Algoritmi e Strutture di Dati I

Diario di bordo

Massimo Franceschet
m.franceschet@unich.it,
http://www.sci.unich.it/∼francesc

24 marzo 2004

Indice

1 Presentazione del corso

- Docente:
  - Nome: Massimo Franceschet
  - Telefono: 085 4546432
  - E-mail: m.franceschet@unich.it
  - Pagina web: http://www.sci.unich.it/~francesc
- Finalità del corso: il corso si propone di studiare le strutture di dati e gli algoritmi di base.
- Programma del corso:
  1. Introduzione: Concetti di problema e di algoritmo. Decidibilità e complessità.
  2. Strutture di dati: Vettori , pile, code, liste, alberi radicati, alberi binari di ricerca, tabelle hash, heap.
  3. Algoritmi di ordinamento: Insertion Sort, Heap Sort, Quick Sort, Merge Sort, Counting Sort.

  Il programma corrisponde alle seguenti parti del libro adottato: Parte I, Capitoli 1, 2, 3, 4. Parte II, Capitoli 6, 7, 8. Parte III, Capitoli 10, 11, 12.
- Libro adottato: *Introduction to Algorithms*, di T. H. Cormen, C. E. Leiserson, R. L. Rivest e C. Stein, MIT Press, Second Edition, 2001 (In Inglese). Si consiglia l'acquisto in rete, ad esempio su http://www.amazon.com.
- Lezioni e ricevimento: le lezioni si terranno il Martedì dalle 13:45 alle 15:30 e il Giovedì dalle 8:45 alle 10:30. Il ricevimento si terrà il Mercoledì dalle 16 alle 18 nel mio ufficio nel Dipartimento di Scienze.
- Modalità di esame: Ci saranno 6 appelli scritti. Non ci saranno i parziali e neppure altri appelli di recupero. Lo scritto si basa su domande teoriche e procedure da implementare. Non sono ammessi appunti, dispense e libri durante lo scritto. Il voto minimo per il superamento dello scritto è 18. In caso di superamento dello scritto, l'orale è facoltativo. L'orale è invece obbligatorio per coloro che sono stati sospettati di copia o di aver favorito la copia durante lo scritto. L'orale verte su domande teoriche e procedure da implementare. A seconda dell'esito dell'orale, il voto dello scritto verrà modificato in positivo, in negativo, oppure rimarrà costante. Il voto finale viene registrato alla fine di ogni appello (normalmente il giorno dell'orale). Non sono ammesse registrazioni successive. Chi, senza un giustificato motivo, non si presenta alla registrazione automaticamente rifiuta il voto.

- Materiale didattico: dispense, lucidi, appelli precedenti con soluzioni, proposte di tesi e altro si trovano sul sito:

  http://www.sci.unich.it/~francesc/teaching

2 Introduzione

Un **problema** specifica in termini generali una *relazione* che intercorrere tra dei dati di ingresso (input) e dei dati di uscita (output).

Esempio 2.1 (*Problema dell'ordinamento*)

Il problema dell'ordinamento consiste nell'ordinare (in senso crescente) una sequenza di numeri: data in ingresso una sequenza $\langle a_1, a_2, \ldots, a_n \rangle$ *di* n *numeri, fornire in uscita una permutazione* $\langle a'_1, a'_2, \ldots, a'_n \rangle$ *della sequenza di ingresso tale che* $a'_1 \leq a'_2, \ldots, \leq a'_n$.

Un **algoritmo** è una *procedura computazionale* che prende un valore in **ingresso** (**input**) e restituisce un valore in **uscita** (**output**). Un algoritmo viene descritto mediante una sequenza di operazioni che trasformano i dati di ingresso nei dati di uscita. Un algoritmo è **corretto** rispetto ad un problema se, per ogni ingresso del problema, l'algoritmo termina fornendo l'uscita del problema. Se un algoritmo è corretto per un problema, diremo che l'algoritmo risolve il problema in questione.

Esercizio 2.1 (*Monte Carlo Sort*)

Consideriamo il seguente algoritmo per ordinare una sequenza S *di* n *numeri. L'algoritmo consiste di due passi. Il primo passo genera casualmente una permutazione* S' *di* S. *Il secondo passo verifica se* S' *è ordinata, e, se lo è, la restituisce in uscita terminando la procedura. Dire se l'algoritmo risolve il problema dell'ordinamento nei casi: (A) il generatore casuale di permutazioni ha memoria (quindi ogni permutazione generata è diversa dalle precedenti); (B) il generatore casuale di permutazioni non ha memoria (quindi può generare una permutazione precedentemente uscita)*

Soluzione

Nel caso (A) l'algoritmo risolve il problema dell'ordinamento. Infatti, dato che le permutazioni di n numeri sono un numero finito, prima o poi il generatore produce la sequenza ordinata, e quindi termina. Nel caso (B) l'algoritmo non risolve il problema dell'ordinamento. Poiché le permutazioni sono generate casualmente, e il generatore non ha memoria, è possibile che il generatore non produca mai la sequenza ordinata, e quindi che l'algoritmo non termini. Essendo la terminazione è una condizione necessaria per la correttezza, l'algoritmo in questo caso non è corretto.

