le collisioni tramite l'indirizzamento aperto, richiederà una modifica fondamentale per il funzionamento corretto della funzione di ricerca.
Cancellato un elemento dalla tabella, esso non dovrà essere sostituito da $NULL$. Esso dovrà infatti essere contrassegnato come D, di $DELETED$. Esponiamo un caso esempio che spieghi il motivo dietro questa modifica.

Caso esempio

Una tabella con $m = 4$ contiene i numeri $[1, 2, 3, 5]$.
Eliminiamo il numero 3 dalla tabella, ottenendo $[1, 2, NULL, 5]$.
La sequenza di ispezione della chiave 5 è la seguente:

$$h(5, 0) = 1;\ h(5, 1) = 0;\ h(5, 2) = 2;\ h(5, 3) = 3;$$

Supponiamo adesso di usare la funzione di ricerca per verificare la presenza (evidente) del numero 5, ricordandoci che la funzione si fermerà (con esito negativo) trovato uno slot contrassegnato con $NULL$.
La ricerca si interromperà a $h(5, 2) = 2$. Ciò non sarebbe avvenuto prima dell'eliminazione del numero 3, ed è causato proprio dal $NULL$ in posizione 2.

Contrassegnare lo slot con $DELETED$ espliciterà il fatto che lo slot è libero in caso di inserimento, ma anche che la ricerca non andrà interrotta nonostante lo slot vuoto.

6.4.3 Quando preferire l'indirizzamento aperto

Considerando tutti questi fattori, usare l'indirizzamento aperto risulta molto comodo quando non sono contemplate operazioni di cancellamento. Non dover usare puntatori ci permetterà di poter dedicare più memoria alla tabella stessa e meno agli oggetti da contenere, permettendo ad una tabella ad indirizzamento aperto, di mantenere una complessità molto più bassa, spesso a parità delle risorse usate da una tabella hash con concatenazione contenente gli stessi elementi.

6.4.4 Pseudo-codice

Hash-Insert

```
1. hash_insert(T,k)
2.  IF n = m:        //tabella piena
3.      RETURN
4.  i = 0
5.  WHILE(T[h(k,i)] != NULL AND T[h(k,i)] != DELETED):
6.      i = i + 1
7.  T[h(k,i)] = k
8.  n = n + 1
```

Hash-Search

```
1. hash_search(T,k)
2.  i = 0
3.  WHILE T[h(k,i)] != NULL AND i < m
4.      IF T[h(k,i)] = k:
5.          RETURN True
6.      i++
7.  RETURN False
```

Hash-Delete

```
1. hash_delete(T,k)
2.  i <- 0
3.      WHILE(T[h(k,i)] != NULL AND i < m):
4.          if T[h(k,i)] = DELETED:
5.              RETURN True
6.          i = i+1
7.      RETURN False
```

6.5 Funzioni hash per indirizzamento aperto

P_m è l'insieme di tutte le permutazioni sull'insieme $\{0, \ldots, m-1\}$. La cardinalità di quell'insieme è $m!$.
Sia $p \in P_m$ una permutazione. La probabilità che $h(k) = p$ deve essere la stessa per qualsiasi permutazione.

$$Pr[h(k) = p\ ] = \frac{1}{m!}\ \forall p \in P_m$$

Costruire funzioni hash per l'indirizzamento aperto che rispettino questa proprietà è piuttosto complesso.

6.5.1 Scansione lineare

La funzione di hashing per la scansione lineare è definita in questo modo

$$h(k, i) = (h'(k) + i) \mod m$$

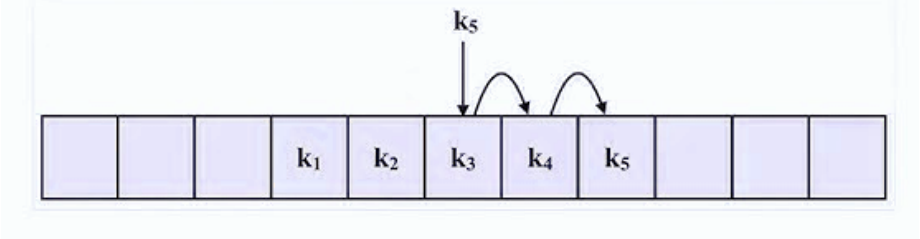
Con $h' : U \to \{0, 1, \ldots, m-1\}$. Le sequenze della nostra funzione di hashing saranno del tipo

$$\langle h'(k), h'(k)+1, h'(k)+2, \ldots, h'(k)+(m-1)\rangle$$

Questa funzione di hashing non gode della proprietà di hashing uniforme, in quanto, le permutazioni ottenute non possono non iniziare con $h(k)$. Moltissime permutazioni avranno probabilità 0, e le restanti $\frac{1}{m}$.

Agglomerazione primaria

Un altro fenomeno relativo al non-uniformismo della funzione di hashing ottenuta con scansione lineare, è l'agglomerazione primaria.



In breve, la probabilità che le chiavi siano inserite in slot successivi aumenta ad ogni inserimento, favorendo agglomerati di chiavi e allungando le tempistiche relative alle operazioni.
Un'altra funzione di hashing basata sulla scansione lineare, è la seguente

$$h(k, i) = (h'(k) + 2i) \mod m$$

Con $h' : U \to \{0, 1, \ldots, m-1\}$. Le sequenze della nostra funzione di hashing saranno del tipo

$$\langle h'(k), h'(k)+2, h'(k)+4, \ldots \rangle$$

6.5.2 Scansione quadratica

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m$$

Con c_1 e c_2 costanti scelte in modo tale che l'intera tabella venga scansionata dalla funzione $h(k,i)$.
Questa funzione di hashing soffre di aggregazione secondaria, un fenomeno che risulta essere molto meno grave.

6.5.3 Hashing doppio

Definiamo due funzioni hashing diverse tra loro

$$h', h'' : U \to \{0, 1, \ldots, m-1\}$$

Possiamo usare entrambe le funzioni hash per realizzare la seguente funzione hash a indirizzamento aperto:

$$h(k,i) = (h'(k) + i h''(k)) \mod m$$

Il numero di permutazioni totale è $m \times m = m^2$, in quanto la prima funzione di hashing da la prima posizione, la seconda le successive, e quindi m possibili posizioni dopo altre m.
In questo modo, anche se due chiavi diverse hanno lo stesso hashing in h', lo stesso non succederà per h''.
Anche nel caso peggiore, abbiamo sempre m permutazioni.

Capitolo 7

Alberi rosso-neri

7.1 Cosa sono gli alberi rosso neri?

Sono una particolare estensione degli alberi binari di ricerca (BST). Sono spesso definiti come degli alberi auto bilancianti in quanto, grazie alle proprietà che rendono gli alberi rosso-neri tali, mantengono un buon bilanciamento rispetto al numero di nodi.

7.2 Perché preferire un albero rosso-nero ad un BST?

Per rispondere a questa domanda, basta porsene un'altra:
perché vogliamo mantenere un buon bilanciamento all'interno di un albero di ricerca?
La ricerca all'interno di un Binary Search Tree è molto sensibile al suo bilanciamento. La complessità delle operazioni dipende dall'altezza dell'albero, che può variare tra $\log n$ nel caso migliore (albero bilanciato) e n nel caso peggiore (albero degenere). Introdurre un meccanismo di auto bilanciamento permetterebbe di non ottenere mai un albero degenere, e mantenere una complessità poco più che logaritmica.

7.2.1 Ripasso: proprietà dei BST

Un albero binario di ricerca è costituito esclusivamente da nodi che rispettano la seguente proprietà: ogni nodo è maggiore o uguale a tutti i nodi del proprio sottoalbero sinistro, e minore o uguale a tutti i nodi del proprio sottoalbero destro.

7.3 Proprietà degli alberi rosso-neri

Quando andiamo ad operare con gli RB-Trees, dobbiamo andare a introdurre un meccanismo per specificare il colore dei nodi: includiamo quindi un bit che permetta di identificare un nodo nero (bit : 1) e un nodo rosso (bit : 0). Inoltre, a parte le proprietà relative ai BST, ogni RB-Tree rispetta le seguenti proprietà:

1. Ogni nodo o è rosso o è nero.
2. La radice è nera.
3. Le foglie sono nere.*
4. Un nodo rosso ha solo figli neri.
5. Un cammino da un nodo ad una sua foglia contiene sempre lo stesso numero di nodi neri.

7.3.1 Rilassamento terza proprietà*

Non consideriamo la terza proprietà, supponiamo essa venga sempre rispettata. È possibile aggiungere delle foglie contenenti $NULL$ a cui associamo il colore nero. Il numero di nodi passa da n a $2n$, l'altezza incrementa di 1. Computazionalmente le prestazioni rimangono tali. A livello implementativo, evitiamo addirittura di creare ogni singola foglia $NULL$, facendo puntare tutte le foglie ad un unico nodo $NULL$, in gergo chiamato "sentinella". Tutto ciò non è obbligatorio da rappresentare nell'astrazione, ma è da tenere a mente per l'implementazione.

