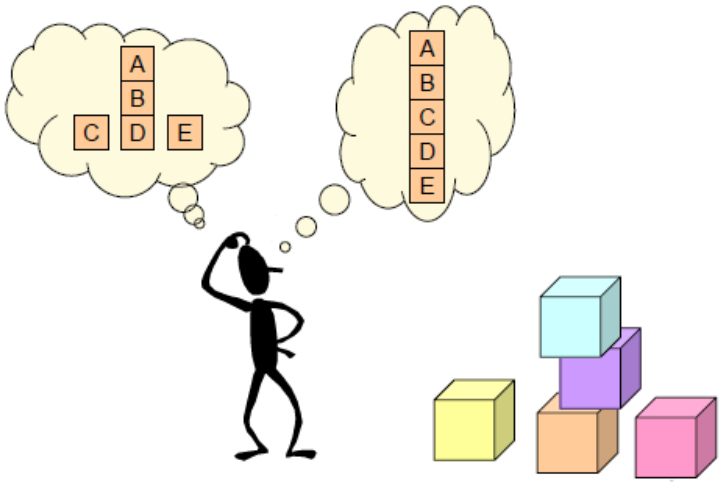
Algoritmi
e
Strutture Dati

v0.12.0



Luca De Franceschi Università degli studi di Padova

Diario delle modifiche

Autore	Versione	Data	Descrizione
Luca De Franceschi	0.12.0	17/06/2014	Inseriti esercizi su strutture dati per insiemi disgiunti
Luca De Franceschi	0.11.0	17/06/2014	Inserito capitolo strutture dati per insiemi disgiunti
Luca De Franceschi	0.10.0	16/06/2014	Inserita immagine di copertina
Luca De Franceschi	0.9.0	16/06/2014	Inserito capitolo dei B-alberi
Luca De Franceschi	0.8.0	16/06/2014	Inserito capitolo codici di Huffman
Luca De Franceschi	0.7.0	15/06/2014	Inserito capitolo hashing e risposte a domande degli appelli
Luca De Franceschi	0.6.0	14/06/2014	Inserito capitolo algoritmi golosi con esercizio del pulmino
Luca De Franceschi	0.5.0	13/06/2014	Inserito capitolo analisi ammortizzata con due esercizi, incrementata sezione metodo dell'integrale
Luca De Franceschi	0.4.0	11/06/2014	Inserito capitolo analisi complessità con metodo dell'integrale e metodo dell'esperto
Luca De Franceschi	0.3.0	11/06/2014	Inserita spiegazione metodo di sostituzione
Luca De Franceschi	0.2.0	11/06/2014	Inserita teoria su programmazione dinamica
Luca De Franceschi	0.1.0	11/06/2014	Creata struttura del documento

Indice

1 Stima della complessità di un algoritmo

Identifichiamo dei casi base, studiando la complessità degli algoritmi noti.

1. Le operazioni elementari, messe al di fuori dei cicli, e che riguardano l'uso di variabili hanno complessità costante c_i, aprossimabile a 0 nello studio della complessità asintotica;
2. Da *insertion-sort* si vede che un *for i = 2 to n* ha complessità pari a n. Constatiamo dunque che un ciclo for che va dall'indice 1 all'indice n avrà complessità $c_i(n+1)$;
3. Tutte le operazioni elementari che compaiono all'interno di un ciclo for di complessità $c_i(n+1)$ hanno complessità $c_i n$;
4. Per i cicli annidati in altri cicli a complessità è data da $c_i \sum_{j=x}^{n}(c_j)$, dove gli estremi della sommatoria sono gli estremi del del ciclo esterno.

Per valutare la complessità si scrive l'equazione $T(n)$ sommando tutte le complessità. Per studiare le sommatorie si utilizza il **metodo dell'integrale**.

1.1 Metodo dell'integrale

Se $f(x)$ è una funzione **non decrescente**:

$$\int_a^{b+1} f(x)\mathrm{d}x \leq \sum_{i=a}^{b} f(i) \leq \int_{a-1}^{b+1} f(x)\mathrm{d}x$$

Se $f(x)$ è una funzione **non decrescente**:

$$\int_a^{b+1} f(x)\mathrm{d}x \leq \sum_{i=a}^{b} f(i) \leq \int_{a-1}^{b} f(x)\mathrm{d}x$$

1.1.1 Integrali immediati

- $\int [f(x)]^\alpha f'(x)\mathrm{d}x = \frac{[f(x)]^{\alpha+1}}{(\alpha+1)} + C \qquad \alpha \neq 1$
- $\int \frac{f'(x)}{f(x)}\mathrm{d}x = \log|f(x)| + C$
- $\int f'(x)e^{f(x)}\mathrm{d}x = e^{f(x)} + C$
- $\int f'(x)a^{f(x)}\mathrm{d}x = a^{f(x)}\log_a e + C$

1.1.2 Integrazione per parti

Siano f, g, g' continue nell'intervallo $[a, b]$ e sia γ una primitiva di f, allora:

$$\int f(x)g(x)\mathrm{d}x = \gamma(x)g(x) - \int \gamma(x) + g'(x)\mathrm{d}x$$

1.1.3 Serie aritmetica e geometrica

- Serie aritmetica: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$;
- Serie geometrica: $\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}$ $q \neq 1$

1.2 Andamento asintotico

Una volta ottenuta una funzione che rappresenta la complessità dell'algoritmo ci può interessare prendere in esame l'andamento asintotico della medesima. Per farlo introduciamo le seguenti notazioni:

- **"O" grande**: date due funzioni $f(n)$ e $g(n)$ si dice che $f(n)$ è "O" grande di $g(n)$ se esiste un $c > 0$ e un h_0 tali che:

  $$f(n) \leq cg(n)$$

  per $n \geq h_0$ (**limite asintotico superiore**). In pratica l'ordine di crescita di $f(n)$ è non superiore a quello di $g(n)$;

- **"Ω" grande**: date $f(n)$ e $g(n)$ si dice che $f(n)$ è "Ω" grande di $g(n)$ se esiste una costante $c > 0$ e un h_0 tale che:

  $$f(n) \geq cg(n)$$

  per $n \geq h_0$ (**limite asintotico inferiore**). In pratica l'ordine di crescita di $f(n)$ è non inferiore a quello di $g(n)$;

- **"Θ" grande**: date $f(n)$ e $g(n)$ si dice che $f(n)$ è "Θ" grande di $g(n)$ se ci sono costanti positive c_1, c_2 e un h_0 tali che:

  $$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

  per $n \geq h_0$ (**limite asintotico stretto**). In pratica diciamo che se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$ allora è vero anche che $f(n) = \Theta(g(n))$

Di una funzione non ci interessa la sua forma ma il suo comportamento asintotico. Spesso è possibile determinare dei limiti asintotici calcolando il limite di un rapporto:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

In base al risultato di questo limite ho tre casi:

1. Ottengo un valore costante $k > 0$: in questo caso $f(n)$ è dello stesso ordine di $g(n)$, e dunque:

   $$\forall \epsilon > 0, \exists h_0 | h \geq h_0 : k - \epsilon \leq f(n)/g(n) \leq k + \epsilon$$

   ponendo:

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Dunque concludo dicendo che $f(n) = \Theta(g(n))$;

2. Il limite tende a ∞: $f(n) = \Omega(g(n))$;
3. Il limite tende a 0: $f(n) = O(g(n))$.

1.3 Metodo dell'esperto

Per risolvere le ricorrenze il primo metodo da utilizzare è il **metodo dell'esperto**. Se la ricorrenza è espressa nella forma:

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

e se $a \geq 1$ e $b < 1$ allora:

1. Tolgo eventuali arrotondamenti;
2. Calcolo $\log_b a$ e calcolo il limite: $\lim_{n\to\infty} \frac{f(n)}{n^{\log_b a}}$;

A questo punto, in base al valore del limite ho 3 possibili casi:

1.3.1 Caso 2

Se il limite è **finito** e diverso da zero:

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

1.3.2 Caso 1

Se il limite è **uguale a zero** devo trovare un valore $\epsilon > 0$ per il quale risulta finito il limite:

$$\lim_{n\to\infty} \frac{f(n)}{n^{\log_b a - \epsilon}} = k$$

Se lo trovo allora posso affermare che:

$$f(n) = O(n^{\log_b a - \epsilon}) \Rightarrow T(n) = \Theta(n^{\log_b a})$$

1.3.3 Caso 3

Se il limite è $\int$ allora devo trovare un $\epsilon > 0$ per il quale risulti:

$$\lim_{n\to\infty} \frac{f(n)}{n^{\log_b a + \epsilon}} \neq 0$$

Se lo trovo allora devo studiare l'equazione:

$$af(n/b) \leq k(f(n))$$

se trovo un $k < 1$ allora posso concludere che:

$$f(n) = \Omega(n^{\log_b a + \epsilon}) \Rightarrow T(n) = \Theta(f(n))$$

1.4 Metodo di sostituzione

Se non riesco ad applicare il metodo dell'esperto allora devo utilizzare il **metodo di sostituzione**.
Per capire il metodo di sostituzione proviamo a risolvere il seguente esercizio:

La ricorrenza $T(n) = 4T(n/2) + n^2 \log n$ si può risolvere con il metodo dell'esperto? Giustificare la risposta. Se la risposta è negativa usare il metodo di sostituzione per dimostrare che $T(n) = O(n^2 \log^2 n)$.

Anzitutto vediamo i dati a disposizione:

$$a = 4, b = 2$$
$$f(n) = n^2 \log n$$
$$g(n) = n^{\log_b a} = n^{\log_2 4} = n^2$$

Calcoliamo ora il limite:

$$\lim_{n \to +\infty} \frac{n^2 \log n}{n^2} = \infty$$

Da cui deduco che:

$$f(n) = \Omega(n^2)$$

Potrei dunque essere nel caso 3. Devo trovare un

$$\epsilon > 0$$

tale che:

$$\lim_{n \to +\infty} \frac{n^2 \log n}{n^{2+\epsilon}} \neq 0$$

Ma mi accorgo subito che la cosa è impossibile, in quanto il denominatore, incrementando l'esponente, crescerà molto più velocemente rispetto al numeratore, per cui avrò sempre un valore tendente allo zero. Da questa considerazione deduco che la ricorrenza **non è risolvibile con il metodo dell'esperto**.