Con il termine *programma* si intende la trascrizione (implementazione) dell'algoritmo in un linguaggio formale di programmazione (Pascal, C, Java, ...). Sia una algoritmo che un programma debbono essere specificati in modo preciso e non ambiguo. La differenza tra algoritmo e programma è la seguente: un algoritmo è una procedura computazionale intelligibile dall'uomo, un programma è una procedura computazionale comprensibile dalla macchina. Ne deriva che il linguaggio in cui scriviamo gli algoritmi è più astratto (più ad alto livello) del linguaggio di programmazione. Dato che un algoritmo può sempre essere tradotto in un programma, ci interessiamo solo agli algoritmi, e tralasciamo in questo corso i programmi.

Un problema è **decidibile** se esiste almeno un algoritmo che lo risolve. Esistono problemi indecidibili? Si, e parecchi. Se associamo un numero reale ad ogni problema, i problemi decidibili possono essere associati ai numeri naturali, mentre i problemi indecidibili possono essere associati ai rimanenti numeri reali. Quindi i problemi decidibili sono una rarità.

Esempio 2.2 (*Problema di corrispondenza di Post*)

Il problema di corrispondenza di Post è il seguente. Date due sequenze di n parole $A = \langle w_1, \dots, w_n \rangle$, $B = \langle v_1, \dots, v_n \rangle$, trovare, se esiste, una sequenza di indici $\langle i_1, \dots, i_m \rangle$, con $m \geq 1$, tale che

$$w_{i_1} \dots w_{i_m} = v_{i_1} \dots v_{i_m}.$$

Ad esempio, se $A = \langle 1, 10111, 10 \rangle$, e $B = \langle 111, 10, 0 \rangle$, la soluzione è $\langle 2, 1, 1, 3 \rangle$. Infatti

$$10111\,1\,1\,10 = 10\,111\,111\,0.$$

In generale, il problema in questione non si può risolvere algoritmicamente, cioè è indecidibile.

Esempio 2.3 (*Problema del domino*)

Il Problema del domino è il seguente. Dato un insieme finito di colori e un insieme di piastrelle quadrate con i quattro lati colorati di un qualche colore tra quelli a disposizione, trovare, se esiste, un modo per piastrellare una parete infinita in lunghezza e larghezza assicurando che i lati adiacenti delle piastrelle abbiano lo stesso colore. Anche questo problema è indecidibile.

Dato un problema, la prima cosa da fare è capire se è decidibile o meno. Nel caso sia indecidibile, non abbiamo speranza di trovare un algoritmo che lo risolva. Qualora sia decidibile, esiste almeno un algoritmo che lo risolve. In generale, esistono più algoritmi diversi che lo risolvono. Quale scegliere? Quello di costo (complessità) inferiore. La **complessità computazionale** di un algoritmo è la quantità di risorse che l'algoritmo richiede per terminare. La complessità computazionale di un problema è la complessità computazionale dell'algoritmo più efficiente che lo risolve. Quali risorse sono significative per il calcolo della complessità? **Tempo** e **spazio**. Per complessità temporale si intende il tempo che l'algoritmo impiega per terminare. Se vediamo un algoritmo come una sequenza di operazioni elementari, e associamo un tempo costante ad ogni operazione elementare, allora la complessità temporale è direttamente proporzionale al numero di operazioni elementari fatte dall'algoritmo per terminare. Per complessità spaziale si intende la quantità di spazio (cioè di memoria) utilizzata dall'algoritmo durante l'elaborazione.

Esercizio 2.2 *Argomentare che la complessità spaziale è non superiore alla complessità temporale.*

Soluzione

Ogni calcolatore è basato su una architettura chiamata *macchina di Von Neumann*. Tale macchina possiede una memoria (per archiviare i dati), un processore (per elaborare i dati), dei collegamenti tra memoria e processore (per leggere e scrivere dati dalla/in memoria) e delle periferiche di input/output (per consentire all'utente di inserire/leggere le informazioni). La memoria è suddivisa in unità elementari chiamate celle, che possono essere lette e scritte. Ad ogni scrittura su una cella di memoria, corrisponde una operazione elementare, e quindi un costo temporale. Inoltre, lo spazio di memoria è riutilizzabile, il tempo no (una volta passato, è perso per sempre). Ne deriva che non posso usare memoria senza perdere tempo. Quindi il tempo che un algoritmo impiega è sempre non inferiore alla memoria utilizzata.