7.3.2 Sulla quinta proprietà

Definiamo la funzione $bh(x)$ come l'altezza nera del nodo x. L'altezza nera del nodo x indica il numero di nodi neri che si trova tra il nodo x e tutte le sue foglie, senza considerare il nodo di partenza. [N.B. le foglie $NULL$ sono nere, e da tenere in considerazione.]

7.3.3 Altezza degli RB-Tree

Queste proprietà dell'albero rosso-nero, limitano la sua altezza ad un valore massimo, ovvero:

$$h \leq 2log_2(n+1)$$

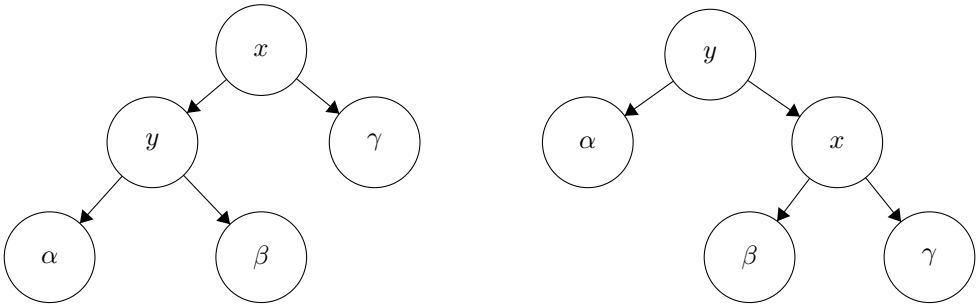
E la differenza tra la lunghezza del ramo più lungo e quello più corto, non sarà più grande di 2.

7.4 Operazioni di base su alberi rosso-neri

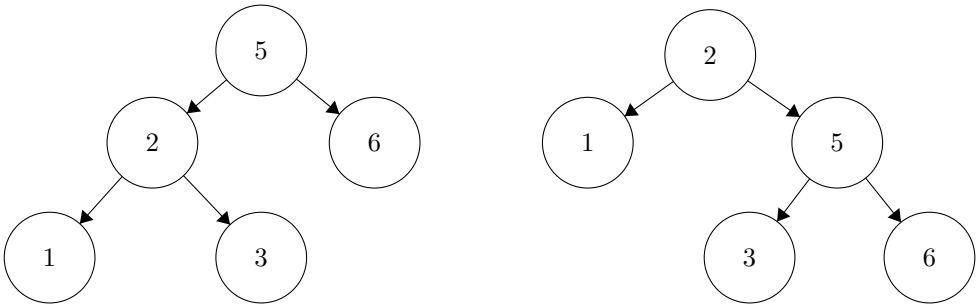
Mantenere configurazioni che rispettino le cinque proprietà precedentemente citate, implica l'ausilio delle seguenti operazioni.

7.4.1 Rotazioni

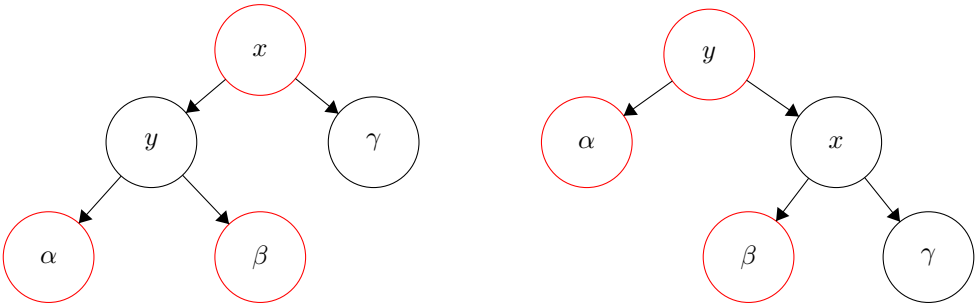
Rotazione destra su $x \Leftrightarrow$ Rotazione sinistra su y.



Le operazioni di rotazione destra e sinistra sono definite su qualsiasi tipo di albero.
Inoltre, negli alberi binari di ricerca, le rotazioni non alterano le sue proprietà.



Da ora in poi, quando applicheremo una rotazione su un albero rosso-nero, i nodi che si invertono di posizionedovranno scambiare la propria colorazione.



7.4.2 Ricolorazione

Ricolorazione su $x \Leftrightarrow$ Ricolorazione su x.

Banalmente, un nodo rosso ricolorato diventa nero, un nodo nero ricolorato diventa rosso.

7.4.3 Inserimento

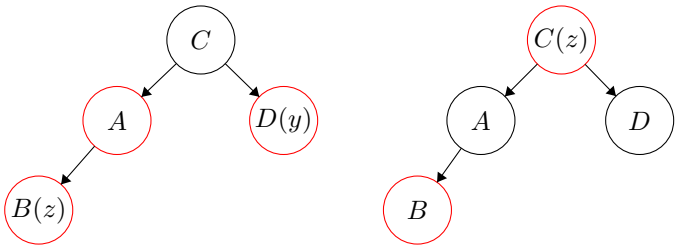
L'inserimento è probabilmente la seconda operazione più complessa all'interno degli RB-Tree. Non è infatti possibile implementare un'operazione che, data qualsiasi configurazione di un RB-Tree, inserisca un nodo senza influenzarne le cinque proprietà. Se non con l'ausilio di una funzione, chiamata Insert-Fixup, capace di ripristinare le proprietà successivamente a un qualsiasi inserimento, e che quindi richiameremo alla fine della funzione di inserimento.
Detto ciò, la procedura d'inserimento è analoga a quella dei BST. Ogni nodo nuovo è di default rosso. Avremo quindi:

1. Creazione del nuovo nodo rosso
2. Inserimento analogo ai BST
3. Chiamata a funzione RB-Insert-Fixup sul nuovo nodo

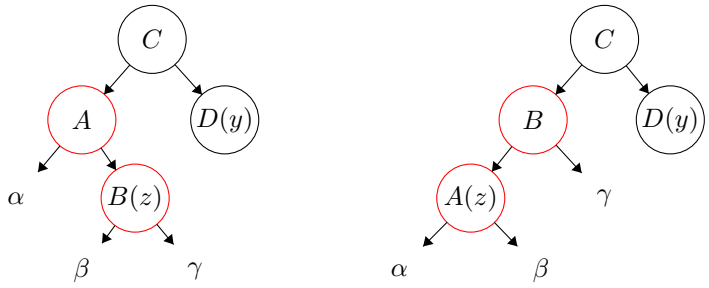
7.4.4 RB-Insert-Fixup

La funzione RB-Insert-Fixup basa il suo funzionamento su tre casi. Studiamo i tre casi in questione, supponendo che z sia il nuovo nodo, e studiando y, suo zio.

1. Caso 1, lo zio y è rosso:
   Se lo zio di z è rosso, allora il nonno di z (parent sia del parent di z sia di y) è nero. La colorazione del nonno di z viene messa ad entrambi i figli. Richiamiamo RB-Inserti-Fixup sul nonno di z, per correggere possibili violazioni nel resto dell'albero.

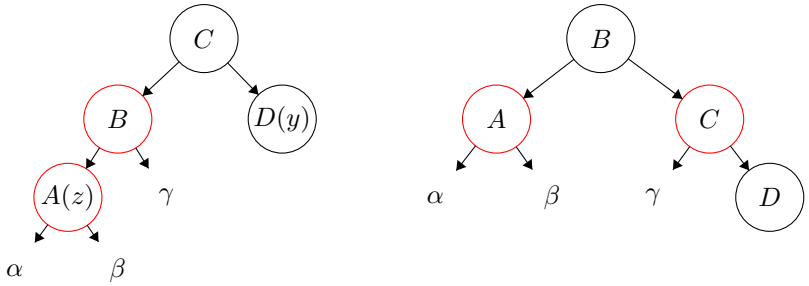


2. Caso 2, lo zio y è nero e z è interno:
   Ruotiamo il parent di z in modo tale da mettere z nella posizione del parent.



   Avremmo ottenuto una situazione tale da poterci ricondurre al caso 3. Richiamiamo ricorsivamente la procedura sulla nuova z.

3. Caso 3, lo zio y è nero e z è esterno:
   Ruotiamo il nonno di z in maniera tale da mettere il parent di z nella posizione del nonno.



7.4.5 Rimozione negli RB-Trees

Ripasso: rimozione negli alberi BST

- Primo caso: cancellazione foglia:
  Caso banale, cancello il nodo.
- Secondo caso: cancello nodo con un solo figlio:
  L'unico figlio prende il posto del padre
- Terzo caso: cancello nodo con due figli:
  Cancello la chiave del nodo che voglio eliminare, la sostituisco alla chiave del nodo più a sinistra del sottoalbero destro, elimino il nodo più a sinistra del sottoalbero destro.

Rimozione negli RB-Trees

Il nodo che vogliamo rimuovere sarà indicato con z.