Procedo dunque con la sostituzione. Proviamo $T(n) = O(n^2 \log^2 n)$.
Assumiamo che per un'opportuna costante $C > 1$ e $\forall x < n$ sia verificata la disuguaglianza $T(x) \leq C(x^2 \log^2 x)$ e dimostriamo che vale anche per n:

$$\begin{aligned} T(n) = 4T(n/2) + n^2 \log n &\leq 4C(n/2)^2 \log^2(n/2) + n^2 \log n \\ &= Cn^2(\log n - 1)^2 + n^2 \log n \\ &= Cn^2(\log^2 n - 2\log n + 1) + n^2 \log n \\ &= Cn^2 \log^2 n - 2Cn^2 \log n + Cn^2 \log n + Cn^2 + n^2 \log n \\ &= Cn^2 \log^2 n - (C-1)n^2 \log n - Cn^2(\log n - 1) \end{aligned}$$

Ora applico una **maggiorazione**:

$$\leq Cn^2 \log^2 n$$

Dunque ho dimostrato che: $T(n) = O(n^2 \log^2 n)$

2 Hashing

Molte operazioni richiedono un insieme dinamico che supporta soltanto le operazioni di dizionario *INSERT*, *SEARCH* e *DELETE*. Una **tavola hash** è una struttura dati efficace per implementare i **dizionari**. Sebbene la ricerca di un elemento in una tavola hash richieda, nel caso peggiore, lo stesso tempo $\Theta(n)$ richiesto per cercare un elemento in una lista concatenata, l'hashing si comporta molto bene nella pratica. Sotto ipotesi ragionevoli, il tempo medio per cercare un elemento in una tavola hash è $O(1)$. Una tavola hash è una generalizzazione della nozione più semplice di array ordinario. Quando il numero di **chiavi** effettivamente memorizzate è piccolo rispetto al numero totale di chiavi possibili, le tavole hash diventano una valida alternativa all'indirizzamento diretto di un array, in quanto una tavola hash tipicamente usa un array di dimensione proporzionale al numero di chiavi effettivamente memorizzate.

2.1 Tavole a indirizzamento diretto

L'indirizzamento diretto è una tecnica semplice che funziona bene quando l'universo U delle chiavi è ragionevolmente **piccolo**. Per rappresentare l'insieme dinamico, utilizziamo un array o **tavola a indirizzamento diretto**, che indicheremo con $T[0...m-1]$, dove ogni posizione o **cella** corrisponde a una chiave nell'universo U. La cella k punta ad un elemento dell'insieme con chiave k. Se l'insieme non contiene l'elemento con chiave k, allora $T[k] = nil$.

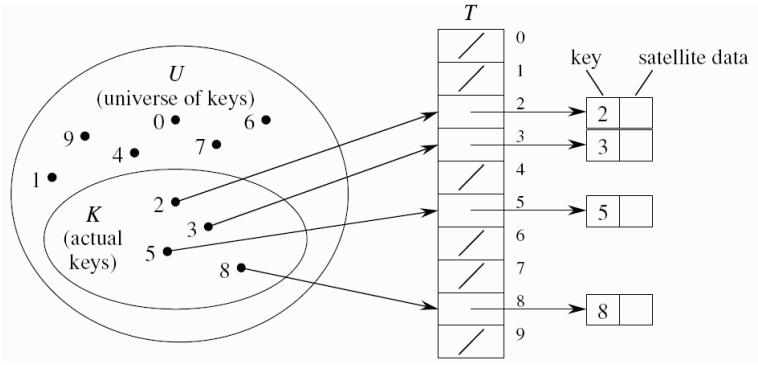


Figura 1: Tavola a indirizzamento diretto

```
DIRECT-ADDRESS-SEARCH (T, k)
       return T[k]
```

```
DIRECT-ADDRESS-INSERT (T,x)
       T[x.key] = x
```

```
DIRECT-ADDRESS-DELETE (T, x)
       T[x.key] = nil
```

Ciascuna di queste operazioni richiede tempo $O(1)$.

2.2 Tavole hash

La difficoltà dell'indirizzamento diretto è ovvia: se l'universo U delle chiavi è troppo grande, memorizzare una tavola T di dimensione $|U|$ può essere impraticabile. Inoltre, l'insieme K delle chiavi *effettivamente memorizzate* può essere così piccolo rispetto a U che la maggior parte dello spazio allocato per la tavola T sarebbe sprecato.
Con l'indirizzamento diretto un elemento con chiave k è memorizzato nella cella k. Con l'hashing, questo elemento è memorizzato nella cella $h(k)$: cioè utilizziamo una **funzione hash** h per calcolare la cella dalla chiave k. Qui h associa l'universo U delle chiavi alle celle di una **tavola hash** $T[0...m-1]$:

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

dove la dimensione m della tavola hash è generalmente molto più piccola di $|U|$. Diciamo che un elemento con chiave k viene mappato nella cella $h(k)$ o anche che $h(k)$ è il **valore hash** della chiave k.
Il compito della funzione hash è quello di ridurre l'intervallo degli indici e di conseguenza la dimensione dell'array. L'array ha dimensione m invece di $|U|$.

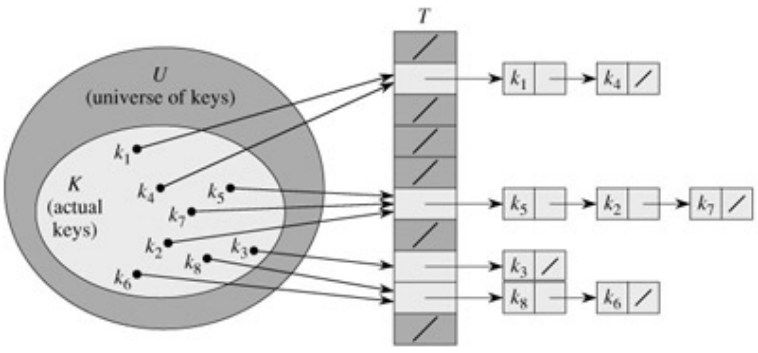


Figura 2: Utilizzo di una funzione hash

C'è un problema: due chiavi possono essere mappate nella stessa cella. Questo evento si chiama **collisione**. Fortunatamente ci sono tecniche efficaci per risolvere i conflitti creati dalle collisioni.

2.2.1 Risoluzione delle collisioni mediante concatenamento

Nel **concatenamento** poniamo tutti gli elementi che sono associati alla stessa cella in una lista concatenata. La cella contiene un puntatore alla testa della lista di tutti gli elementi memorizzati che vengono mappati in j; se non ce ne sono, la cella j contiene la costante nil. Le operazioni di dizionario sono facili da implementare:

```
CHAINED-HASH-INSERT (T, x)
       inserisce x in testa alla lista T[h(x.key)]
```

```
CHAINED-HASH-SEARCH (T, x)
       ricerca un elemento con chiave k nella lista T[h(k)]
```

```
CHAINED-HASH-DELETE (T, x)
       cancella x dalla lista T[h(x.key)]
```

Il tempo di esecuzione nel caso peggiore per l'inserimento è $O(1)$.

2.2.2 Analisi dell'hashing con concatenamento

Data una tavola hash T con m celle dove sono memorizzati n elementi, definiamo **fattore di carico** α della tavola T il rapporto n/m, ossia il numero medio di elementi memorizzati in una lista.
Il comportamento nel caso peggiore dell'hashing con concatenamento è pessimo: tutte le n chiavi sono associate alla stessa cella, creando una lista di lunghezza n. Il tempo di esecuzione della ricerca è quindi $\Theta(n)$, più il tempo per calcolare la funzione hash.
Le prestazioni dell'hashing nel caso medio dipendono dal modo in cui la funzione hash h distribuisce mediamente l'insieme delle chiavi da memorizzare tra le m celle. Per adesso supponiamo che qualsiasi elemento abbia la stessa probabilità di essere mandato in una qualsiasi delle m celle,

indipendentemente dalle celle in cui sono mandati gli altri elementi. Questa ipotesi è detta **hashing uniforme semplice**.

Teorema

In una tavola hash le cui collisioni sono risolte con il concatenamento, una ricerca senza successo richiede un tempo $\Theta(1+\alpha)$ nel caso medio, nell'ipotesi di hashing uniforme semplice.

Teorema

In una tavola hash le cui collisioni sono risolte con il concatenamento, una ricerca con successo richiede un tempo $\Theta(1+\alpha)$ nel caso medio, nell'ipotesi di hashing uniforme semplice

Se il numero di celle della tavola hash è almeno proporzionale al numero di elementi della tavola, abbiamo $n = O(m)$ e, di conseguenza,
$\alpha = n/m = O(m)/m = O(1)$ nel caso peggiore e la cancellazione richiede il tempo $O(1)$ nel caso peggiore quando le liste sono doppiamente concatenate, tutte le operazioni di dizionario possono essere svolte, in media, nel tempo $O(1)$.

2.3 Funzioni hash

Una buona funzione hash soddisfa l'ipotesi dell'hashing uniforme semplice: ogni chiave ha la stessa probabilità di essere mandata in una qualsiasi delle m celle, indipendentemente dalla cella cui viene mandata qualsiasi altra chiave. Purtroppo, di solito non è possibile verificare questa condizione, in quanto raramente è nota la distribuzione delle probabilità secondo la quale vengono estratte le chiavi. Ad esempio se le chiavi sono numeri reali casuali k distribuiti in modo indipendente e uniforme nell'intervallo $0 \leq k < 1$, la funzione hash:

$$h(k) = \lfloor km \rfloor$$

soddisfa la condizione dell'hashing uniforme semplice.
La maggior parte delle funzioni hash suppone che l'universo delle chiavi sia l'insieme dei numeri naturali: $\mathbb{N} = \{0, 1, 2, ...\}$. Quindi se le chiavi non sono numeri naturali, occorre un metodo per interpretarle come tali.