La complessità di un dato algoritmo è funzione della **dimensione** e della **forma** dei dati in ingresso. Prendiamo l'esempio 2.1 del problema di ordinamento di una sequenza di numeri. Ci aspettiamo che più è lunga la sequenza, più risorse l'algoritmo impieghi per risolvere il problema. Inoltre, ci aspettiamo che, a parità di lunghezza, più grande è il numero di coppie ordinate in partenza nella sequenza, meno risorse l'algoritmo impieghi per terminare. Al fine di rendere univoca la definizione di complessità, esprimiamo la complessità in funzione della dimensione dell'ingresso. Inoltre, fissata la dimensione dell'input, consideriamo tre tipi di complessità:

- **complessità ottima**: è la complessità dell'algoritmo nel caso migliore. Cioè, scelgo la più piccola tra tutte le complessità generate dall'algoritmo sugli input di una certa dimensione;
- **complessità media**: è la complessità dell'algoritmo nel caso medio. Cioè, faccio la media tra tutte le complessità generate dall'algoritmo sugli input di una certa dimensione;
- **complessità pessima**: è la complessità dell'algoritmo nel caso peggiore. Cioè, scelgo la più grande tra tutte le complessità generate dall'algoritmo sugli input di una certa dimensione.

Si noti che la complessità ottima è non superiore a quella media che è non superiore a quella pessima. Solitamente, con il termine complessità computazionale, ci si riferisce alla complessità temporale pessima.

Esercizio 2.3 *Si riconsideri l'algoritmo di ordinamento Monte Carlo (esempio 2.1). Supponendo che esista una procedura che verifica se una sequenza è ordinata con costo unitario, calcolare nei casi (A) e (B) la complessità ottima, media e pessima dell'algoritmo.*

Soluzione

Sia n la lunghezza della sequenza da ordinare. Esistono quindi $n!$ possibili permutazioni della sequenza.

Caso (A) Il generatore di permutazioni ha memoria quindi non estrae mai una permutazione precedentemente estratta. Il caso migliore si ha quando la prima permutazione estratta è quella ordinata. In questo caso l'unico costo è quello dovuto alla verifica che la sequenza è ordinata, cioè 1. La complessità ottima è dunque unitaria. In media, la permutazione ordinata viene estratta dopo metà delle estrazioni. Quindi la complessità media risulta $n!/2$. Nel caso pessimo, la permutazione ordinata viene estratta per ultima. Quindi la complessità pessima è $n!$.

Caso (B) Il generatore di permutazioni non ha memoria quindi può estrarre una permutazione precedentemente estratta. Il caso migliore non cambia e si ha quando la prima permutazione estratta è quella ordinata. Quindi la complessità ottima è unitaria. Nel caso medio, sia X una *variabile casuale* tale che $X = i$ se la permutazione ordinata viene estratta alla i-esima estrazione, con $i \geq 1$. La complessità media è dunque il valor medio della variabile casuale X. Il valor medio è

$$\mu(X) = \sum_{i=1}^{\infty} i \Pr\{X = i\}.$$

Dato che

$$\Pr\{X = i\} = (1 - 1/n!)^{i-1} 1/n!,$$

abbiamo che

$$\mu = \sum_{i=1}^{\infty} i(1 - 1/n!)^{i-1} 1/n! = (1/n!)(1 - 1/n!)^{-1} \sum_{i=1}^{\infty} i(1 - 1/n!)^{i}.$$

Poiché $\sum_{i=1}^{\infty} ix^i = x/(1-x)^2$, abbiamo che

$$\mu = (1/n!)(1 - 1/n!)^{-1}(1 - 1/n!)(n!)^2 = n!$$

Dunque la complessità media è pari a $n!$. La complessità pessima è infinito, in quanto, nel caso peggiore, la sequenza ordinata non viene mai generata.

2.1 Limiti asintotici

Siamo interessati a misurare le differenze significative di complessità tra gli algoritmi. Se la differenza di complessità tra due algoritmi non è significativa, allora consideriamo i due algoritmi computazionalmente equivalenti. Come detto, la complessità è funzione della dimensione dei dati di ingresso. Assumiamo senza perdita di generalità che la dimensione dell'input sia un numero naturale e il valore della complessità sia anche un numero naturale. Quindi la funzione di complessità sarà una funzione $f(n)$ da $\mathbb{N}$ in $\mathbb{N}$. Useremo frequentemente le seguenti **notazioni asintotiche**.