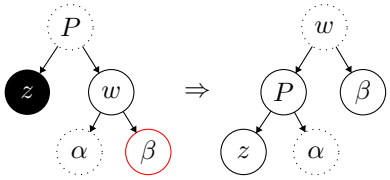
- Caso a, z foglia rossa
  Caso banale, rimuovo z.
- Caso b, z rosso con un solo figlio e padre nero:
  Banale, cancello il figlio di z diventa figlio del parent di z e elimino z.
- Caso c, z nero con figlio rosso:
  Il figlio prende il posto di z e diventa nero. z viene eliminato.
- Caso d, il nodo è una foglia nera:
  Denoto il $NULL$ come **doppio-nero** e cancello il nodo.
- Caso e, il nodo da cancellare è nero, ha un figlio nero:
  Rimuovo il nodo, denoto il figlio nero come **doppio-nero**.

Gli ultimi due casi ci danno configurazioni anomale con un colore non accettato nella proprietà (1)! Ciò richiederà una funzione chiamata RB-Delete-Fixup.

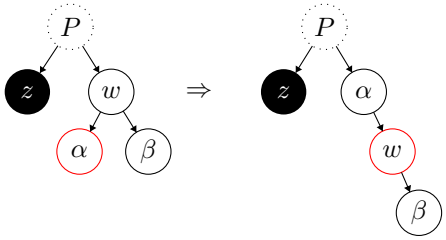
7.4.6 RB-Delete-Fixup

Osserviamo il fratello w del nodo doppio-nero z:

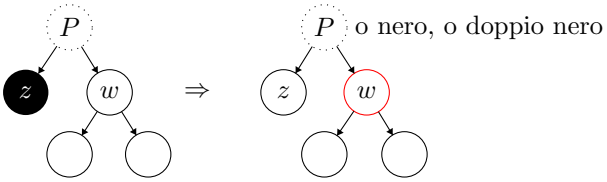
1. Caso 1, w è nero, almeno uno dei suoi figli è rosso:
   (a) Figlio esterno rosso:
   Ruoto il parent di w facendo "salire" w, il doppio nero diventa nero e il figlio esterno, precedentemente rosso, diventa nero.



   (b) Figlio esterno nero ma interno rosso:
   Ruoto il figlio rosso verso l'esterno, facendolo diventare padre di w, che diventerà figlio esterno rosso, potendoci rifare al caso 1(a).



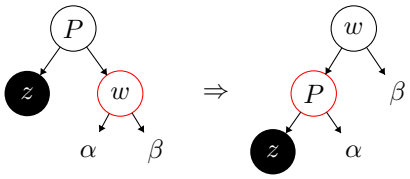
2. Caso 2, w nero e entrambi i suoi figli sono neri: si propaga la colorazione di w e del doppio-nero z al padre, che se è rosso diventerà nero, se è nero diventerà doppio nero.



   (a) Il padre è diventato nero:
   Procedura finita.

   (b) Il padre è diventato nero:
   Richiamo la funzione di Fix-Up sul padre doppio-nero.

3. Caso 3, w è rosso: ruoto il padre di w facendo risalire w e richiamo di nuovo la procedura sul doppio nero.



Se un nodo doppio-nero diventa la radice dell'intero albero, diventerà singolarmente nero, riducendo l'altezza nera dell'albero di 1.

Capitolo 8

Programmazione dinamica

8.1 Introduzione ai problemi di ottimizzazione

Alcuni tra i più importanti problemi computazionali, ricadono nella categoria dei problemi di ottimizzazione.
Lo scopo di un problema di ottimizzazione, è quello di trovare una soluzione "ottima".
Una soluzione è ammissibile se rispetta un determinato numero di criteri (stabiliti dal problemi), ma non è sempre detto sia ottima!
Una soluzione ottima, infatti, è una soluzione migliore delle altre. Ciò verrà stabilito tramite un criterio, chiamato "funzione costo".

8.1.1 Formalizzazione problemi di ottimizzazione

Il termine "ottimizzazione" fa proprio riferimento alla scelta della soluzione ottima, e non, come si potrebbe pensare erroneamente, al ridurre la complessità temporale e spaziale di un algoritmo (anche se il nostro obiettivo, è sempre trovare soluzioni valide ed efficienti).

Definizione

Dato un problema P, esiste un insieme $S^P = (S_1, S_2, S_3, ..., S_n)$ di soluzioni al problema P.
Esiste inoltre una funzione $f_{costo} : S^P \to R$, che associa un valore alla soluzione S_n. Il nostro obiettivo sarà trovare il valore S_i tale che :

$$S_i : f_{costo}(S_i) = \min\{f(S_j) : S_j \in S^P\}$$

oppure

$$S_i : f_{costo}(S_i) = \max\{f(S_j) : S_j \in S^P\}$$

Ovvero, trovare la soluzione che minimizza (o massimizza) il valore di $f_{costo}(S_i)$.

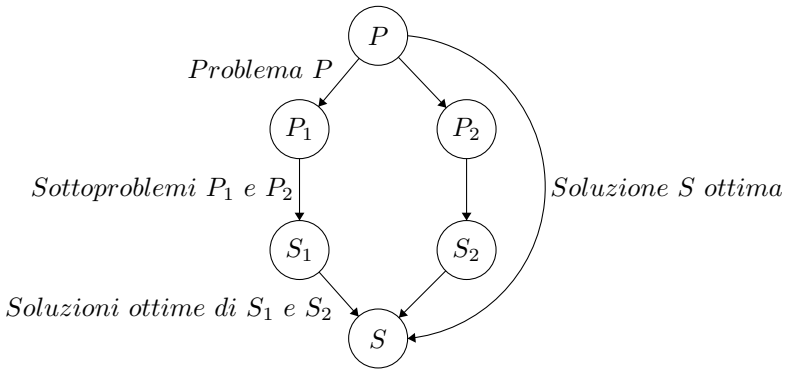
8.2 La programmazione dinamica

La programmazione dinamica è una tecnica che può essere applicata per risolvere alcuni problemi di ottimizzazione.
I problemi in questione, dovranno però presentare determinate caratteristiche.

8.2.1 Quali problemi possiamo risolvere?

Un problema di ottimizzazione risolvibile con la programmazione dinamica presentare i seguenti elementi:

- **Overlapping subproblems.**
  Il problema deve presentare un numero polinomiale di sottoproblemi che si ripetono[1], con una struttura ricorsiva.
- **Sottostruttura ottima.**
  Data una soluzione ottima, le sue restrizioni ai sottoproblemi sono ottime per i sottoproblemi.



[1] Un esempio di sottoproblemi che si ripetono, sono quelli relativi al calcolo dell'n-esimo numero della serie di Fibonacci, affrontato nel terzo capitolo di questi appunti

8.3 Rod-Cutting Problem

Il Rod-Cutting problem è un problema di ottimizzazione tipico, e risolvibile con la programmazione dinamica. Ci approcceremo a questo problema usando tre soluzioni differenti, due ricorsive e una iterativa.

8.3.1 Introduzione al Rod-Cutting problem

Testo del problema

Un'azienda si occupa di lavorazione di barre metalliche. Data una barra di lunghezza n, ad ogni taglio di barra è associato un costo.

lunghezza n	1	2	3	4	5	6	7	8	9	10
prezzo p	1	5	8	9	10	17	17	20	24	30

Bisogna massimizzare il guadagno scegliendo una combinazione di tagli ottima. Nota bene, la versione base del problema chiede il guadagno massimo, piuttosto che la combinazione in se. Scriveremo tuttavia un algoritmo che ci permetterà di esplicitare la combinazione (che sarà una soluzione ottima).

Funzione r(n)

$r(n)$ è una funzione che associa ad una barra il guadagno massimo che posso ottenere da essa.

$$r(n) = \begin{cases} 0 \text{ se } n = 0 \\ \max\{P(i) + r(n-i) : 1 \leq i \leq n\} \text{ altrimenti} \end{cases}$$

8.3.2 Rod-Cutting ricorsivo top-down

Soluzione ricorsiva in pseudo-codice:

```
1.  ROD-CUTTING(n,p)
2.      IF n=0 THEN RETURN 0
3.      q = -infinito
4.      FOR i=1 TO n+1 DO
5.          r = p[i-1] + ROD-CUTTING(n-i,p)
6.          IF q<r THEN
7.              q = r
8.          RETURN q
```

Soluzione ricorsiva in Python:

```python
# A Recursive Solution to the Rod-Cutting problem
INT_MIN = -32767

def cutRod(price, n):
if n == 0:
return 0
ans = INT_MIN
for i in range(1,n+1):
r = p[i-1] + cutRod(price,n-i)
if ans<r:
ans = r
return ans

p = [1,5,8,9,10,17,17,20,24,30]
n = 10
print(cutRod(p,n)) #output : 30
```

8.3.3 Rod-Cutting ricorsivo con memorizzazione

La seguente è una soluzione top-down ricorsiva che implementa un meccanismo di memorizzazione per evitare computazioni ridondanti.

```
1.  M-ROD-CUTTING(n,p)
2.      r = NEW ARRAY(n+1)
3.      FOR i = o TO n DO r[i] = -infinito
4.      r[0] = 0
5.      ROD-CUTTING-AUX(n,p)
```