2.3.1 Il metodo della divisione

Quando si applica il **metodo della divisione** per creare una funzione hash, una chiave k viene associata ad una delle m celle prendendo il resto della divisione fra k ed m, cioè la funzione hash è:

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

Un numero primo non troppo vicino a una potenza esatta di 2 è spesso una buona scelta per m. Per esempio supponiamo di allocare una tavola hash per contenere circa $n = 2000$ stringhe di caratteri, dove ogni carattere ha 8 bit. Poichè riteniamo accettabile esaminare in media 3 elementi in una ricerca senza successo, allochiamo una tavola hash di dimensione $m = 701$. Abbiamo

scelto 701 perchè è un numero primo vicino a 2000/3, ma non a una potenza di 2. Trattando ogni chiave k come un numero intero, la funzione hash diventa:

$$h(k) = k \mod 701$$

2.3.2 Il metodo della moltiplicazione

Il **metodo della moltiplicazione** per creare funzioni hash si svolge in due passi. Prima moltiplichiamo la chiave k per una costante A nell'intervallo $0 < A < 1$ ed estraiamo la parte frazionaria di kA. Poi moltiplichiamo questo valore per m e prendiamo la parte intera inferiore del risultato. In sintesi la funzione hash è:

$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

Un vantaggio del metodo di moltiplicazione è che non è critico. Tipicamente, lo scegliamo come una potenza di 2 ($m = 2^p$ per qualche intero p), il che rende semplice implementare la funzione hash nella maggior parte dei calcolatori.

2.4 Indirizzamento aperto

Nell'**indirizzamento aperto**, tutti gli elementi sono memorizzati nella tavola hash stessa; ovvero ogni cella della tavola contiene un elemento dell'insieme dinamico o la costante *nil*. Quando cerchiamo un elemento, esaminiamo sistematicamente le celle della tavola finchè non troviamo l'elemento desiderato o finchè non ci accorgiamo che l'elemento non si trova nella tavola.s Diversamente dal concatenamento non ci sono nè liste nè elementi memorizzati all'esterno della tavola, quindi la tavola può "riempirsi" al punto tale che non possono essere effettuati altri inserimenti. Una conseguenza è che il fattore di carico α non supera mai 1.
Il vantaggio dell'indirizzamento aperto sta nel fatto che esclude completamente i puntatori. Anzichè seguire i puntatori, *calcoliamo* la sequenza delle celle da esaminare. La memoria extra liberata per non aver memorizzato i puntatori offre alla tavola hash un maggior numero di celle, a parità di memoria occupata, consentendo potenzialmente di ridurre il numero di collisioni e di accelerare le operazioni di ricerca.
Per effettuare un inserimento mediante l'indirizzamento aperto, esaminiamo in successione le posizioni della tavola hash (**ispezione**), finchè non troviamo una cella vuota in cui inserire la chiave. La sequenza delle posizioni esaminate durante un'ispezione *dipende dalla chiave da inserire*.
Con l'indirizzamento aperto si richiede che, per ogni chiave k, la **sequenza di ispezione**:

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

sia una permutazione di $\langle 0, 1, ..., m-1 \rangle$, in modo che ogni posizione della tavola hash possa essere considerata come possibile cella in cui inserire una nuova chiave mentre la chiave si riempe.

2.4.1 Ispezione lineare

Data una funzione hash ordinaria $h' : U \to \{0, 1, ..., m-1\}$, che chiameremo **funzione hash ausiliaria**, il metodo dell'**ispezione lineare** usa la funzione hash:

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

per $i = 0, 1, ..., m-1$.
L'ispezione lineare è facile da implementare, ma presenta un problema noto come **addensamento primario**: si formano lunghe file di celle occupate, che aumentano il tempo medio di ricerca.

2.4.2 Ispezione quadratica

L'**ispezione quadratica** usa una funzione hash della forma:

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

dove h' è una funzione hash ausiliaria, c_1 e $c_2 \neq 0$ sono costanti ausiliarie e $i = 0, 1, ..., m-1$. La posizione iniziale esaminata è $T[h'(k)]$; le posizioni successivamente esaminate sono distanziate da quantità che dipendono in modo quadratico dal numero di ispezione i. Questa tecnica funziona molto meglio dell'ispezione lineare, ma per fare pieno uso della tavola hash, i valori di c_1, c_2 ed m non si possono scegliere arbitrariamente.

2.4.3 Doppio hashing

Il doppio hashing è uno dei metodo migliori disponibili per l'indirizzamento aperto, perchè le permutazioni prodotte hanno molte delle caratteristiche delle permutazioni scelte a caso. Il **doppio hashing** usa una funzione hash della forma:

$$h(k, i) = (h_1(k) + i h_2(k)) \mod m$$

dove h_1 e h_2 sono funzioni hash ausiliarie. L'ispezione inizia dalla posizione $T[h_1(k)]$; le successive posizioni sono distanziate dalle precedenti posizioni di una quantità $h_2(k)$, modulo m. Quindi, diversamente dal caso dell'ispezione linare o quadratica, la sequenza di ispezione qui dipende in due modi dalla chiave k, perchè possono variare sia la posizione iniziale di ispezione sia la distanza fra due posizioni successive di ispezione.
Il valore $h_2(k)$ dev'essere relativamente primo con la dimensione m della tavola hash perchè venga ispezionata l'intera tavola hash. Un modo pratico per garantire questa condizione è scegliere m potenza di 2 e definire h_2 in modo che produca sempre un numero dispari.

2.5 Domande

Domanda 1

Spiegare cos'è l'ipotesi di hash uniforme semplice. Come bisogna scegliere la funzione hash perchè tale ipotesi sia soddisfatta?

Risposta: l'ipotesi di hash uniforme semplice assume che una chiave da inserire nella tavola abbia uguale probabilità di finire in una qualsiasi delle celle della tavola, indipendentemente dalle chiavi inserite precedentemente. Poichè sia soddisfatta tale ipotesi, occorre scegliere la funzione hash tenendo conto della distribuzione di probabilità delle chiavi in input. Alternativamente possiamo scegliere casualmente la funzione hash in un insieme universale.

Domanda 2

Sia $h(k)$ una funzione hash che mappa un insieme U di chiavi in una tavola hash di m celle. Mostrare che esiste una cella in cui la funzione hash manda almeno $\lceil |U|/m \rceil$ chiavi distinte.

Risposta: se la funzione hash $h(k)$ mandasse al più $\lceil |U|/m \rceil - 1$ chiavi, in ogni cella dovremmo avere al massimo $m(\lceil |U|/m \rceil)$ chiavi. Sia $|U| = qm + r$, con q ed r quoziente e resto della divisione per m. Se $r = 0$ allora $m(\lceil |U|/m \rceil - 1) = |U| - 1 < |U|$: assurdo. Se $r > 0$ allora $m(\lceil |U|/m \rceil - 1) = mq < |U|$: assurdo. Deve quindi esistere almeno una cella in cui la funzione hash $h(k)$ manda almeno $\lceil |U|/m \rceil$ chiavi.

Domanda 3

Una tavola hash con risoluzione delle collisioni mediante liste, usa la funzione hash $h(k) = k \mod 257$. Tale tavola hash viene usata per memorizzare delle stringhe di caratteri ASCII. Mostrare che se la stringa y è ottenuta dalla stringa x permutandone i caratteri di posto pari allora x ed y vengono messe nella stessa cella.

Risposta: siccome $256 \mod 257 \equiv 1$ e quindi $256^i \mod 257 \equiv (-1)^i$ abbiamo che:

$$h(k) = k \mod 257 = (\sum_{i=0}^{n} c_i(-1)^i) \mod 257 =$$

$$= (\sum_{i=0}^{n} c_i(-1)^i) \mod 257$$

I caratteri di posto pari risultano tutti moltiplicati per 1 e quindi per la commutatività della somma, permutandoli tra loro non cambia il valore di $h(k)$.

Domanda 4

Dire perchè in una tavola hash con indirizzamento aperto non è possibile eliminare un elemento ponendo il valore *nil* nella cella ma occorre introdurre un nuovo valore *deleted* diverso sia da *nil* che da ogni chiave.

Risposta: Se l'elemento eliminato non è l'ultimo elemento della sequenza di ispezione a cui esso appartiene, ponendo il valore a *nil* nella cella la ricerca degli elementi successivi (che non sono stati rimossi) si ferma non appena trova il valore *nil* e dà risultato negativo.

Domanda 5

Spiegare il fenomeno dell'addensamento primario in una tavola hash con indirizzamento aperto e che usa l'ispezione lineare.

Risposta: L'ispezione lineare usa la funzione hash:

$$h(k,i) = (h'(k) + 1) \mod m$$

assumendo che $h'(k)$ soddisfi l'ipotesi di hash uniforme semplice, se la tavola è vuota l'inserimento della prima chiave può avvenire con uguale probabilità $1/m$ in una qualsiasi delle m celle della tavola. Ma essendo la cella $h'(k_1)$ occupata dalla chiave k_1, se $h'(k_2) = h'(k_1)$ la chiave k_2 viene rimandata nella cella successiva $h'(k_1) + 1$. Di conseguenza la probabilità che k_2 venga mandata nella cella $h'(K_1) + 1$ è doppia, ossia pari a $2/m$. In generale, se una cella libera è preceduta da n celle consecutive occupate, la probabilità che la prossima chiave venga messa in tale cella è pari ad $(n + 1)/m$. Di conseguenza sequenze consecutive di celle occupate tendono ad allungarsi sempre di più.

Domanda 6

Come la domanda 1 ma con $h(k) = k \mod 17$, e con y ottenuta da x permutando i caratteri.

Risposta: siccome $256^i \mod 17 = 1$ abbiamo che:

$$h(k) = k \mod 17 = (\sum_{i=0}^{n} c_i 256^i) \mod 17 = (\sum_{i=0}^{n} c_i) \mod 17$$

Per la commutatività della somma, permutando i caratteri non cambia il valore di $h(k)$.

3 Programmazione dinamica

In maniera del tutto generale la programmazione dinamica può essere descritta nel seguente modo:

1. Identifichiamo dei **sottoproblemi** del problema originario e utilizziamo una *tabella* per memorizzare i risultati intermedi;
2. Inizialmente vanno definiti i **valori iniziali** di alcuni elementi della tabella, corrispondenti a sottoproblemi più semplici;
3. Al generico passo, avanziamo in modo opportuno sulla tabella calcolando il valore della soluzione di un sottoproblema in base alla soluzione di sottoproblemi precedentemente risolti (corrispondenti ad elementi della tabella precedentemente calcolati);
4. Alla fine restituiamo la soluzione del problema originario, che è stato memorizzato in un particolare elemento della tabella.

La programmazione dinamica è usata normalmente per **problemi di ottimizzazione**, il termine "programmazione" si riferisce al metodo tabulare, non alla scrittura di codice.

La programmazione è applicabile con vantaggi se:

- Gode della proprietà di **sottostruttura ottima**: una soluzione si può costruire a partire da soluzioni ottime di sottoproblemi;
- Il numero di sottoproblemi distinti è molto minore del numero di soluzioni possibili tra cui cercare quella ottima, altrimenti c'è la **ripetizione di sottoproblemi**, ovvero se il numero di sottoproblemi distinti è molto minore del numero di soluzioni possibili tra cui cercare quella ottima, allora uno stesso sottoproblema deve comparire molte volte come sottoproblema di altri sottoproblemi.