- **Limite superiore asintotico** O. Date due funzioni $f(n)$ e $g(n)$, scriveremo
$$f(n) = O(g(n))$$
qualora esistano $c > 0$ e $n_0 \geq 0$ tali che
$$f(n) \leq cg(n)$$
per ogni $n \geq n_0$;

- **Limite inferiore asintotico** Ω. Date due funzioni $f(n)$ e $g(n)$, scriveremo
$$f(n) = \Omega(g(n))$$
qualora esistano $c > 0$ e $n_0 \geq 0$ tali che
$$f(n) \geq cg(n)$$
per ogni $n \geq n_0$;

- **Limite asintotico stretto** Θ. Date due funzioni $f(n)$ e $g(n)$, scriveremo
$$f(n) = \Theta(g(n))$$
qualora esistano $c_1, c_2 > 0$ e $n_0 \geq 0$ tali che
$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$
per ogni $n \geq n_0$.

Si noti che $f(n) = \Theta(g(n))$ se e solo se $f(n) = \Omega(g(n))$ e $f(n) = O(g(n))$.

Date due funzioni $f(n)$ e $g(n)$, diremo che le funzioni $f(n)$ e $g(n)$ sono asintoticamente equivalenti, scritto $f(n) \sim g(n)$, se $f(n) = \Theta(g(n))$. Diremo che $f(n)$ è asintoticamente inferiore a $g(n)$, scritto $f(n) \prec g(n)$, se $f(n) = O(g(n))$ e $f(n) \neq \Omega(g(n))$.

Esercizio 2.4 *Dimostrare che:*

- $\sim$ *è una relazione di equivalenza; e*
- $\prec$ *è una relazione di ordinamento stretto.*

Soluzione

- Occorre mostrare che $\sim$ è riflessiva (cioè $f(n) \sim f(n)$), simmetrica (cioè se $f(n) \sim g(n)$ allora $g(n) \sim f(n)$) e transitiva (cioè se $f(n) \sim g(n)$ e $g(n) \sim h(n)$ allora $f(n) \sim h(n)$).
- Occorre mostrare che $\prec$ è irriflessiva (cioè $f(n) \not\prec f(n)$) e transitiva.

Di seguito è indicata una scala asintotica crescente di complessità:

$$\log n \prec \sqrt[p]{n} \prec n \log n \prec n^k \prec 2^n \prec n! \prec n^n \prec 2^{2^n},$$

ove $p \geq 1$ e $k > 1$. Diremo che un algoritmo ha *complessità polinomiale* se $f(n) = \Theta(n^k)$, con $k \geq 1$, ed ha complessità (singolarmente) esponenziale se $f(n) = \Theta(2^n)$. Solitamente, un algoritmo è efficiente (o un problema è facile) se la sua complessità è al massimo polinomiale, un algoritmo è inefficiente (o un problema è difficile) se la sua complessità è almeno esponenziale.

Esempio 2.4 (*Problema del cammino minimo*)

Abbiamo un insieme di città, alcune delle quali sono collegate da vie percorribili. Ogni collegamento è associato ad un costo (ad esempio il tempo di percorrenza). Il problema del cammino minimo consiste nel trovare il cammino di costo minimo tra due città.

Esempio 2.5 (*Problema del commesso viaggiatore*)

Abbiamo un insieme di città, collegate a due a due da vie percorribili. Ad ogni collegamento è associato un costo (ad esempio il tempo di percorrenza). Il problema del commesso viaggiatore consiste nel trovare un ciclo di costo minimo che passa esattamente una volta per ogni città.

Il problema del cammino minimo è facile: la sua complessità è polinomiale. Il problema del commesso viaggiatore è difficile: non si conosce ad oggi un algoritmo che lo risolve con complessità polinomiale, quindi si congettura che abbia complessità esponenziale.

Esercizio 2.5 *Mostrare che:*

1. $c = \Theta(1)$.
2. $2n^2 + 1 = \Theta(n^2)$.
3. $2n \neq \Theta(n^2)$.
4. $2^{n+1} = \Theta(2^n)$.
5. $2^{2n} \neq \Theta(2^n)$
6. $\log_b f(n) = \Theta(\log f(n))$

Soluzione

1. Infatti $c1 \leq c \leq c1$.
2. Infatti $n^2 \leq 2n^2 + 1 \leq 3n^2$ per ogni $n \geq 1$. In generale vale che

   $$a_0 n^k + a_1 n^{k-1} + \ldots a_k = \Theta(n^k),$$

   qualora $a > 0$. Quindi, ogni polinomio di grado k con coefficiente positivo è asintoticamente equivalente a n^k.

3. Infatti $2n = O(n^2)$ ma $2n \neq \Omega(n^2)$. Supponiamo per assurdo che esistano $c > 0$ e $n_0 \geq 0$ tali che $2n \geq cn^2$ per ogni $n \geq n_0$. Quindi $2/c \geq n$ per ogni $n \geq n_0$. Questo è assurdo.

4. Infatti $2^n \leq 2^{n+1} \leq 22^n$ per ogni $n \geq 0$. In generale vale che $2^{n+k} = \Theta(2^n)$ per ogni $k \geq 0$.