Sarà richiesta anche una funzione ausiliaria.

```
1.  M-ROD-CUTTING-AUX(n,p)
2.      IF n = 0 THEN RETURN 0
3.      q = -infinito
4.      FOR i = 1 TO n DO
5.          IF(r[n-1] = -infinito) THEN
6.              r[n-1] = ROD-CUTTING-AUX(n-i,p)
7.          IF(q<P(i) + r[n-i]) THEN
8.              q = P(i) + r[n-1]
9.      RETURN q
```

8.3.4 Soluzione bottom-up con tabulazione

Implementiamo ora una soluzione bottom-up con tabulazione senza ricorsione.

```
1.  D-ROD-CUTTING(n,p)
2.      r = NEW ARRAY(n+1)
3.      r[0] = 0
4.      FOR k = 1 TO n DO
5.          r[k] = -infinito
6.          FOR i = 1 TO k DO
7.              IF r[k] < P(i) + r[k-i] THEN
8.                 r[k] = P(i) + r[k-i]
9.      RETURN r[n]
```

Implementeremo una versione con un nuovo array t contenente i tagli.

```
1.  D-ROD-CUTTING(n,p)
2.      t = NEW ARRAY(n+1)
3.      r = NEW ARRAY(n+1)
4.      r[0] = 0
5.      FOR k = 1 TO n DO
6.          r[k] = -infinito
7.          FOR i = 1 TO k DO
8.              IF r[k] < P(i) + r[k-i] THEN
9.                 r[k] = P(i) + r[k-i]
10.                t[k] = i
11.     RETURN r[n]
```

Soluzione bottom-up in Python senza array t:

```python
# A Bottom-Up Solution to the Rod-Cutting problem
INT_MIN = -32767

def bottomUpCutRod(price, n):
r = [0] * (n + 1) # Inizializzazione Array r
for j in range(1,n+1):
q = INT_MIN
for i in range(1, j+1):
if q < price[i-1]+r[j-i]:
q = price[i-1]+r[j-i]
r[j] = q
print('Max Sell Price for', j, ':',r[j],'\n')
return r[n]

p = [1,5,8,9,10,17,17,20,24,30]
n = 10
print(bottomUpCutRod(p,n)) #output : 30
```

8.4 Problema della moltiplicazione di matrici

Conosciuto come "Matrix chain multiplication problem", è un problema di ottimizzazione basato proprio sul prodotto riga per colonna tra matrici.
Questa operazione gode della proprietà associativa.
Associando in maniera diversa i vari prodotti tra matrici, è possibile ottenere lo stesso risultato con numeri diversi di operazioni scalari.

8.4.1 Come si effettua una moltiplicazione tra matrici?

Date due matrici A e B, il prodotto tra queste due matrici sarà uguale alla somma dei prodotti tra l'i-esimo elemento della j-esima riga di A e l'i-esimo elemento della j-esima colonna di B.
L'unica condizione che queste matrici dovranno rispettare, è la seguente:
Il numero di colonne di A deve essere uguale al numero di righe di B. Riportiamo di seguente un algoritmo per il prodotto riga per colonna tra due matrici

```
1.  MATRIX-MULTIPLTY(A,B,p,q,w)
2.      C = NEW MATRIX[p,w]
3.      FOR i = 1 TO p DO
4.          FOR j = 1 TO w DO
5.              C[i,j] = 0
6.              FOR k = 1 TO q DO
7.                  C[i,j] = C[i,j] + A[i,k]*B[k,j]
8.      RETURN c
```

Esiste un modo per definire il numero di prodotti scalari (a cui ci riferiremo col termine "complessità") compiuti da questo algoritmo, ed è alla base di questo problema di ottimizzazione.

8.4.2 La complessità del prodotto tra matrici

Supponendo di voler moltiplicare una sequenza di tre matrici

$$A_1 \cdot A_2 \cdot A_3$$

Con:
$A_1 : p \times q$
$A_2 : q \times w$
$A_3 : w \times r$
Il numero di operazioni scalari di $A_1 \cdot A_2$ sarà uguale a $p \cdot q \cdot w$. Il risultato di $A_1 \cdot A_2$ sarà una matrice di dimensioni $p \times w$.
La complessità totale del prodotto $A_1 A_2 A_3$ sarà uguale alla somma della complessità tra i vari prodotti riga per colonna, quindi, la complessità di $(A_1 \cdot A_2) \cdot A_3$ sarà uguale a

$$p \cdot q \cdot w + p \cdot w \cdot r$$

Proviamo ad associare diversamente il prodotto, facendo $A_1 \cdot (A_2 \cdot A_3)$.
Otterremo:

$$q \cdot w \cdot r + p \cdot q \cdot r$$

Esempi con valori effettivi

$$A_1 \cdot A_2 \cdot A_3$$

Con:
$A_1 : 10 \times 100$
$A_2 : 100 \times 5$
$A_3 : 5 \times 50$
La complessità totale sarà uguale alla somma della complessità tra i vari prodotti riga per colonna, quindi, la complessità di $(A_1 \cdot A_2) \cdot A_3$ sarà uguale a

$$10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 5000 + 2500 = 7500$$

Proviamo ad associare diversamente il prodotto, facendo $A_1 \cdot (A_2 \cdot A_3)$.
otterremo:

$$100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 25000 + 50000 = 75000$$

Osserviamo quinid che "associazioni" differenti, daranno complessità differenti, e non di poco! La scelta dell'ordine con cui andremo a effettuare le moltiplicazioni tra matrici, è fondamentale per ridurre al minimo il tempo computazionale. Detto ciò, questo sarà anche il nostro problema di ottimizzazione. Verifichiamone prima le proprietà.

Parentesizzazione

Scriviamo la definizione di parentesizzazione delle matrici.

$$S = \begin{cases} A \\ (S; S_2) \end{cases}$$

S si dirà espressione completamente parentesizzata se vale una delle due seguenti condizioni:

- S è una singola matrice
- S è della forma (S_1, S_2), dove S_1 e S_2 sono espressioni completamente parentesizzate

Ed è in questa forma che si presenteranno le soluzioni del nostro problema di ottimizzazione, ovvero sotto forma di parentesizzazioni.
Una sequenza può essere denotata come $(A_i, \ldots, A_j)$. Individuato un valore k, la sequenza può essere suddivisa in sottosequenze da $(A_i, \ldots, A_k)$ e $(A_{k+1}, \ldots, A_j)$. In questo modo verifichiamo che il problema è un problema ricorsivo. Verifichiamo ora l'altra proprietà dei problemi di ottimizzazione.

Sottostruttura ottima?

Con m funzione costo, la soluzione sarà data da:

$$m(i,j) = m(i,k) + m(k+1,j) + numero\ operazioni\ scalari\ (\#ms)$$

È facile dimostrare per assurdo la sottostruttura ottima.
Supponiamo infatti che esista una parentesizzazione ottima della sequenza

$$A_i \ldots A_j \quad con\ i < j$$

Supponiamo per assurdo che questa parentesizzazione suddivida la sequenza in due sotto sequenze:

$$A_i A_{i+1} \ldots A_k \quad e \quad A_{k+1} A_{k+2} \ldots A_j$$

Supponiamo adesso che la prima parentesizzazione non sia ottima. Vorrà dire che esiste un'altra parentesizzazione di $A_i \ldots A_k$ ottima, ma non contenuta nella soluzione che abbiamo ipotizzato come ottima di $A_i \ldots A_j$. Questa sarà una contraddizione, in quanto implicherebbe l'esistenza di una soluzione migliore di quella ottima. Ne consegue quindi che la soluzione ottima di $A_i \ldots A_j$ contiene già (e deve contenere) la soluzione ottima di $A_i A_k$. La stessa dimostrazione vale considerando una soluzione non ottima a $A_{k+1} A_j$.

8.4.3 Programmazione dinamica - Prodotto tra matrici

Definizione ricorsiva:

$$m(i,j) = \begin{cases} \emptyset \;\; se\ i = j; \\ m[i,k] + m[k+1,j] + \#ms \end{cases}$$

Il numero di modi in cui è possibile suddividere una sequenza in questo caso, sarà uguale alla lunghezza della sequenza - 1. Osserviamo infatti che:

$$|A_i ... A_i| = 1 \;\; k \in \emptyset$$

$$|A_i ... A_{i+1}| = 1 \;\; k \in \{i\}$$

$$|A_i ... A_{i+2}| = 3 \;\; k \in \{i, i+1\}$$

Con $1 \leq i \leq k < j \leq n$ Ora, andiamo a scrivere una tabella contenente le dimensioni delle righe delle matrici.

numero matrice A		0	1	2	3	4	...	n
dimensione riga	p_0	p_1	p_2	p_3	p_4	...	p_n	

$$m(i,j) = \begin{cases} \emptyset \;\; se\ i = j; \\ \min\{m[i,k] + m[k+1,j] + P_{i-1}P_kP_j\} : i \leq k < j \end{cases}$$

Implementeremo, piuttosto che un array dei risultati, una matrice m dei risultati, di n righe e colonne. Denoteremo con i le righe e j le colonne.