3.1 Ordine di calcolo delle soluzioni dei sottoproblemi

Bottom-up: le soluzioni dei sottoproblemi del problema in esame sono già state calcolate. È il metodo migliore se per il calcolo della soluzione globale servono le soluzioni di tutti i sottoproblemi.

Top-down: è una procedura ricorsiva che dall'alto scende verso il basso. È la soluzione migliore se per il calcolo della soluzione globale servono soltanto alcune delle soluzioni dei sottoproblemi.

4 Algoritmi golosi

Si parla sempre di **problemi di ottimizzazione**.

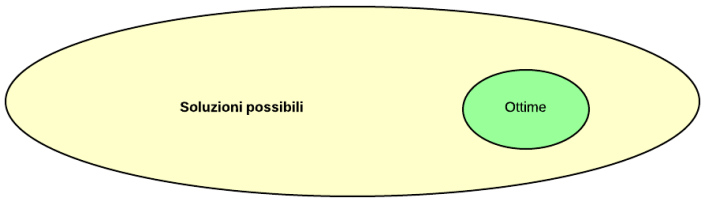


Figura 3: Soluzioni di un problema

Se utilizziamo l'**enumerazione esaustiva**:

- Si generano tutte le soluzioni possibili;
- Si calcola il costo di ciascuna di esse;
- Se ne seleziona una di ottima.

Questo metodo è chiaramente efficace ma comporta **tempi esponenziali**, perchè l'insieme delle soluzioni è generalmente enorme.
Un **algoritmo goloso** sceglie sempre una soluzione **localmente ottima**:

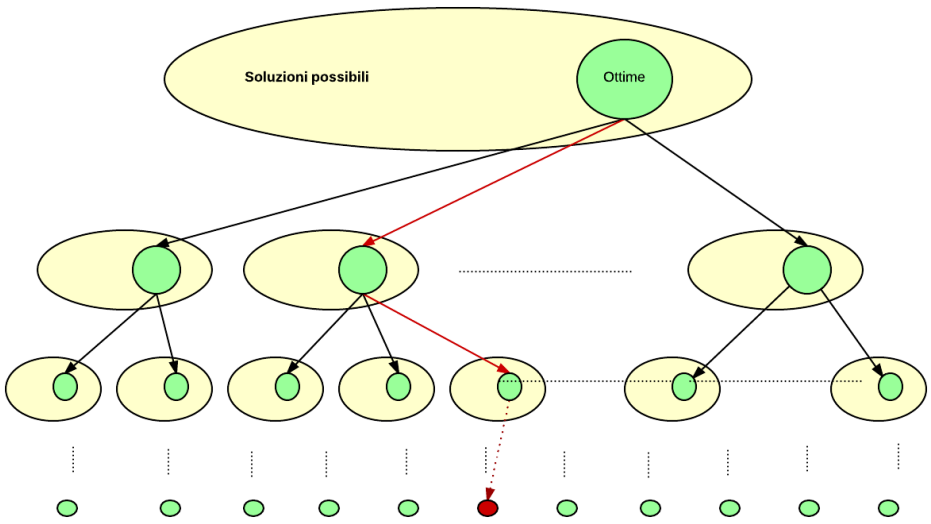


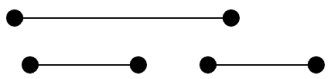
Figura 4: Schema di un algoritmo goloso

1. Ogni volta si fa una scelta che sembra migliore localmente;
2. In questo modo per alcuni problemi si ottiene una soluzione globalmente ottima.

4.1 Problema della scelta di attività

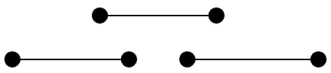
Un esempio tipico è quello del **problema della scelta di attività**.
Supponiamo di avere a disposizione n attività $a_1, a_2, ..., a_n$ che utilizzano la stessa risorsa (ad esempio un'aula). Ciascuna attività ha un tempo di inizio s_i e un tempo di fine f_i, con $0 \leq s_i < f_i < \infty$.
a_i occupa la risorsa nell'intervallo $[s_i, f_i)$.
a_i e a_j sono **compatibili** se gli intervalli $[s_i, f_i)$ e $(s_j, f_j]$ sono disgiunti.
Strategie golose:

- Scegliere l'attivitò che inizia per prima:



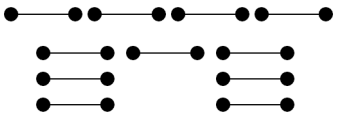
  Non funziona!

- Scegliere l'attività che dura meno tempo:



  Non funziona!

- Scegliere l'attività incompatibile con il minor numero di altre attività:



  Non funziona!

La strategia che funziona è quella di scegliere l'attività che **termina per prima**.

Listato 1: Algoritmo di selezione delle attività

```
ActivitySelector(a,s,f,n) // f1 ≤ f2 ≤ ... ≤ fn
       A = {a1}, k=1
       for m=2 to n
              if s[m] >= f[k]
                     A = A "unito" {am}, k = m
       return A
```

Sappiamo che durante tutta l'esecuzione dell'algoritmo esiste sempre una soluzione ottima contenente le attività scelte fino a quel momento. L'algoritmo

è goloso, perchè ad ogni passo sceglie l'attività che termina prima. Questa scelta è **localmente ottima**, perchè è quella che lascia più tempo a disposizione per le attività successive.

4.2 Problema dello zaino frazionario

Dati n tipi di merce, $M_1, M_2, ..., M_n$ in quantità $q_1, q_2, ..., q_n$ e con costi unitari $c_1, c_2, ..., c_n$ si vuole riempire uno zaino di capacità Q in modo che il contenuto abbia costo massimo.

Listato 2: Algoritmo goloso dello zaino frazionario

```
RiempiZaino(q, c, n, Q) // c_1 ≥ c_2 ≥ c_3 ≥ ... ≥ c_n
       Spazio = Q
       for i=1 to n
              if Spazio >= q[i]
                     Z[i] = q[i], Spazio = Spazio - Z[i]
              else
                     Z[i] = Spazio, Spazio = 0
       return Z
```

4.3 Problema del pulmino con numero fissato di posti

Un'azienda di una grande città ha diverse agenzie. Su richiesta sindacale l'azienda ha istituito un servizio sperimentale di trasporto per i dipendenti che utilizza un solo pulmino con 10 posti. Al mattino il pulmino effettua un percorso prefissato che passa per tutte le agenzie raccogliendo gli n dipendenti che hanno fatto richiesta nei punti del percorso per loro più comodi per portarli alle rispettive agenzie. Per ogni dipendente è noto il punto di partenza $s[i]$ e il punto di arrivo $f[i]$. Essendo il servizio sperimentale non ci si aspetta che tutte le richieste possano essere soddisfatte.
Descrivere un algoritmo goloso che determina un'assegnazione dei posti del pulmino che permette di trasportare il massimo numero di dipendenti.
Naturalmente lo stesso posto può essere utilizzato da più dipendenti, purchè il loro tragitti non si sovrappongano.
Soluzione: Ordiniamo i dipendenti per i punti di arrivo, dal più vicino al più lontano. La scelta golosa consiste poi nell'assegnare a ogni dipendente il posto che si è liberato per ultimo, in modo da massimizzare il numero di dipendenti.

Listato 3: Algoritmo goloso del pulmino

```
Pulmino(s, f, n, m) // PRE: f_1 ≤ f_2 ≤ ... ≤ f_n
       for j=1 to m
              t_j = 0
```

```
t_0 = -1
for i=1 to n
        h = 0
        for j=1 to m
                if s_i ≥ t_j and t_j ≥ t_h then h = j
                // A_h Ã¨ il posto che si libera per ultimo tra
                     quelli in cui Ã¨ possibile far sedere una
                     persona.
                // Se h == 0 in nessuno degli m posti si puo' far
                     sedere una persona
        J[i] = h
        if h ≠ 0 then t_h = f_i
// POST: J[1...n] Ã¨ una programmazione ottima del massimo
    numero di persone che si possono trasportare con m posti.
// J[i] == j ≠ 0 significa che la persona non puo' sedersi in
    nessuno degli m posti
return J
```

5 Codici di Huffman

I codici di Huffman vengono tipicamente utilizzati nella **compressione dati**, consentendo un risparmio tra il 20% e il 90%. Sulla base delle frequenze con cui ogni carattere appare nel file, l'algoritmo di Huffman trova un **codice ottimo**, ossia un modo ottimale di associare ad ogni carattere una sequenza di bit detta **parola di codice**.

Carattere	a	b	c	d	e	f
Frequenza	57	13	12	24	9	5

Occorrono 3 bit per rappresentare 6 caratteri:

Carattere	a	b	c	d	e	f
Codice fisso	000	001	010	011	100	101

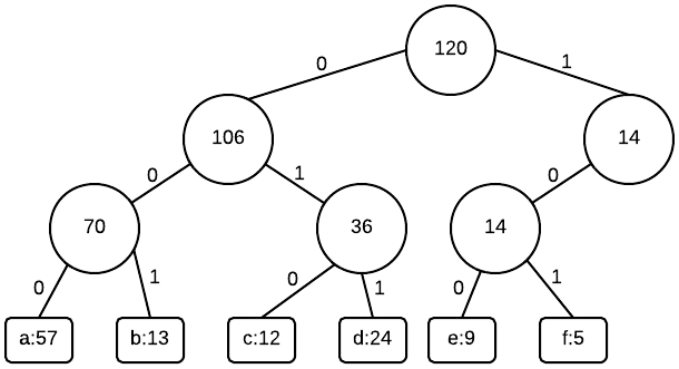
Per codificare il file occorrono $120 * 3 = 360$ bit.

Un codice si dice **prefisso** se nessuna parola codice è prefisso (parte iniziale) di un'altra. Ogni codice a lunghezza fissa è prefisso.