5. Infatti $\log_b f(n) = \frac{\log f(n)}{\log b} = \Theta(\log f(n))$

6. Infatti $2^{2n} \neq O(2^n)$. Supponiamo per assurdo che esistano $c > 0$ e $n_0 \geq 0$ tali che $2^{2n} \leq c2^n$ per ogni $n \geq n_0$. Dividendo per 2^n entrambi i membri abbiamo che $2^n \leq c$ per ogni $n \geq n_0$, il che è assurdo.

Quindi, le costanti additive e moltiplicative sono asintoticamente irrilevanti se applicate alla base della funzione. Inoltre, le costanti additive ad esponente sono asintoticamente irrilevanti, ma le costanti moltiplicative ad esponente sono significative.

Esercizio 2.6 *Mostrare che:*

1. *se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ *allora* $f(n) = O(g(n))$ *e* $f(n) \neq \Omega(g(n))$ *(cioè* $f(n) \prec g(n)$*, ossia* $f(n)$ *è asintoticamente inferiore a* $g(n)$*).*

2. *se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = l$*, con* l *un numero diverso da zero, allora* $f(n) = \Theta(g(n))$ *(cioè* $f(n) \sim g(n)$*, ossia* $f(n)$ *e* $g(n)$ *sono asintoticamente equivalenti).*

Soluzione

1. Per definizione di limite, abbiamo che $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ se e solo se per ogni $c > 0$, esiste $n_0 \geq 0$ tale che $|\frac{f(n)}{g(n)} - 0| \leq c$, per ogni $n \geq n_0$, cioè, dato che entrambe le funzioni sono positive, $f(n) \leq cg(n)$, per ogni $n \geq n_0$. Ciò implica che esistono $c > 0$ e $n_0 \geq 0$ tali che $f(n) \leq cg(n)$, per ogni $n \geq n_0$, ossia $f(n) = O(g(n))$. Mostriamo che $f(n) \neq \Omega(g(n))$. Per assurdo, supponiamo il contrario, quindi che esistano $c_0 > 0$ e $n_1 \geq 0$ tali che $c_0 g(n) \leq f(n)$ per ogni $n \geq n_1$. Dato che per ogni $c > 0$, esiste $n_0 \geq 0$ tale che $f(n) \leq cg(n)$, in particolare vale che esiste $n_0 \geq 0$ tale che $f(n) \leq c_0 g(n)$ per ogni $n \geq n_0$. Quindi, abbiamo che $f(n) \leq c_0 g(n)$ da un cento punto in poi, e, allo stesso tempo, $c_0 g(n) \leq f(n)$ da un certo punto in poi. Quindi, l'unica caso possibile è che $f(n) = c_0 g(n)$. Ma se cosí fosse, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c_0 \neq 0$. Assurdo.

2. Per definizione di limite, abbiamo che $\lim_{n\to\infty} \frac{f(n)}{g(n)} = l$ se e solo se per ogni $c > 0$, esiste $n_0 \geq 0$ tale che $|\frac{f(n)}{g(n)} - l| \leq c$, cioè $-c \leq \frac{f(n)}{g(n)} - l \leq c$, per ogni $n \geq n_0$. Dunque, per ogni $c > 0$ esiste $n_0 \geq 0$ tale che $(-c+l)g(n) \leq f(n) \leq (c+l)g(n)$ per ogni $n \geq n_0$. Quindi, esistono $c_1, c_2 > 0$ e $n_0 \geq 0$ tali che $c_1 g(n) \leq f(n) \leq c_2 g(n)$ per ogni $n \geq n_0$. Basta porre $c_1 = -c + l$ e $c_2 = c + l$, con $-l < c < l$. Perciò $f(n) = \Theta(g(n))$.

In definitiva, abbiamo imparato che, date due funzioni $f(n)$ e $g(n)$, possono accadere le seguenti cose:

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. In tal caso $f(n) \prec g(n)$;
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$. In tal caso $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$ e quindi $g(n) \prec f(n)$;
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = l \neq 0$. In tal caso $f(n) \sim g(n)$;
- $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ non esiste. In tal caso, $f(n)$ e $g(n)$ non sono confrontabili. Ad esempio, n e $n^{1+\sin n}$ non sono confrontabili.

Quindi $\prec$ è una relazione di ordinamento stretto *parziale*.

Esercizio 2.7 *Mostrare che* $\log(n!) = \Theta(n \log n)$.