Implementazione procedura

```
1.  MCO(P,n)
2.      m = NEW MATRIX(n,n)
3.      FOR i = 1 TO n DO
4.          m[i] = NaN
5.      FOR l = 2 TO n DO
6.          FOR i = 1 TO n-l+1 DO
7.              j = i + l - 1
8.              m[i,j] = +infinito
9.              FOR k = i TO j-1 DO
10.               IF m[i,k] + m[k+1,j] + P[i-1]*P[k]*P[j]<m[i,j] THEN
11.                   m[i,j] = m[i,k] + m[k+1,j] + P[i-1]*P[k]*P[j]
12. RETURN m[1,n]
```

Procedura per stampare soluzione

```
1.  PCO(i,j,S)
2.      IF(i == j) THEN
3.          PRINT "A"i
4.      ELSE
5.          k = s[i,j]
6.          PRINT"("
7.          PCO(i,k,S)
8.          PCO(k+1,j,S)
9.          PRINT")"
```

Soluzione con matrice dei risultati

```
1.  MCO(P,n)
2.      m = NEW MATRIX(n,n)
3.      S = NEW MATRIX(n,n)
4.      FOR i = 1 TO n DO
5.          m[i] = NaN
6.      FOR l = 2 TO n DO
7.          FOR i = 1 TO n-l+1 DO
8.              j = i + l - 1
9.              m[i,j] = +infinito
10.              FOR k = i TO j-1 DO
11.               IF m[i,k] + m[k+1,j] + P[i-1]*P[k]*P[j]<m[i,j] THEN
12.                  m[i,j] = m[i,k] + m[k+1,j] + P[i-1]*P[k]*P[j]
13.                  S[i,j] = k
14. RETURN m[1,n]
```

Capitolo 9

Programmazione greedy

9.1 Introduzione alla strategia greedy

Un'altra strategia che possiamo sfruttare per risolvere problemi di ottimizzazione, è la strategia greedy, in italiano strategia avida, golosa.
È una strategia molto potente, ma i problemi risolvibili con la strategia greedy sono più specifici di quelli risolvibili con la programmazione dinamica: i primi, infatti, devono godere di una determinata proprietà, chiamata appunto "proprietà di scelta greedy".

9.1.1 Cosa si intende per scelta greedy?

La scelta greedy è una scelta localmente ottima.
Cambia da problema a problema e dopo un numero finito di iterazioni, deve dare come risultato una soluzione ottima.
Un algoritmo greedy farà sempre una scelta localmente ottima, senza tornare sui suoi passi. Anche per questo, gli algoritmi greedy tendono ad essere più veloci di quelli basati sulla programmazione dinamica.

La proprietà di scelta greedy

Rispetto alla programmazione dinamica, la strategia greedy richiede una condizione in più. Oltre infatti il numero polinomiale di sottoproblemi e la sottostruttura ottima, il problema dovrà rispettare la proprietà di scelta greedy:
deve essere possibile creare una soluzione ottima, facendo la scelta greedy ad ogni passaggio.
Per dimostrare questa proprietà, solitamente, andremo a prendere un'ipotetica soluzione ottima, per poi sostituirne una scelta con la scelta greedy. Questo ci permetterà di dimostrare che la scelta greedy costruisce una soluzione ottima.

9.2 Problema dello zaino - esempio

Un valido esempio di scelta greedy è quella che possiamo utilizzare per risolvere il seguente problema dello zaino.

9.2.1 Problema dello zaino

In questo specifico problema dello zaino, poniamo il valore di ciascun oggetto $= 1$.

Variabili

$k =$ capienza (peso massimo) dello zaino.
$A = \{a_1, a_2, ..., a_n\}$ oggetti.
$P(a_i) =$ peso dell'oggetto a_i.
$V(a_i) =$ valore dell'oggetto a_i, $V(a)$ $\forall a \in A$.

Goal

Massimizzare $\sum_{a \in S} V(a)$ con il seguente vincolo $\sum_{a \in S} P(a) \leq k$.

9.2.2 Riformulazione del problema

Riformuliamo il problema ponendo il $v(a) = 1 \ \ \forall a \in A$.

$$\sum_{a \in S} v(a) = \sum_{a \in S} 1 = |S|$$

Banalmente, ciò che dobbiamo massimizzare diventerà il numero di oggetti. Un'ottima strategia (greedy!) può essere quella di scegliere ogni volta l'oggetto di peso minimo, in quanto ogni oggetto ha lo stesso valore.
Quella che facciamo a ogni passo, è una scelta localmente ottima!

Dimostrazione scelta Greedy

$$a_w : P(a_w) = \min\{Peso(a_j) : a_j \in A\}$$

Ovvero a_w scelta greedy. $S \subseteq A$ è una soluzione ottima. Allora $a_w \in S$. Ipotizziamo per assurdo che $a_w \notin S$.

$$a_\phi \rightarrow P(a_\phi) = min\{P(a_j) : a_j \in S\}$$

Ovvero a_ϕ elemento meno pesante della soluzione ottima.
Allora consideriamo $S - \{a_\phi\}$ e considero

$$S' = (S - \{a_\phi\}) \cup \{a_w\}$$

Per ipotesi, S è una soluzione ottima, per cui

$$\sum_{a \in S} P(a) \leq k$$

Possiamo dire che il peso complessivo di S' è uguale al peso di S meno il peso di a_ϕ più il peso di a_w.

$$\sum_{a \in S'} P(a) = \sum_{a \in S} P(a) - P(a_\phi) + P(a_w)$$

E concludiamo che

$$\sum_{a \in S'} P(a) \leq \sum_{a \in S} P(a) \leq k$$

E quindi S' è una soluzione ottima, e contiene a_w, quindi la tesi è dimostrata!

9.3 Problema della compressione - Huffman

L'algoritmo di Huffman stabilisce il numero minimo di bit richiesti per rappresentare un informazione.

9.3.1 Definizioni

Alfabeto $\Sigma : a,\ b,\ c,\ d,\ e,\ f$;
Testo $T : aabcaedfccaabafaeafd$ (60 bit).

Σ	codifica	occorrenze
a	000	8
b	001	2
c	010	3
d	011	3
e	100	2
f	101	3

Si basa sul presupposto che, all'interno di una stringa di simboli dell'alfabeto Σ, esistono caratteri con occorrenze maggiori o minori. Si associano dei codici, diversi dalla codifica in binario, tali che stringhe più complesse sono associate a caratteri con meno occorrenze.

9.3.2 Come scegliere i codici?

Scegliendo i codici senza alcun criterio specifico, può creare ambiguità nella lettura della stringa. Per fare ciò, bisogna scegliere dei codici consoni, tale che il prefisso di ciascun codice sia univoco. Moltiplicando il numero di occorrenze

Σ	codifica	occorrenze	codice
a	000	8	00
b	001	2	0110
c	010	3	111
d	011	3	0101
e	100	2	1101
f	101	3	101

di ciascun carattere con il numero di bit richiesto per rappresentarlo, otteniamo il nuovo numero di bit usato. In questo caso 58 bit, abbiamo risparmiato due bit.

9.3.3 Algoritmo Greedy - Codifica Huffman

Rappresentando i prefissi come un albero, il nostro obiettivo sarà portare quanto più in basso, nell'albero, i caratteri con meno occorrenze.

1. Sostituiamo due caratteri di occorrenza minima con un unico carattere nuovo, che avrà occorrenze pari alla soma delle occorrenze dei caratteri di partenza.

2. Rappresentiamo il nuovo carattere come un nodo che ha come figli i precedenti caratteri, e frequenza pari alla somma delle frequenze dei figli.

3. Effettuo iterativamente l'intera procedura, fino a quando non avrò costruito un albero unico.

9.3.4 Pseudo-Codice

Sigma alfabeto, f frequenza di ogni elemento

```
1. HUFFMAN(Sigma, f)
2.  N = {node (f(c), c) | c in Sigma} //set di coppie lettera frequenza
3.  n = |Sigma|                       //n = size di sigma
4.  Q = BUILD-MIN-HEAP(N)             //costruisce min-heap su N
5.  FOR i = 1 TO n-1 DO
6.      a = EXTRACT-MIN(Q);       //unisce a due a due i nodi
7.      b = EXTRACT-MIN(Q);       //di frequenza minore a,b in un nuovo
8.      c = NEW NODE(f(a) + f(b)) //nodo di frequenza somma di f(a)+f(b)
9.      LEFT(c) = a; RIGHT(c) = b;//e con figli a e b
10.     INSERT(Q,c)               //inserisce nella coda con priorità
11. RETURN EXTRACT-MIN(Q)         //il nodo. La procedura all'n-1 esima
```

Riga 2 $O(n)$, Riga 4 $O(n)$, $n-1$ volte riga 6,7 e 10, tutte e tre $O(\log n)$. In via definitiva, l'algoritmo di Huffman ha complessità $O(n + n \log n) = O(n \log n0)$.