Esempio: codice a **lunghezza fissa**

Carattere	a	b	c	d	e	f
Frequenza	57	13	12	24	9	5
Codice fisso	000	001	010	011	100	101

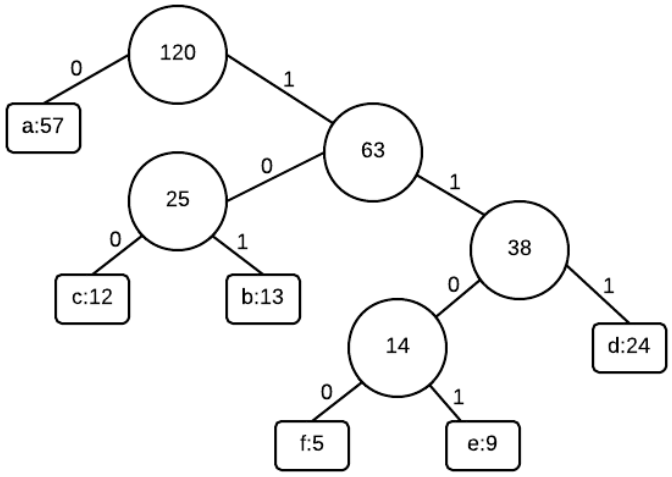
Vediamo la rappresentazione ad albero:



Tutti i caratteri sono foglie e sono allo stesso livello.

Esempio: codice a **lunghezza variabile**

Carattere	a	b	c	d	e	f
Frequenza	57	13	12	24	9	5
Codice fisso	0	101	100	111	1101	1100



La lunghezza in bit del file codificato con il codice rappresentato da un albero T è:

$$B(T) = \sum_{c \in \sum} f_c d_t(c)$$

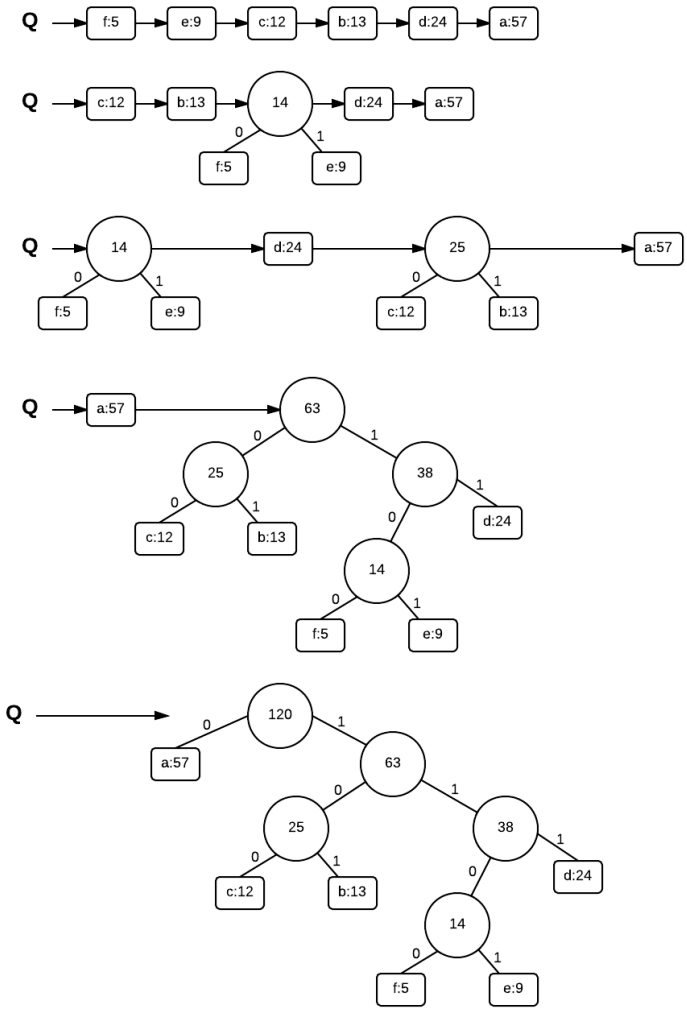
dove $c \in \sum$ significa una sommatoria estesa a tutti i caratteri dell'alfabeto, f_c è la frequenza del carattere c, $d_t(c)$ è la profondità della foglia che rappresenta il carattere c nell'albero T.

Nota: assumiamo che l'alfabeto $\sum$ contenga almeno due caratteri. In caso contrario basata un numero per rappresentare il file: la sua lunghezza.

5.1 Costruzione dell'albero del codice

Carattere	a	b	c	d	e	f
Frequenza	57	13	12	24	9	5

Vediamo di seguito l'algoritmo utilizzato per costruire l'albero:


Q
f:5
e:9
c:12
b:13
d:24
a:57
Q
c:12
b:13
14
d:24
a:57
0
1
f:5
e:9
Q
14
d:24
25
a:57
0
1
0
1
f:5
e:9
c:12
b:13
Q
a:57
63
0
1
25
38
1
0
1
0
1
c:12
b:13
d:24
14
0
1
f:5
e:9
Q
120
0
1
a:57
63
0
1
25
38
1
0
1
0
1
c:12
b:13
d:24
14
0
1
f:5
e:9

Listato 4: Huffman

```
Huffman(c, f, n)
       Q = 0∅
       for i=1 to n
              Push(Q, Nodo(f_i, c_i))
       for j=n dowto 2
              x = ExtractMin(Q)
              y = ExtractMin(Q)
              Push(Q, Nodo(x,y))
       return ExtractMin(Q)
```

$Nodo(f, c)$ è il costruttore dei nodi foglia. $Nodo(x, y)$ è il costruttore dei nodi interni. La complessità dell'algoritmo è $O(n \log n$.

La tecnica di base per generare l'albero di un codice prefisso ottimo è quindi la seguente:

1. Ordinare la lista di caratteri per frequenza crescente;
2. Finchè la lista non contiene un solo elemento:
   (a) Estrarre i due caratteri con frequenza minore;
   (b) Creare un nuovo albero che ha per figli i due caratteri del punto precedente e per radice la somma delle loro frequenze;
   (c) Inserire l'albero nella lista (cancellando i fieli dalla lista).
3. Assegnare ad ogni carattere un codice secondo la definizione di albero del codice: un albero del codice è un albero binario le cui foglie sono i caratteri e una parola in codice viene interpretata come il cammino semplice dalla radice a quel carattere, dove 0 significa raggiungi il figlio sinistro e 1 significa raggiungi il figlio destro.

6 Analisi ammortizzata

Si consideri il tempo richiesto per eseguire, nel **caso pessimo**, un'intera sequenza di operazioni. Se le operazioni costose sono relativamente meno frequenti allora il costo richiesto per eseguirle può essere ammortizzato con l'esecuzione delle operazioni meno costose.

6.1 Metodo dell'aggregazione

Si basa sul concetto di **costo ammortizzato**: data una sequenza di n istruzioni aventi complessità $O(f(n))$, il costo ammortizzato della singola operazione si ottiene dividendo la complessità totale per il numero di istruzioni.

$$\hat{c} = \frac{O(f(n))}{n}$$

6.2 Metodo degli accantonamenti (o dei crediti)

Si caricano le operazioni meno costose di un costo aggiuntivo. Il costo aggiuntivo viene assegnato come **credito prepagato** a certi oggetti nella struttura dati. I crediti accumulati saranno usati per pagare le operazioni più costose su tali oggetti.
Il costo ammortizzato delle operazioni meno costose è il costo effettivo aumentato del costo aggiuntivo.
Il costo ammortizzato delle operazioni più costose è il costo effettivo diminuito del credito prepagato utilizzato.
Il costo artificiale fornisce un **limite superiore** al costo ammortizzato.

6.3 Metodo del potenziale

Si associa alla struttura dati D un **potenziale** $\Phi(D)$ tale che la modifica della struttura dati dovuta alle operazioni meno costose comporti un aumento del potenziale, mentre le operazioni meno costose lo facciano diminuire.
Il costo ammortizzato è quindi la **somma algebrica** del costo effettivo e della variazione di potenziale. D_i è la struttura dati dopo la c-esima operazione e c_i + il costo dell'i-esima operazione.

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

Il costo ammortizzato di una sequenza di n operazioni è:

$$\begin{aligned}\hat{c} = \sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} [c_i + \Phi(D_i) - \Phi(D_{i-1})] \\ &= c + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

Se la variazione $\Phi(D_n) - \Phi(D_0)$ del potenziale relativo all'esecuzione di tutta la sequenza non è negativa allora il costo ammortizzato $\hat{c}$ è una **maggiorazione** del costo reale c.
Altrimenti un valore $\Phi(D_n) - \Phi(D_0)$ negativo deve essere compensato con un aumento adeguato del costo ammortizzato delle operazioni.

6.4 Esercizio 1

Si vuole realizzare un contatore binario usando un array $A[0...k]$ per memorizzare i $k+1$ bit $b_k...b_1b_0$ della rappresentazione binaria del valore x del contatore. Si vuole che il contatore possa iniziare anche con un valore maggiore di zero. A tale scopop si vuole che, oltre all'operazione **increment**, che aumenta di 1 il valore del contatore, sia prevista anche un'operazione iniziale **SET(A,n)**, che inizializza ad n il valore del contatore. Usare il **metodo di aggregazione** per dimostrare che le operazioni di una sequenza costituita da una set seguita da m increment, con $m = \Omega(k)$ hanno costo ammortizzato costante.

Soluzione:Vediamo anzitutto gli pseudo-codici delle due funzioni:

```
Increment(A)
        i = 0
        while i<=k and A[i]==1
                A[i] = 0
                i++
        if i<=k
                A[i] = 1
```

```
Set(A,n) // O(k)
        // PRE: A azzerato
        while i<=k and n>0
                A[i] = n%2
                n = n/2    // preso per difetto
                i++
```

Sia $\Phi(A) =$"numero di bit impostati a 1". Il costo di una increment è $c = 1 + t$, dove $t \geq 0$ è il numero di bit trasformati in 1 da 0. Sia A_0 lo stato iniziale del contatore azzerato ed A_1 il suo stato dopo aver eseguito una Set(A,n). Supponiamo di effettuare m esecuzioni di increment per valutare l'analisi ammortizzata:

$$\Delta\Phi = \Phi(A_m) - \Phi(A_1) = t - 1$$

Il numero di bit 1 rispetto ad A_1 varia di $-t + 1$ in A_m. Dunque aggiungiamo k alla formula per calcolare $\hat{c}$, distribuendolo a tutte le operazioni;

$$\hat{c} = c + \Phi(A_n) - \Phi(A_0) + \frac{k}{m} = 1 + t - t + 1 + \frac{k}{m}$$

Alla fine il costo ammortizzato è $O(1 + \frac{k}{m})$, $m = \Omega(k), m \geq k$.

$$\frac{O(k)+O(1+\frac{k}{m})}{m+1} = O(\frac{k+1+k/m}{m+1}) \leq O(\frac{k+1+k/k}{k+1}) = O(\frac{k+2}{k+1}) = O(1)$$

6.5 Esercizio 2

Si vuole realizzare un timer usando un array A per memorizzare i $k+1$ bit $b_k, ..., b_1, b_0$ della rappresentazione binaria del valore del timer. Le operazioni previste per un timer sono *Set(A,n)* che carica il timer ad n secondi e *Decrement(A)* che diminuisce di un secondo il valore del timer. L'operazione Set si può eseguire solamente quando il timer è azzerato mentre l'operazione Decrement si può eseguire soltanto quando il timer ha valore maggiore di 0. Scrivere le due funzioni Set e Decrement ed analizzarne la complessità ammortizzata.