Soluzione

Mostriamo che
$$\lim_{n\to\infty} \frac{\log(n!)}{n \log n} = 1,$$
e dunque la tesi per l'Esercizio 2.6. Per la formula di Stirling, abbiamo che
$$n! = \sqrt{2\pi n}(n/e)^n e^{\alpha_n},$$
ove $1/(12n+1) < \alpha_n < 1/(12n)$. Dunque
$$\log(n!) = \log \sqrt{2\pi n} + n \log(n/e) + \alpha_n \log e.$$
Abbiamo che
$$\lim_{n\to\infty} \frac{\log \sqrt{2\pi n}}{n \log n} = 0,$$
$$\lim_{n\to\infty} \frac{n \log(n/e)}{n \log n} = 1,$$
$$\lim_{n\to\infty} \frac{\alpha_n \log e}{n \log n} = 0,$$
e dunque la tesi.

Esercizio 2.8 *Dimostrare che:*

1. $\Theta(f(n)) + \Theta(g(n)) = \Theta(f(n) + g(n))$;
2. $\Theta(f(n))\Theta(g(n)) = \Theta(f(n)g(n))$;
3. $\Theta(2^{f(n)}) \neq 2^{\Theta(f(n))}$.

Soluzione

1. $\Theta(f(n)) + \Theta(g(n)) = \Theta(f(n) + g(n))$. Siano $f'(n)$ e $g'(n)$ tali che $f'(n) = \Theta(f(n))$ e $g'(n) = \Theta(g(n))$. Occorre mostrare che $f'(n)+g'(n) = \Theta(f(n)+g(n))$. Abbiamo che esistono $c_1, c_2 > 0$ e $n_0 \geq 0$ tali che $c_1 f(n) \leq f'(n) \leq c_2 f(n)$ per ogni $n \geq n_0$, e inoltre esistono $c_3, c_4 > 0$ e $n_1 \geq 0$ tali che $c_3 g(n) \leq g'(n) \leq c_4 g(n)$ per ogni $n \geq n_1$. Dunque $min\{c_1, c_3\}(f(n) + g(n)) \leq f'(n)+g'(n) \leq max\{c_2, c_4\}(f(n)+g(n))$ per ogni $n \geq max\{n_0, n_1\}$. Dunque $f'(n) + g'(n) = \Theta(f(n) + g(n))$.

2. $\Theta(f(n))\Theta(g(n)) = \Theta(f(n)g(n))$. Siano $f'(n)$ e $g'(n)$ tali che $f'(n) = \Theta(f(n))$ e $g'(n) = \Theta(g(n))$. Occorre mostrare che $f'(n)g'(n) = \Theta(f(n)g(n))$. Abbiamo che esistono $c_1, c_2 > 0$ e $n_0 \geq 0$ tali che $c_1 f(n) \leq f'(n) \leq c_2 f(n)$ per ogni $n \geq n_0$, e inoltre esistono $c_3, c_4 > 0$ e $n_1 \geq 0$ tali che $c_3 g(n) \leq g'(n) \leq c_4 g(n)$ per ogni $n \geq n_1$. Dunque $c_1 c_3 f(n)g(n) \leq f'(n)g'(n) \leq c_2 c_4 f(n)g(n)$ per ogni $n \geq max\{n_0, n_1\}$. Dunque $f'(n)g'(n) = \Theta(f(n)g(n))$.

3. $\Theta(2^{f(n)}) \neq 2^{\Theta(f(n))}$. Sia $f(n) = n$ e $f'(n) = 2n = \Theta(n) = \Theta(f(n))$. Occorre mostrare che $2^{2n} \neq \Theta(2^n)$. Si veda l'esercizio 2.5.

Ad esempio, $\Theta(n) + \Theta(1) = \Theta(n + 1) = \Theta(n)$, e $\Theta(n) + \Theta(n) = \Theta(n + n) = \Theta(2n) = \Theta(n)$. Inoltre $\Theta(n)\Theta(1) = \Theta(n1) = \Theta(n)$, e $\Theta(n) + \Theta(n) = \Theta(nn) = \Theta(n^2)$.

2.2 Equazioni ricorsive

In generale, il programma che implementa un algoritmo consiste di più procedure. Ogni procedura più invocare altre procedure che fanno parte del programma. Una **procedura ricorsiva** è una procedura che richiama se stessa. Vediamo ora come calcolare la complessità di una procedura ricorsiva. Le procedure ricorsive spesso implementano una tecnica di programmazione nota come *dividi e conquista*. Tale tecnica consiste di tre passi:

- *dividi* il problema principale in sottoproblemi;
- *conquista*, cioè risolvi, i sottoproblemi;
- *combina* i risultati dei sottoproblemi ottenendo un risultato per il problema principale.

La complessità di una procedura ricorsiva dividi e conquista può spesso essere espressa da una **equazione ricorsiva** del tipo

$$C(n) = aC(n/b) + f(n),$$

ove $a \geq 1$ and $b > 1$, e per n/b si intende $\lfloor n/b \rfloor$ oppure $\lceil n/b \rceil$. L'equazione descrive la complessità di una procedura che divide un problema di dimensione n in a sottoproblemi ognuno di dimensione n/b. Il costo della divisione del problema e della combinazione delle soluzioni dei sottoproblemi è $f(n)$.