9.3.5 Codice aggiuntivo

Ottenere codice di un nodo (carattere) x

Versione iterativa:

```
1. GET-CODE(x)
2.  S = new stack()
3.  p = parent(x)
4.  WHILE p != NULL
5.      IF x = left(p) THEN
6.          PUSH(S,0)
7.      ELSE PUSH(S,1)
8.      x = p
9.      p = parent(x)
```

Versione ricorsiva:

```
1. GET-CODE(x)
2.  if x=NULL RETURN
3.  p = parent(x)
4.  GET-CODE(P)
5.      IF x = left(p) THEN
6.          PRINT(0)
7.      ELSE PRINT(1)
```

Per ottenere la codifica di un carattere, leggiamo dal basso verso l'alto. La codifica sarà tuttavia da leggere dall'alto verso il basso, e quindi in una versione iterativa sarà necessario uno stack.
Al contrario, per ottenere il simbolo a partire da un codice (0 e 1) andremo dalla radice e sceglieremo di andare a sinistra quando incontriamo 0, altrimenti a destra.

Capitolo 10

Algoritmi sui Grafi

10.1 Introduzione agli algoritmi sui grafi

Dedicheremo questo capitolo allo studio di algoritmi sui grafi. Riprendiamo la definizione formale di grafo e come è possibile rappresentarli in memoria.

10.1.1 Grafi orientati e non orientati

$$G = (V, E)$$

In un grafo non orientato, se (i, j) è una relazione, sarà uguale a (j, i).
In un grafo orientato, (i, j) non implica (j, i).

10.1.2 Come rappresentare un grafo

Sono due i metodi principali che vengono usati per rappresentare grafi.

Liste di Adiacenza

ovvero un vettore di lunghezza n, contenente puntatori a liste concatenate. Nell'array, è dedicato un indice a ciascun nodo del grafo. La lista associata a ciascun nodo, conterrà tutti i nodi ad esso adiacente.

Matrici

Una matrice $n \times n$ ci permetterà di rappresentare un grafo con n nodi. Nella matrice in questione, se i nodi l ed m sono adiacenti, allora $a_{l,m} = 1$. In un grafo non orientato, la matrice di adiacenza sarà sempre simmetrica, e quindi $a_{m,l} = 1 \Leftrightarrow a_{l,m} = 1$.

10.2 Gli algoritmi di visita sui grafi

L'obiettivo è quello di visitare tutti i nodi all'interno di un grafo, senza ridondanze. Determinati algoritmi ci permettteranno di individuare caratteristiche all'interno del grafo.

10.2.1 BFS

Breadth-First Search, o visita in ampiezza.
Scelto un nodo sorgente, ci permette di trovare i cammini minimi che partono da quel nodo. I nodi vengono visitati in ordine di distanza dal nodo S, che sarà di distanza 0 da se stesso. Verranno poi visitati i nodi di distanza 1, poi 2, ..., poi il $k-1$esimo e il k-esimo.
L'algoritmo BFS costruirà un albero, con S radice. I nodi di distanza k all'interno del grafo saranno situati al k-esimo livello nel grafo. L'albero va a esplicitare l'ordine di visita dei nodi.

10.2.2 Implementazione della BFS

L'algoritmo in questione sfrutta una coda (queue), e un meccanismo di colori che andremo ad associare ai nodi.

Colori

$$color(v) = \begin{cases} Bianco \Rightarrow Mai\ visitato \\ Gray \Rightarrow\ \grave{E}\ in\ coda \\ Nero \Rightarrow\ scito\ dalla\ coda \end{cases}$$

Procedimento base

1. (Solo alla prima iterazione) Inserisco in coda il nodo sorgente S, che diventerà grigio.
2. Dequeue, estraggo dalla testa (colorando di nero il nodo estratto) e inserisco in coda tutti i nodi adiacenti al nodo estratto, i quali diventeranno grigi.
3. Elimino la testa, colorandola di nero
4. Ripeto i passaggi 2 e 3 fino a quando tutti i nodi non saranno segnati di colore nero.

L'albero BFS ci offrirà un albero dei cammini minimi, una sorta di mappa topologica del grafo. La mappa in questione non includerà i nodi non raggiungibili da S, i quali non saranno mai inseriti in coda e colorati di grigio.

10.2.3 Pseudo-codice procedura BFS

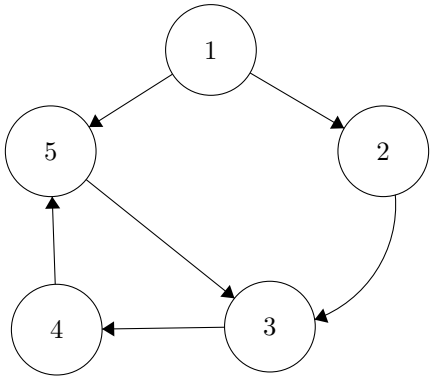
```
1. BFS(V,S)
2.  FOR EACH v IN V DO
3.      color[v] = Bianco
4.      d[v] = +infinito //IPOTESI nessun nodo raggiungibile
5.      pi[v] = NULL //pi[v] è il predecessore di v
6.  d[s] = 0
7.  Q = {S} // Q è LA CODA INIZIALIZZATA CON S
8.  color[S] = Grigio
9.      //FINO A QUI ABBIAMO LA FASE DI INIZIALIZZAZIONE
10. WHILE Q != {} DO //{} insieme vuoto
11.     u = DEQUEUE(Q)
12.     FOR EACH v in ADJ(u) DO
13.     IF color[v] == Bianco THEN
14.         d[v] = d[u]+1
15.         pi[v] = u
16.         color[v] = Grigio
17.         ENQUEUE(Q,v)
```

10.2.4 Complessità dell'algoritmo BFS

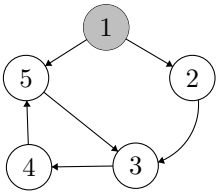
Riga 10 n volte, dove n è il numero di nodi. Riga 12 n volte. Su due piedi potremmo dire che la complessità è $O(n^2)$, ed è corretto, ma non è un limite stretto.
Infatti, mentre il ciclo a riga 10 verrà computata un numero di volte pari al numero dei nodi, la riga 12 verrà computata un numero di volte pari a $|E| = m$, ma solo se l'algoritmo è implementato con liste di adiacenza, altrimenti sarà $O(n)$.

10.2.5 Esempio di BFS applicata ad un grafo

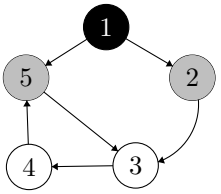
$V = \{1, 2, 3, 4, 5, 6, 7\}$;
$E = \{(1, 6), (1, 2), (2, 4), (2, 5), (4, 7), (3, 2)\}$
$s = 1$



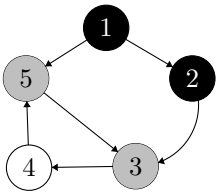
1. $Queue : [Tail*]\ \ 1\ \ [Head*]$



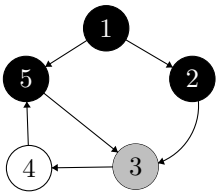
2. $Queue : 5 \rightarrow 2$



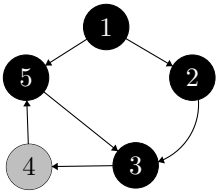
3. $Queue : 3 \rightarrow 5$



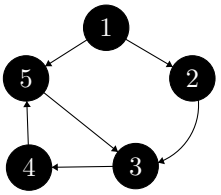
4. *Queue* : 3



5. *Queue* : 4



6. *Queue* :



10.2.6 Calcolo delle distanze

Avendo il vettore $\pi[v]$, è possibile calcolare la distanza di ciascun nodo dal nodo s, radice dell'albero BST.

Procedura iterativa

```
1. Distance(pi, v)
2.  d = 0
3.  WHILE pi[v] != NULL DO
4.      d = d+1
5.      v = pi[v]
6.  RETURN d
```

Procedura ricorsiva

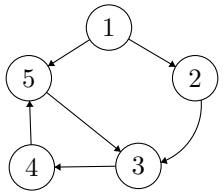
```
1. Distance(pi, v)
2.  IF pi[v] = NULL RETURN 0
3.  RETURN 1 + Distance(pi,pi[v])
```

10.3 Struttura topologica di un grafo

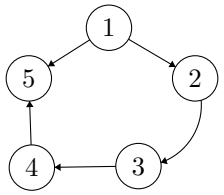
L'ordinamento topologico di un grafo è definito come un ordinamento lineare dei suoi nodi. Essi sono disposti in una sequenza che rispetta la seguente regola: ogni nodo viene prima dei nodi connessi ad esso tramite archi uscenti.
Questo implica che non esiste un ordinamento topologico di un grafo che presenta un ciclo.

10.3.1 Ordinamento topologico

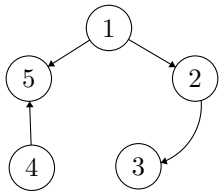
Il seguente grafo non ha un ordinamento topologico a causa del ciclo $5 \to 3 \to 4$.



Eliminiamo il ciclo, ottenendo un grafo aciclico.



Ordinamento: $1, 2, 3, 4, 5$.



Ordinamento: $4, 1, 5, 2, 3$ oppure $1, 4, 2, 3, 5$, ma anche altri.