Suggerimento: Memorizzare l'indice m del bit 1 più significativo ($m = -1$ se tutti i bit sono 0, $m = k$ se tutti i bit sono a 1) ed usare come funzione potenziale il numero di bit uguali a 0 che precedono $A[m]$ più due volte il numero di bit che seguono $A[m]$ (che sono tutti 0), ossia
$\Phi = \sum_{i=0}^{m-1}(1-b_i) + 2(k-m)$.

Soluzione: Vediamo anzitutto gli pseudo-codici delle due operazioni:

```
Set(A,n) // tutti i bit a 0, n<2^{k+1}
        k = A.lenght
        i = 0 // parto dal bit meno significativo
        while n>0 and i<=k
                A[i] = n%2 // il resto della divisione per 2
                n = n/2  // divisione intera per difetto
                i++
        if i>k
                error "underflow"
        else
                A.m = i-1
```

```
Decrement(A)
        i = 0
        while A[i] == 0
                A[i] = 1
                i++
        A[i] = 0 // ora devo mettere apposto l'n
        if A.m == i
                A.m = A.m-1
```

Analisi ammortizzata di Set:

$$\hat{c} = c + \Delta\Phi$$

$$\Delta\Phi = \Phi + \Phi' = 2(k-m) + \sum_{i=0}^{m-1}(1-b_i) - 2(k+1) =$$

$$= 2k - 2m + \sum_{i=0}^{m-1}(1-b_i) - 2k - 2$$

$$\hat{c} \leq m + 1 - 2m + m - 2 \leq -1$$

Il costo è costante quindi ho concluso la dimostrazione.

Analisi ammortizzata di Decrement

$c = c + 1$, proporzionale a $t - 1$, dove t è il numero di bit trasformati in 0.

$$\hat{c} = c + \Delta\Phi$$

$$\Delta\Phi = \Phi - \Phi' = 2(k-m) + \sum_{i=0}^{m-1}(1-b_i) - 2(k-m') + \sum_{i=0}^{m'-1}(1-b_i) =$$

$$-2m + 2m' - t + 1$$

m ed m' possono al massimo diversi tra loro di 1, quindi:

$$\Delta\Phi \leq 2 - t + 1 = 3 - t$$

$$\hat{c} = t + 1 + 3 - t = 4$$

Ottengo un valore costante, inoltre:

$$\Phi_0 = 2(k+1)$$

In questo caso perchè il costo ammortizzato vada bene deve essere maggiore o uguale del costo reale.
Prendiamo le operazioni di segno:

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$$

$$\sum_{i=1}^{n}\hat{c}_i = \sum_{i=1}^{n}\hat{c}_i + \Phi_n + \Phi_0 \leq \sum_{i=0} n(\hat{c}_i - \frac{\Phi_0}{n})$$

$$\hat{c}_i = c_i + \frac{\Phi_0}{n}$$

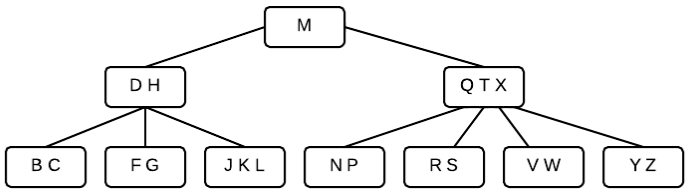
Devo distribuire, il costo ammortizzato è quindi:

$$\hat{c} = 4 + \frac{2(k+1)}{n}$$

Se $n = \Omega(k) \Rightarrow \hat{c} = O(1)$

7 B-alberi

I B-alberi sono **alberi bilanciati** adatti per memorizzare grandi masse di dati in memoria secondaria. Sono simili agli alberi rosso-neri, ma sono progettati per minimizzare gli accessi alla memoria secondaria. Le operazioni sono **insert**, **delete**, **search**, **split** e **join**.
I nodi dei B-alberi possono avere un numero di chiavi $n \geq 1$ ed $n + 1$ figli.



7.1 Definizione di un B-albero

Un B-albero T è un albero di radice $T.root$ tale che:

1. Ogni nodo x contiene i seguenti campi:
   (a) $x.n$: il numero di chiavi presenti nel nodo;
   (b) $x.key_1 \leq x.key_2 \leq ... \leq x.key_n$: le $x.n$ chiavi in ordine crescente;
   (c) $x.leaf$: valore booleano che è true se il nodo x è una foglia, false altrimenti.
2. Se il nodo non è una foglia contiene anche $x.c_1, X.c_2, x.c_{n+1}$: gli $n + 1$ puntatori ai figli;
3. Le chiavi $x.key_1, x.key_2, ..., x.key_n$ di un nodo interno separano le chiavi dei sottoalberi. Se k_i è una chiave qualsiasi del sottoalbero $x.c_i$, allora:

$$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq ... \leq x.key_{x.n} \leq k_{x.n+1}$$

4. Le foglie sono tutte alla stessa profondità h, detta **altezza dell'albero**;
5. Vi sono limiti inferiori e superiori al numero di chiavi in un nodo, e tali limiti dipendono da una costante t detta **grado minimo** del B-albero.
   (a) Ogni nodo, eccetto la radice, ha almeno $t - 1$ chiavi e, se non è una foglia, ha almeno t figli;
   (b) Se l'albero non è vuoto la radice contiene almeno una chiave;
   (c) Ogni nodo ha al più $2t - 1$ chiavi e, se non è foglia, ha al più $2t$ figli.

Ad ogni chiave sono generalmente associate delle informazioni ausiliarie. Assumeremo implicitamente che quando viene copiata una chiave vengono copiate anche tali informazioni. I B-alberi più semplici sono quelli con grado

minimo $t = 2$. Ogni nodo interno ha 2,3 o 4 figli. Ogni albero non vuoto di grado minimo t con N chiavi ha altezza:

$$h \leq \log_t \frac{N+1}{2}$$

7.2 Operazioni sui B-alberi

Adottiamo le seguenti convenzioni per le operazioni sui B-alberi:

1. La radice del B-albero è sempre in memoria;
2. I nodi passati come parametri alle procedure sono stati preventivamente caricati in memoria.

7.2.1 Creazione di un B-albero vuoto

La procedura ausiliaria **ALLOCATE-NODE** alloca una pagina del disco da utilizzare come nuovo nodo nel tempo $O(1)$. Supponiamo che un nodo creato da ALLOCATE-NODE non richieda l'operazione DISK-WRITE.

Listato 5: Creazione di un B-albero

```
B-TREE-CREATE(T)
        x = ALLOCATE-NODE()
        x.leaf = true
        x.n = 0
        DISK-WRITE(x)
        T.root = x
```

La procedura B-TREE-CREATE richiede $O(1)$ operazioni su disco e quindi un tempo di CPU pari a $O(1)$.

7.2.2 Ricerca in un B-albero

Listato 6: Ricerca in un B-albero

```
B-TREE-SEARCH(x, k)
        i = 1
        while i ≤ x.n and k > x.key_i
                i = i+1
        if i≤ x.n and k == x.key_i
                return(x, i)
        elseif x.leaf
                return nil
        else DISK-READ(x,c_i)
```

```
        return B-TREE-SEARCH(x.c_i, k)
```

La procedura richiede tempo $O(\log_2 N)$. Il numero di $DISK - READ$ è al più uguale all'altezza h dell'albero, quindi $O(\log_t N)$. Il tempo di CPU della procedura è:

$$\begin{aligned} T &\leq (h+1)\log_2(2t-1) \\ &= O(\log_t N \log_2 t) \\ &= O(\log_2 N) \end{aligned}$$

7.2.3 Inserimento di una chiave

L'aggiunta di una chiave ad un B-albero può avvenire soltanto in una foglia e soltanto se la foglia non è piena, cioè non ha il numero massimo $2t-1$ di chiavi. Possiamo garantirci che la foglia a cui arriveremo non sia piena se ad ogni passo nella discesa dalla radice ci assicuriamo che il figlio su cui scendiamo non sia pieno. Nel caso in cui tale figlio sia pieno chiamiamo prima una particolare funzione $SPLIT - CHILD$ che lo divide in due.

Listato 7: Procedura di divisione di un nodo

```
B-TREE-SPLIT-CHILD(x, i)
       z = ALLOCATE-NODE()
       y = x.c_i
       z.leaf = y.leaf
       z.n = t-1
       for j=1 to t-1
              z.key_j = y.key_{j+t}
       if not y.leaf
              for j=1 to t
                     z.c_j = y.c_{j+t}
       y.n = t-1
       for j=x.n+1 downto i+1
              x.c_{j+1} = x.c_j
       x.c_{i+1} = z
       for j=x.n downto i
              x.key_{j+1} = x.key_j
       x.key_i = y.key_t
       x.n = x.n+1
       DISK-WRITE(y)
       DISK-WRITE(z)
       DISK-WRITE(x)
```

Inseriamo ora una chiave k nel B-albero T di altezza h con un singolo passaggio in discesa nell'albero, richiedendo $O(h)$ accessi. Il tempo di CPU richiesto è

$O(th) = O(t \log_t n)$. La procedura B-TRSERT usa B-TREE-SPLIT-CHILD per garantire che la ricorsione non arrivi mai a un nodo pieno.

Listato 8: Inserimento di una chiave

```
B-TREE-INSERT(T,k)
        r = T.root
        if r.n == 2t-1
                s = ALLOCATE-NODE()
                T.root = s
                s.leaf = false
                s.n = =
                s.c_1 = r
                B-TREE-SPLIT-CHILD(s,1)
                B-TREE-INSERT-NONFULL(s,k)
        else
                B-TREE-INSERT-NONFULL(r,k)
```

La procedura finisce chiamando B-TREE-INSERT-NONFULL per inserire la chiave k nell'albero che ha un nodo radice non pieno. La procedura B-TREE-INSERT-NONFULL effettua la ricorsione quante volte serve per discendere l'albero, assicurandosi che il nodo in cui effettua la ricorsione non sia mai pieno, chiamando B-TREE-SPLIT-CHILD se necessario.