La soluzione della precedente equazione ricorsiva può essere calcolata mediante il seguente teorema, noto come Master Theorem:

Teorema 2.1 (Master Theorem)

Siano $a \geq 1$ e $b > 1$ due costanti intere, e $f(n)$ una funzione di complessità. Sia $C(n) = aC(n/b) + f(n)$ una equazione ricorsiva definita sui naturali, ove per n/b si intende $\lfloor n/b \rfloor$ oppure $\lceil n/b \rceil$. Allora:

1. *Se $f(n) = O(n^{\log_b a - \epsilon})$ per qualche $\epsilon > 0$, allora $C(n) = \Theta(n^{\log_b a})$.*
2. *Se $f(n) = \Theta(n^{\log_b a})$, allora $C(n) = \Theta(n^{\log_b a} \log n)$.*
3. *Se $f(n) = \Omega(n^{\log_b a + \epsilon})$ per qualche $\epsilon > 0$, e se esiste una costante $c < 1$ tale che $af(n/b) \leq cf(n)$ per tutti gli n sufficientemente grandi, allora $C(n) = \Theta(f(n))$.*

Esercizio 2.9 *Risolvere le seguenti equazioni ricorsive usando il Master Theorem:*

1. *$C(n) = 2C(n/2) + \Theta(1)$.*
2. *$C(n) = C(n/2) + \Theta(1)$.*
3. *$C(n) = 2C(n/2) + \Theta(n)$.*
4. *$C(n) = C(n/2) + \Theta(n)$.*

Soluzione

1. se $C(n) = 2C(n/2) + \Theta(1)$, allora $a = 2$, $b = 2$, $f(n) = \Theta(n^{\log_2 2 - \epsilon}) = \Theta(1)$, ponendo $\epsilon = 1$. Dunque il caso 1 del Master Theorem si applica, e quindi $C(n) = \Theta(n)$.
2. se $C(n) = C(n/2) + \Theta(1)$, allora $a = 1$, $b = 2$, $f(n) = \Theta(n^{\log_2 1}) = \Theta(1)$. Dunque il caso 2 del Master Theorem si applica, e quindi $C(n) = \Theta(\log n)$.
3. se $C(n) = 2C(n/2) + \Theta(n)$, allora $a = 2$, $b = 2$, $f(n) = \Theta(n^{\log_2 2}) = \Theta(n)$. Dunque il caso 2 del Master Theorem si applica, e quindi $C(n) = \Theta(n \log n)$.
4. se $C(n) = C(n/2) + \Theta(n)$, allora $a = 1$, $b = 2$, $f(n) = \Theta(n^{\log_2 1 + \epsilon}) = \Theta(n)$, con $\epsilon = 1$. Inoltre, $f(n/2) \leq cf(n)$, per una opportuna $c < 1$, in quanto $f(n)$ è lineare. Dunque il caso 3 del Master Theorem si applica, e quindi $C(n) = \Theta(f(n)) = \Theta(n)$.

Altre volte l'equazione ricorsiva assume forme diverse per le quali il Master Theorem non è applicabile. In alcuni casi, è possibile procedere per *induzione matematica*.

Esercizio 2.10 *Risolvere le seguenti equazioni ricorsive usando l'induzione matematica:*

1. *$C(0) = 1$ e, per $n > 0$, $C(n) = C(a) + C(b) + 1$ con $a, b \geq 0$ e $a + b = n - 1$;*

2. $C(0) = 1$ *e, per* $n > 0$, $C(n) = C(n-1) + 1$*;*

3. $C(0) = 1$ *e, per* $n > 0$, $C(n) = C(n-1) + (n+1)$*;*

Soluzione

1. Mostriamo che $C(n) = 2n + 1$, e dunque $C(n) = \Theta(n)$. Se $n = 0$, allora $C(n) = 1 = 2n + 1$. Se $n > 0$, allora, applicando l'ipotesi induttiva, $C(n) = 2a + 1 + 2b + 1 + 1 = 2(a + b + 1) + 1 = 2n + 1$.

2. Mostriamo che $C(n) = n + 1$, e dunque $C(n) = \Theta(n)$. Se $n = 0$, allora $C(n) = 1 = n + 1$. Se $n > 0$, allora, applicando l'ipotesi induttiva, $C(n) = C(n-1) + 1 = n + 1$.

3. Mostriamo che $C(n) = (n+1)(n+2)/2$, e dunque $C(n) = \Theta(n^2)$. Se $n = 0$, allora $C(0) = 1 = (n+1)(n+2)/2$. Se $n > 0$, allora, applicando l'ipotesi induttiva, $C(n) = C(n-1)+(n+1) = n(n+1)/2+(n+1) = (n+1)(n+2)/2$.