Alcune osservazioni sugli ordinamenti topologici

Il numero minimo di ordinamenti topologici in un grafo è 0, e si ha quando il grafo è vuoto, o se presenta un ciclo.
Il numero massimo di ordinamenti topologici è uguale a $n!$, in un grafo con n nodi, e si ha quando il grafo non presenta archi. Il seguente grafo ha ben 5040 ordinamenti topologici distinti!

10.4 Ricerca in profondità - DFS

10.4.1 Introduzione alla DFS

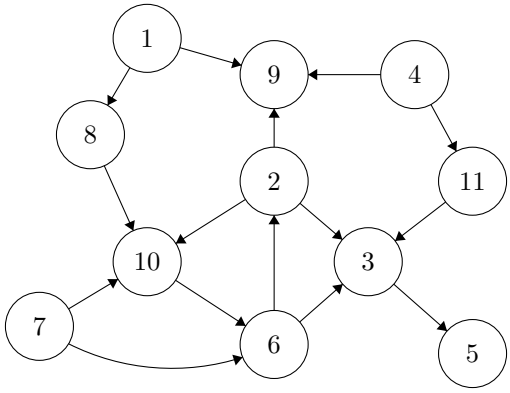
La procedura DFS, ovvero Depth-First Search, effettua una ricerca in profondità all'interno di un grafo.
Il principio è semplice: visitato un nodo, si visiterà un suo adiacente a scelta. Arrivati ad un vicolo cieco (ovvero quando non ci sono nodi adiacenti, o sono esclusivamente nodi già visitati), si torna indietro nel percorso e si visitano gli altri non visitati in precedenza, facendo ricominciare la procedura regolarmente. Detto ciò, se non tutti i nodi del grafo sono stati visitati partendo da un singolo nodo, la DFS sceglierà un nuovo nodo sorgente (mai visitato) da cui continuare le visite. La procedura continua fino a quando tutti i nodi non saranno stati visitati.

10.4.2 Il filo di Arianna?

Il principio di funzionamento della DFS riprende un po' il mito di Arianna e il Minotauro: l'algoritmo visiterà i cammini andando sempre più in profondità, marcando il suo cammino con un filo.
Questo filo è in realtà implementato con un meccanismo di colorazione, che segue le seguenti regole:

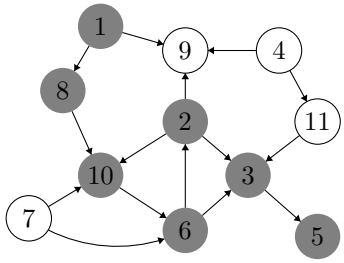
$$nodo.color = \begin{cases} bianco \Leftrightarrow Nodo\ mai\ visitato \\ grigio \Leftrightarrow Visita\ iniziata\ ma\ non\ conclusa \\ nero \Leftrightarrow Visita\ conclusa \end{cases}$$

Tutti i nodi del grafo sono inizializzati come bianchi. Quando un nodo viene visitato per la prima volta, diventa grigio. Quando da un nodo grigio non è possibile spostarsi verso alcun nodo bianco, si ritorna indietro nel percorso colorando il primo di nero. Quando dalla visita partita dal nodo sorgente s non è più possibile visitare altri nodi, l'algoritmo DFS crea degli altri percorsi partendo da i nodi rimasti bianchi, sempre seguendo l'ordine stabilito (che all'interno di questi esempi sarà un semplice ordine lessicografico). L'output della DFS è una foresta di alberi DFS, in quanto non tutti i percorsi sono collegati tra loro.

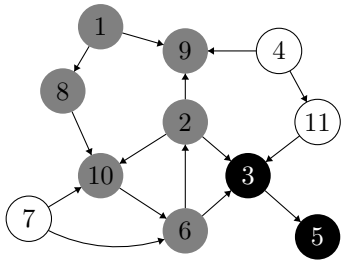


Esempio di DFS sul grafo in questione:

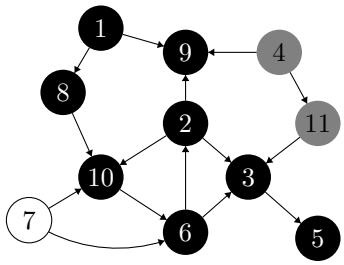
1. Nodi visitati: $1 \to 8 \to 10 \to 6 \to 2 \to 3 \to 5$



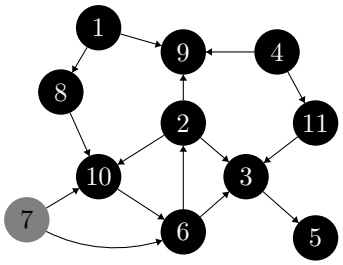
2. Nodi colorati di nero: $5 \to 3$; Nodi visitati: $2 \to 9$



3. Nodi colorati di nero: $9 \to 2 \to 6 \to 10 \to 8 \to 1$; Nodi visitati: $4 \to 11$



4. Nodi colorati di nero: $11 \rightarrow 4$; Nodi visitati: 7



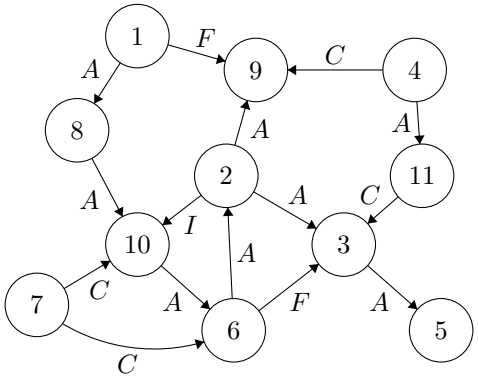
Nota bene!

Tratto dal libro di testo:
Potrebbe sembrare arbitrario il fatto che una visita in ampiezza sia limitata a una sola sorgente, mentre una visita in profondità può cercare da più sorgenti. Anche se una visita in ampiezza potrebbe concettualmente partire da più sorgenti, e una visita in profondità potrebbe essere limitata ad una sorgente, il nostro approccio riflette il modo in cui i risultati di queste visite vengono comunemente usati.
Insomma, scegliamo di definire BFS come ricerca a unica sorgente e DFS come ricerca a più sorgenti, esclusivamente per le applicazioni di ciascuno degli algoritmi.

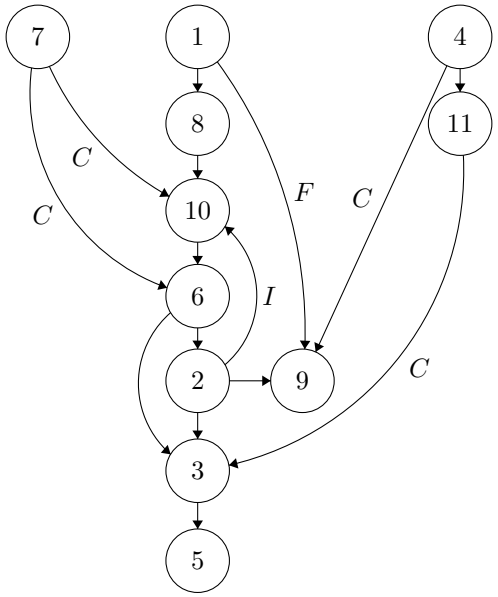
10.4.3 Alberi DFS

All'interno degli alberi DFS, conosciamo degli archi specifici:

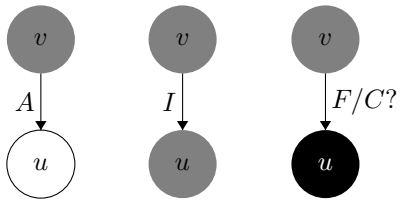
- **Archi d'albero. (A o T)**
- **Archi in avanti. (F)**
  Sono archi che collegano un nodo di un albero con un suo discendente indiretto, dall'alto verso il basso.
- **Archi di attraversamento. (C)**
  Uniscono rami differenti dello stesso albero o di alberi diversi
- **Arco all'indietro. (I o B)**
  Unisce un nodo con un suo predecessore



Ecco i tre alberi DFS, collegati tra loro da nodi di attraversamento C.



10.4.4 Riconoscere i tipi di arco nel grafo



Supponendo di essere in un nodo grigio, all'interno di un grafo, consideriamo i suoi nodi adiacenti:

- Se sono bianchi, allora sono archi d'albero (A).
- Se sono grigi, allora l'arco sarà all'indietro (I o B).
- Se sono neri, allora l'arco può essere in avanti (F) o di attraversamento (C). Se il tempo di fine di u è minore di quello di v, allora sarà un arco in avanti. Altrimenti sarà un arco di attraversamento.

Tempo

Per disambiguare gli archi in avanti e quelli di attraversamento, usiamo un parametro t, chiamato tempo. Quando viene colorato di grigio o di nero un nodo, incrementiamo un parametro tempo. Il parametro tempo quando un nodo n viene colorato di grigio è detto "scoperta di n", e si indica con $n.d$ ($n.discovery$). Quando viene colorato di nero, si dirà "fine della visita del nodo n", e si indica con $n.f$.
Detto ciò, se $u.f < v.f$, allora sarà un arco in avanti.