Listato 9: B-Tree insert nonfull

```
B-TREE-INSERT-NONFULL(x,k)
        i = x.n
        if x.leaf
                while i ≥ 1 and k<x.key_i
                        x.key_{i+1} = x.key_i
                        i = i-1
                x.key_{i+1}=k
                x.n = x.n+1
                DISK-WRITE(x)
        else while i ≥ 1 and k<x.key_i
                        i = i-1
                i = i+1
                DISK-READ(x.c_i)
                if x.c_i.n == 2t-1
                        B-TREE-SPLIT-CHILD(x,i)
                        if k>x.key_i
                                i = i+1
                        B-TREE-INSERT-NONFULL(x.c_i,k)
```

Il tempo totale della CPU è $O(th) = O(t \log_t n)$.

7.2.4 Cancellazione di una chiave

La cancellazione di una chiave da un B-albero è analoga all'inserimento, ma un po' più complicata, perchè una chiave può essere cancellata da un nodo qualsiasi e l'eliminazione di una chiave da un nodo interno richiede che i figli del nodo vengano riorganizzati. Azichè lo pseudocodice, descriveremo il funzionamento dell'operazione di cancellazione.

1. Se la chiave si trova nel nodo x e x è una foglia, cancellare la chiave k da x;
2. Se la chiave k si trova nel nodo x e x è un nodo interno, eseguire le seguenti operazioni:
   (a) Se il figlio y che precede k nel nodo x ha almeno t chiavi, trovare il predecessore k' di k nel sottoalbero con radice in y. Cancellare ricorsivamente k' e sostituire k con k' in x;
   (b) Simmetricamente, se il figlio z che segue k nel nodo x ha almeno t chiavi, trovare il successore k' di k nel sottoalbero con radice in z. Cancellare ricorsivamente k' e sostituire k con k' in x;
   (c) Altrimenti, se entrambi i nodi y e z hanno soltanto $t-1$ chiavi, fondere k e tutto z nel nodo y, in modo che x perda sia k sia il puntatore a z; adesso y contiene $2t-1$ chiavi. Poi, rilasciare z e cancellare ricorsivamente k da y.
3. Se la chiave k non è presente nel nodo interno x, determinare la radice $x.c_i$ del sottoalbero appropriato che deve contenere k, se k è davvero presente nell'albero. Se $x.c_i$ ha soltanto $t-1$ chiavi, eseguire il passo 3a o 3b, se necessario, per avere la garanzia di arrivare a un nodo che contiene almeno t chiavi. Poi, finire effettuando la ricorsione sul figlio appropriato di x.
   (a) Se $x.c_i$ ha soltanto $t-1$ chiavi, ma ha un fratello adiacente con almeno t chiavi, assegnare a $x.c_i$ una chiave extra, spostando in basso una chiave da x in $x.c_i$, spostando in alto una chiave dal fratello sinistro o destro di $x.c_i$ in x e, infine, spostando in $x.c_i$ un puntatore al figlio del fratello (quello adiacente alla chiave spostata);
   (b) Se $x.c_i$ ed entrambi i fratelli adiacenti di $x.c_i$ hanno $t-1$ chiavi, fondere $x.c_i$ con un fratello in un nuovo nodo; questo richiede che una chiave venga spostata in basso da x nel nuovo nodo per diventare la chiave mediana di questo nodo.

8 Strutture dati per insiemi disgiunti

Una **struttura dati per insiemi disgiunti** mantiene una collezione:

$$C = \{S_1, S_2, ..., S_k\}$$

di insiemi dinamici disgiunti. Ciascun insieme è identificato da un **rappresentante**, che è uno degli elementi dell'insieme. Ogni elemento di un insieme è rappresentato da un oggetto. Indicando con x un oggetto, vogliamo supportare le seguenti operazioni:

- **MakeSet**: aggiunge alla struttura dati un nuovo insieme contenente solo l'elemento x. Si richiede che x non compaia in nessun altro insieme della struttura;
- **FindSet**: ritorna il rappresentante dell'insieme che contiene x;
- **Union**: riunisce i due insiemi contenenti x ed y in un unico insieme.

8.1 Rappresentazione tramite liste concatenate

Il modo più semplice per rappresentare una collezione di insiemi disgiunti è usare una lista circolare per ciascun insieme.

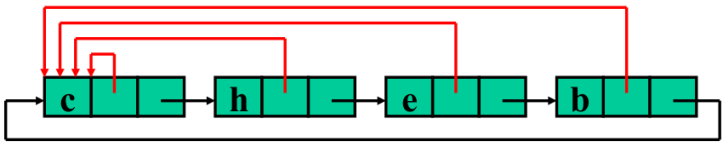


Figura 5: Rappresntazione con liste concatenate

I nodi hanno i seguenti campi:

- **info**: l'informazione contenuta nel nodo;
- **r**: il puntatore al rappresentante;
- **succ**: il puntatore al nodo successivo.

Le operazioni sono:

```
MakeSet(x)
        x.r = x
        x.succ = x
```

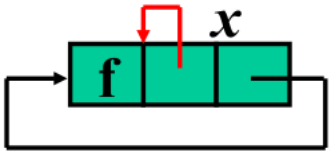


Figura 6: Lista con l'operazione MakeSet

```
FindSet(x)
        return x.r
```

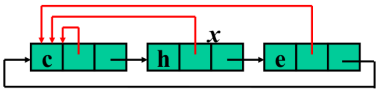


Figura 7: Lista con l'operazione FindSet

```
Union(x,y)
        // cambia i puntatori r nella lista di y
        y.r = x.r, z = y.succ
        while z ≠ y
                z.r = x.r, z = z.succ
        // concatena le due liste
        z = x.succ, x.succ = y.succ, y.succ = z
```

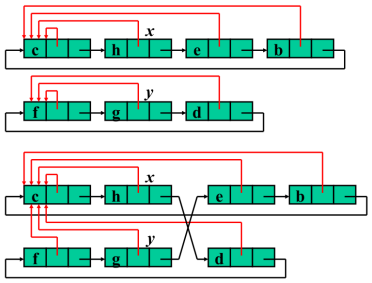


Figura 8: Lista con l'operazione Union

La complessità di Union dipende dal numero di iterazioni richieste dal ciclo

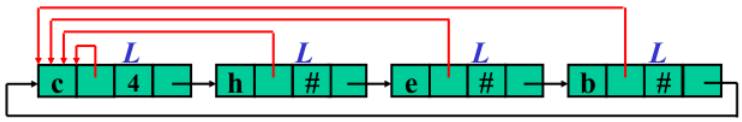
che cambia i puntatori al rappresentante dei nodi della lista contenente y.
Quindi Union ha complessità uguale a: $O(n_2)$, dove n_2 è la lunghezza della seconda lista.
Consideriamo la sequenza di $2n-1$ operazioni su n oggetti:

Operazione	Numero di oggetti aggiornati
MakeSet(x_1)	1
MakeSet(x_2)	1
...	...
...	...
MakeSet(x_n)	1
Union(x_2, x_1)	1
Union(x_3, x_2)	2
Union(x_4, x_3)	3
...	...
...	...
Union(x_n, x_{n-1})	$n-1$

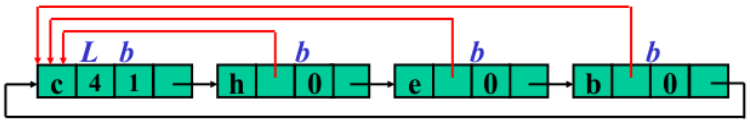
Il costo totale è proporzionale ad $n + n(n-1)/2$ ed è $\Theta(n^2)$ e le operazioni hanno costo ammortizzato $O(n)$.

8.2 Euristica dell'unione pesata

La complessità $\Theta(n^2)$ dell'esempio è dovuta al fatto che, in ogni Union, la seconda lista, quella che viene percorsa per aggiornare i puntatori al rappresentante, è la più lunga delle due.
L'euristica dell'**unione pesata** sceglie sempre la lista più corta per aggiornare i puntatori al rappresentante. Per poter fare ciò basta memorizzare la lunghezza della lista in un nuovo campo L del rappresentante.



Si può risparmiare memoria usando un campo booleano b per distinguere il rappresentante.



Naturalmente occorre modificare le funzioni:

```
MakeSet(x)
        x.b = true
        x.L = 1
        x.succ = x
```

```
FindSet(x)
        if x.b
                return x
        else return x.r
```

```
Union(x,y)
        x = FindSet(x)
        y = FindSet(y)
        // se la lista di x Ã¨ piu' corta scambia x con y
        if x.L < y.L
                z = x, x = y, y = z
        x.L = x.L + y.L
        // cambia rappresentante alla lista di y
        y.b = false, y.r = x, z = y.succ
        while z ≠ y
                z.r = x, z = z.succ
        // concatena le due liste
        z = x.succ, x.succ = y.succ, y.succ = z
```

Con l'euristica dell'unione pesata una sequenza di m operazioni delle quali n sono MakeSet richiede tempo $O(m + n \log n)$. La complessità ammortizzata delle operazioni è quindi:

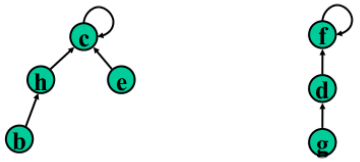
$$O(\frac{m + n\log n}{m}) = O(1 + \frac{n\log n}{m}) = O(\log n)$$

Se il numero n di MakeSet è molto minore di m per cui $n\log n = O(m)$:

$$O(1 + \frac{n\log n}{m}) = O(1)$$

8.3 Rappresentazione con foreste

Una rappresentazione più efficiente si ottiene usando **foreste di insiemi disgiunti**. Ogni insieme è rappresentato da un albero i cui nodi, oltre al campo info che contiene l'informazione, hanno soltanto un campo p che punta al padre.