10.4.5 Procedura algoritmo DFS

La seguente procedura effettua la scelta del/dei nodi sorgente per le visite.

```
1. DFS(G)
2.  FOR EACH v IN V DO
3.      color[v] = bianco
4.      pi[v] = NULL
5.      d[v] = f[v] = +infinito
6.  FOR EACH v in V DO
7.      IF color[v] = bianco THEN
8.          DFS-VISIT(v)
```

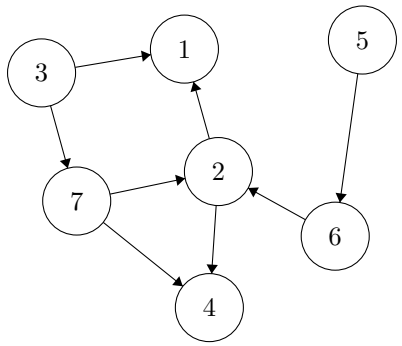
La seguente si occupa delle singole visite vere e proprie.

```
1. DFS-VISIT(v)
2.  color[v] = grigio
3.  d[v] = t
4.  t = t + 1
5.  FOR EACH u IN adj[u]
6.      IF color[u] = B THEN
7.          pi[u] = v
8.          DFS-VISIT(u)
9.  color[v] = nero
10. F[v] = t
11. t = t + 1
```

$O(|V| + |E|)$

10.4.6 Ordinamento topologico

Introduciamo un grafo aciclico.



Denotando i tempi di inizio e fine di ogni visita, visita effettuata secondo la procedura DFS, è possibile verificare un ordinamento topologico corretto.
Infatti, se $f[v] < f[u]$, allora u è propedeutica a v, in quanto la fine della visita su un nodo avviene solo quando sono finite le visite dei suoi successori.
Per trovare un ordinamento topologico, basta inserire ogni volta i nodi in uno stack, con il corrispettivo tempo di fine f quando la visita viene conclusa. Otterremo un ordinamento topologico estraendo mano a mano gli elementi dello stack.

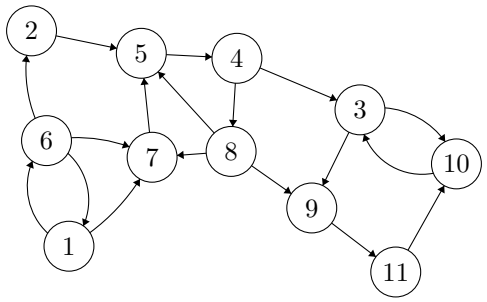
Relazione ordinamento topologico - aciclicità

Il grafo ha un ciclo $\Leftrightarrow$ *Il grafo non ha un ordinamento topologico*

Equivalentemente

Il grafo è aciclico $\Leftrightarrow$ *Il grafo ha un ordinamento topologico*

10.4.7 Calcolo delle componenti fortemente connesse



Fare una visita DFS all'interno di un grafo non ci permette sempre di individuare le singole componenti fortemente connesse come alberi separati. Per fare ciò, basterà:

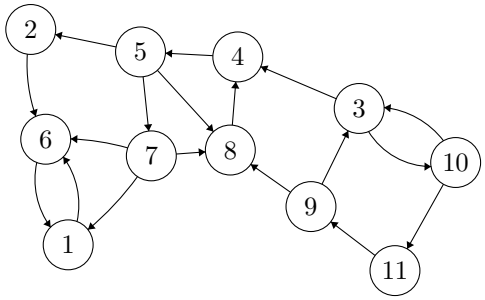
1. Chiama una prima volta DFS per calcolare i tempi di fine $u.f$ per ogni nodo u.
2. Crea G^T, ovvero il grafo trasposto, ottenuto invertendo il verso di ogni singolo arco.
3. Effettua DFS su G^T, ma scegli come sorgenti i nodi in ordine decrescente rispetto ai tempi $u.f$
4. Ciascuno degli alberi ottenuti sarà una componente connessa distinta.

Esempio di calcolo componenti fortemente connesse

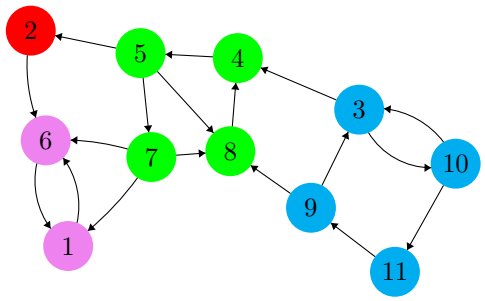
1. DFS sul grafo G. Otteniamo i tempi di ciascun nodo:

Nodo	1	2	3	4	5	6	7	8	9	10	11
Nodo.d	1	3	6	5	4	2	15	14	7	9	8
Nodo.f	22	20	13	18	19	21	16	17	12	10	11

2. Creiamo il grafo trasposto. Intuitivamente, se rappresentato con matrici, basterà trasporre la matrice. Per le liste, il discorso è leggermente più complesso, ma al momento non preoccupiamoci di questo.
   G^T :



3. Effettuaimo le visite DFS usando il tempo di fine delle visite in ordine decrescente.
   Prima visita: $1 \to 6$
   Seconda visita: 2 Terza visita: $5 \to 7 \to 8 \to 4$
   Quarta visita: $3 \to 10 \to 11 \to 9$



L'output della nostra DFS su G^T, eseguita in ordine decrescente rispetto ai tempi di fine visita di G, sarà una foresta di alberi costituita da 4 alberi separati, uno per ciascuna componente connessa.

Capitolo 11

Varie ed eventuali

11.1 "Varie ed eventuali" cosa?

Carote? Patate?
Ciao, sono Dam :). In questo capitolo inserirò domande note e importanti per un qualsiasi esame di algoritmi e strutture dati. Ho deciso di isolare queste domande per velocizzare l'aggiunta all'interno di questi appunti di nuove nozioni.

11.2 Code con priorità

"Parlami delle code con priorità" - non è una domanda da sottovalutare!
Col termine "coda con piorità" non indichiamo un'implementazione specifica, tanto più una struttura dati generica che presenta un meccanismo di priorità delle chiavi e che contempla le seguenti operazioni:

- `ExtractMin() - ExtractMax()`
  Estrazione dell'elemento dalla priorità massima.
- `FindMin() - FindMax()`
  Funzione che ritorni (senza estrarre) l'elemento dalla priorità massima.
- `Insert()`.
  Funzione che inserisce un nuovo elemento nella coda con priorità.

La coda con priorità induce ad una struttura di tipo FIFO, ma non in maniera assoluta.

Perché preferire l'heap all'rb-tree per una coda con priorità?

- `ExtractMin() - ExtractMax()`.
  Albero rosso nero: $\Theta(\log n)$
  Heap: $O(\log n)$
- `FindMin() - FindMax()`.
  Albero rosso nero: $\Theta(\log n)$
  Heap: $O(1)$
- `Insert()`.
  Albero rosso nero: $\Theta(\log n)$
  Heap: $O(\log n)$

È fondamentale portare la nostra attenzione sui motivi dietro gli O-grande e i Θ dell'estrazione del minimo e dell'inserimento.
Un albero rosso-nero, per trovare l'elemento minimo o massimo, dovrà essere percorso in altezza nella sua interezza. Lo stesso vale per l'inserimento di una nuova chiave, per la quale si dovrà arrivare sempre al livello delle foglie.
Un min(max)-heap, impiega tempo costante per trovare il proprio elemento minimo(massimo). Esso si troverà in posizione $H[1]$. Verrà poi chiamata la funzione `Heapify()`, che aggiusterà le proprietà dell'heap.
La funzione `Heapify()` ha complessità $O(\log n)$, ma $\Omega(1)$, in quanto potrebbe non dover arrivare fino al livello delle foglie, o addirittura non dover risolvere alcuna violazione! Per l'inserimento, l'accesso al nuovo slot è costante e la correzione col ciclo for potrebbe interrompersi prima di $\log n$ passi.
In breve, l'heap ha un vantaggio asintotico non indifferente rispetto all'albero rosso nero su operazioni di estrazione e di inserimento.

11.3 Hash tables scansione lineare

11.3.1 Esercizio tipo - 1

Si consideri una tabella hash a indirizzamento aperto da $m = 11$ celle e cui configurazione dopo 8 inserimenti è `[ X - X X X - - X - - X ]`.
Si supponga che la funzione hash utilizzata per generare la sequenza di scansione lineare sia

$$h(k) = (h'(k) + 2i) \mod (m)$$

con $h'(k)$ una funzione di hashing ausiliaria che gode di hashing uniforme e indipendente. Si fornisca la probabilità che l'i-esima cella sia la prossima cella ad essere occupata dopo un'operazione di inserimento.

Soluzione

Teniamo a mente che $i = 2$. La configurazione è `[ X - X X X - - X - - X ]`.

1. $\frac{0}{11}$
2. $\frac{2}{11} : 11 \to 2$
3. $\frac{0}{11}$
4. $\frac{0}{11}$
5. $\frac{0}{11}$
6. $\frac{2}{11} : 4 \to 6$
7. $\frac{4}{11} : 1 \to 3 \to 5 \to 7$
8. $\frac{0}{11}$
9. $\frac{1}{11} : 9$
10. $\frac{2}{11} : 11 \to 2$
11. $\frac{0}{11}$