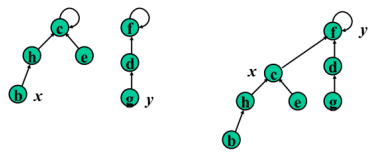


Implementazione semplice:

```
MakeSet(x)
       x.p = x
```

```
FindSet(x)
       while x.p ≠ x
              x = x.p
       return x
```

```
Union(x,y)
       x = FindSet(x)
       y = FindSet(y)
       x.p = y
```



Sia nella rappresentazione con liste circolari che in quella con alberi non abbiamo indicato nessun puntatore esterno alla lista o all'albero. In realtà una struttura dati per insiemi disgiunti non è pensata per memorizzare dei dati ma soltanto per raggruppare in insiemi disgiunti dei dati che sono già memorizzati in qualche altra struttura: array, pila, lista, albero, tavola hash, ecc.
La complessità di FindSet(x) è pari alla lunghezza del cammino che congiunge il nodo x alla radice dell'albero.
La complessità di Union è essenzialmente quella delle due chiamate FindSet(x) e FindSet(y).
Un esempio analogo a quello usato con le liste mostra che una sequenza di n operazioni può richiedere tempo $O(n^2)$.
Possiamo migliorare notevolmente l'efficienza usando due euristiche:

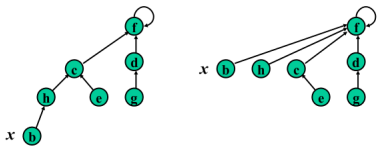
L'euristica dell'unione per rango: è simile a quella dell'unione pesate per le liste. In ogni nodo x manteniamo un campo rank che è un limite superiore all'altezza del sottoalbero di radice x ed è anche un'approssimazione del logaritmo del numero di nodi del sottoalbero. L'operazione Union mette la radice con rango minore come figlia di quella di rango maggiore.

L'euristica della complessità dei cammini: quando effettuiamo una FindSet(x) attraversiamo il cammino da x alla radice. Posiamo approfittarne per far puntare alla radice dell'albero i puntatori al padre di tutti i nodi incontrati lungo il cammino. Le successive operazioni FindSet sui nodi di tale cammino risulteranno molto meno onerose.

L'implementazione con entrambe le euristiche è la seguente:

```
MakeSet(x)
       x.p = x
       x.rank = 0
```

```
FindSet(x)
       if x.p ≠ x
              x.p = FindSet(x.p)
       return x.p
```




```
Union(x,y)
        x = FindSet(x)
        y = FindSet(y)
        Link(x,y)
```

```
Link(x,y)
        if x.rank > y.rank
                y.p = x
        else
                x.p == y
                if x.rank == y.rank
                        y.rank = y.rank+1
```

8.4 Proprietà dei ranghi

Per ogni nodo x:

- x viene creato come radice con $x.rank = 0$;
- $x.rank$ aumenta soltanto finchè x resta radice, dopo di che rimane invariato;
- Se x non è radice $x.rank < x.p.rank$;
- $x.p.rank$ può solo aumentare;
- Dopo aver eseguito n operazioni $x.rank < n$.

8.5 Le due funzioni $level(x)$ e $iter(x)$

Per ogni nodo x non radice definiamo due funzioni che "misurano" la differenza di rango tra x e $x.p$:

$$level(x) = max\{k : x.p.rank \geq A_k(x.rank)\}$$

$$iter(x) = max\{i : x.p.rank \geq A^{(i)}_{level(x)}(x.rank)\}$$

Valgono le seguenti disuguaglianze:

$$0 \leq level(x) < \alpha(n)$$
$$1 \leq iter(x) \leq x.rank$$

8.6 Calcolo del costo ammortizzato delle operazioni

8.6.1 MakeSet(x)

$$\Delta\Phi_i = 0$$
$$\hat{c} = c + \Delta\Phi_i = 1 = O(1)$$

8.6.2 Link(x,y)

Il potenziale $\Phi(x)$ dipende soltanto da $x.rank$ e $x.p.rank$. Quindi gli unici elementi il cui potenziale può cambiare sono x, y e i figli di y. I figli di y non sono radici e il loro potenziale non può aumentare. x è radice e diventa figlio di y ma il suo rango non cambia. Se $x.rank = 0$:

$$\Delta\Phi_i(x) = \alpha(n)x.rank - \alpha(n)x.rank = 0$$

Altrimenti:

$$\Delta\Phi_i(x) = (\alpha(n) - level(x))x.rank - iter(x) - \alpha(n)x.rank$$
$$-level(x)x.rank - iter(x) \leq 0$$

y rimane radice e $y.rank$ o rimane invariato o aumenta di 1. Quindi:

$$\Delta\Phi_i(y) = \alpha(n)y.rank_i - \alpha(n)y.rank_{i-1} \leq \alpha(n)$$

Dunque $\Delta\Phi_i \leq \alpha(n)$ ed il costo ammortizzato di Link è:

$$\hat{c} = c + \Delta\Phi_i \leq 1 + \alpha(n) = O(\alpha(n))$$

8.6.3 FindSet(x)

Il costo reale è proporzionale al numero s di nodi del cammino percorso. Il potenziale della radice non cambia mentre i potenziali degli altri nodi possono solo diminuire.
Consideriamo dapprima come cambia il potenziale dei nodi x del cammino che hanno $x.rank > 0$ e che sono seguiti da almeno un nodo y diverso dalla radice e tale che $level(y) = level(x)$.
Sia $k = level(y) = level(x)$. Allora:

$$y.p.rank \geq A_k(y.rank) \geq A_k(x.p.rank) \geq A_k(A_k^{(iter(x))}(x.rank)) = A_k^{(iter(x)+1)}(x.ranl)$$

dopo la compressione:

$$x.p.rank = y.p.rank \geq A_k^{(iter(x)+1)}(x.rank)$$

Dunque $iter(x)$ aumenta di almeno 1 e quindi:

$$\Delta\Phi_i(x) \leq -1$$

I nodi rimanenti sono la radice, il primo nodo se ha rango 0 e per ogni livello k l'ultimo nodo del cammino avente $level(x) = k$. Siccome i livelli distinti sono al più $\alpha(n)$ ci sono al più $\alpha(n) + 2$ nodi il cui potenziale può rimanere invariato, mentre il potenziale degli altri $s - \alpha(n) - 2$ diminuisce di almeno 1. Quindi:

$$\Delta\Phi_i \leq -s + \alpha(n) + 2$$

ed il costo ammortizzato di FindSet è:

$$\hat{c} = c + \Delta\Phi_i \leq s - s + \alpha(n) + 2 = O(\alpha(n))$$

8.7 Esercizi

8.7.1 Esercizio 1

La funzione FindSet(x) ricerca il rappresentante in una struttura dati per insiemi disgiunti ed effettua la compressione dei cammini viene normalmente definita ricorsivamente come segue:

```
FindSet(x)
       if x.p ≠ x
              x.p = FindSet(x.p)
       return x.p
```

Scrivere una versione **non ricorsiva** efficiente di tale funzione e spiegare la differenza tra le compressioni dei cammini ottenute con le due versioni.

Soluzione:

```
FindSet(x)
       y = x
       while y.p ≠ y
              y = y.p
       // y Ã¨ ora il rappresentante
       z = x
       while z.p ≠ z
              w = z.p
              z.p = y
              z = w
       return z
```

8.7.2 Esercizio 2

La funzione FindSet(x) ricerca il rappresentante in una struttura dati per insiemi disgiunti ed effettua la compressione dei cammini viene normalmente definita ricorsivamente come segue:

```
FindSet(x)
       if x.p ≠ x
              x.p = FindSet(x.p)
       return x.p
```

La ricerca del rappresentante e la compressione del cammino si può effettuare anche utilizzando la seguente versione non ricorsiva:

```
FindSet(x)
       y = x
       ys = y.p
       while ys ≠ y
              y.p = ys.p
              y = ys
              ys = y.p
       return y
```

Entrambe le versioni effettuano una compressione dei cammini. Spiegare la differenza tra le compressioni dei cammini ottenute con le due versioni.

Soluzione: La differenza nella compressione dei cammini tra le due versioni è che quella ricorsiva collega tutti i nodi che percorre alla radice, comprimendo tutti i loro cammini, mentre la versione iterativa divide in due l'albero, collegando i nodi alternativamente tra loro. In questo modo, l'altezza dell'albero diventa $O(\lceil \frac{n+1}{2} \rceil)$, riducendo effettivamente la lunghezza dei cammini per arrivare al rappresentante.

8.7.3 Esercizio 3

Modificare la struttura dati foresta di insiemi disgiunti in modo da poter eseguire in modo efficiente, oltre alle operazioni MakeSet(x), Union(x,y) e FindSet(x), anche l'operazione NumSet(x) che restituisce il numero di elementi dell'insieme a cui appartiene x. Scrivere lo pseudo-codice di NumSet(x) e di MakeSet(x), Union(x,y) e FindSet(x) opportunamente modificate.

Soluzione: risolviamo l'esercizio aggiungendo un campo $x.n$ al rappresentante x dell'insieme.

```
MakeSet(x)
        x.p = x
        x.n = 1
```

```
Union(x,y)
        x = FindSet(x)
        y = FindSet(y)
        Link(x,y)
```

```
Link(x,y)
        if x.rank > y.rank
                y.p = x
                x.n = x.n+y.n
        else
                x.p = y
                y.n = x.n+y.n
                if x.rank == y.rank
                        y.rank = y.rank+1
```

```
NumSet(x)
        return FindSet(x).n
```

La funzione FindSet(x) non ha bisogno di modifiche rispetto all'originale.

8.7.4 Esercizio 4

Modificare la struttura dati foresta di insiemi disgiunti in modo da poter eseguire in modo efficiente, oltre alle operazioni MakeSet(x), Union(x,y) e FindSet(x), anche l'operazione MaxSet(x) che restituisce l'elemento con chiave massima dell'insieme a cui appartiene x. Scrivere lo pseudo-codice di NumSet(x) e di MakeSet(x), Union(x,y) e FindSet(x) opportunamente modificate.

Soluzione: risolviamo l'esercizio aggiungendo $x.max$ al rappresentante x dell'insieme.

```
MakeSet(x)
       x.p = x
       x.max = x
```

```
Union(x,y)
       x = FindSet(x)
       y = FindSet(y)
       Link(x,y)
```

```
Link(x,y)
       if x.rank > y.rank
              y.p = x
              if y.max.info > x.max.info
                     x.max = y.max
       else
              x.p = y
              if x.max.info > y.max.info
                     y.max = x.max
              if x.rank == y.rank
                     y.rank = y.rank+1
```

```
NumSet(x)
       return FindSet(x).max
```

La funzione FindSet non ha bisogno di modifiche rispetto all'originale.