Altre volte ancora è conveniente usare il *metodo dell'albero di ricorsione*. L'albero di ricorsione è associato ad una equazione ricorsiva ben fondata (cioè che termina). Ogni nodo dell'albero viene etichettato con una istanza della equazione ricorsiva con i corrispondenti parametri, e due nodi n e m sono collegati da un arco da n verso m se l'istanza della equazione che etichetta m è stata generate dall'istanza della equazione che etichetta n. Si noti che il grafo costruito in questo modo è un albero in quanto l'equazione ricorsiva è ben fondata. Il particolare, le foglie dell'albero sono etichettate con istanze di equazioni che vengono risolte atomicamente (cioè senza ulteriori passi ricorsivi). Se associamo ad ogni nodo il costo della corrispondente istanza della equazione ricorsiva, è possibile calcolare la complessità della equazione ricorsiva sommando tali costi su tutti i nodi.

Esercizio 2.11 *Si risolva usando il metodo dell'albero di ricorsine la seguente equazione ricorsiva:* $C(n, n) = C(n, 0) = 1$*, e, per* $n \neq k$ *e* $k > 0$, $C(n, k) = C(n-1, k) + C(n-1, k-1) + 1$.

Soluzione

L'albero di ricorsione è il seguente: la radice è etichettata con $C(n, k)$ e un generico nodo etichettato con $C(i, j)$ ha due figli, uno etichettato con $C(i-1, j)$ e l'altro con $C(i-1, j-1)$. Le foglie hanno etichetta $C(i, i)$ oppure $C(i, 0)$, per qualche $i \leq n$. Si noti che l'albero è completo fino al livello $m = min\{k, n-k\}$. Dunque l'albero di ricorsione contiene almeno $2^{m+1} - 1$ nodi. Il costo associato ad ogni nodo è unitario. La soluzione è dunque $C(n, k) = \Omega(2^m) = \Omega(2^{min\{k,n-k\}})$. Nel caso peggiore, $n - k = k$, cioè $k = n/2$. In tal caso $C(n, k) = \Omega(2^{n/2})$.

3 Pseudocodice: sintassi e semantica

Una *variabile* è una locazione di memoria che può essere letta e scritta. Una *costante* è una locazione di memoria che può essere solo letta. Quindi una costante non può modificare il proprio contenuto. Useremo le seguenti costanti: NIL, OVERFLOW, UNDERFLOW, TRUE, FALSE.

Nelle *espressioni aritmetiche* useremo le seguenti operazioni: $+$ (addizione), $-$ (sottrazione), $*$ (moltiplicazione), $/$ (divisione esatta), *div* (divisione intera), *mod* (resto della divisione intera). Ad esempio $7\ div\ 4 = 1$ e $7\ mod\ 4 = 3$. Inoltre, dato un numero reale r, $\lceil r \rceil$ è il più piccolo intero maggiore o uguale a r, e $\lfloor r \rfloor$ è il più grande intero minore o uguale a r. Ad esempio, $\lceil 5 \rceil = \lfloor 5 \rfloor = 5$, $\lceil 5/2 \rceil = 3$ e $\lfloor 5/2 \rfloor = 2$. Infine useremo i seguenti operatori di confronto con l'usuale interpretazione: $<$, $\leq$, $>$, $\geq$, $=$, $\neq$.

Nelle *formule logiche Booleane* useremo i valori di verità TRUE (vero) e FALSE (falso) e i connettivi logici **and** (congiunzione), **or** (disgiunzione) e **not** (negazione). Se A, B sono proposizioni, la proposizione A **and** B è vera se e solo se entrambe A e B sono vere. La proposizione A **or** B è vera se e solo se almeno una tra A e B è vera. Infine **not** A è vera se e solo se A è falsa.

Useremo i seguenti costrutti di programmazione:

- *Assegnamento.* Sia x una variabile e y una variabile o una costante. La sintassi dell'assegnamento è:

  $$x \leftarrow y$$

  La semantica è la seguente: assegno alla variabile x il contenuto di y. La variabile x viene scritta, mentre y viene letta.

- *Condizionale.* Siano C una formula logica Booleana (o semplicemente condizione Booleana), B_1 e B_2 sottoprogrammi. La sintassi del condizionale è:

  if C **then** B_1 **else** B_2 **end if**

  Le semantica è la seguente: verifico la condizione C. Se C è vera, eseguo B_1, altrimenti eseguo B_2. Il ramo *else* può essere omesso.

- *Iterazione.*
  - *While.* Siano C una condizione Booleana e B un sottoprogramma. La sintassi del *while* è la seguente:

    while C **do** B **end while**

    La semantica è la seguente: eseguo il blocco B fino a che la condizione C diventa falsa.
  - *Repeat.* Siano C una condizione Booleana e B un sottoprogramma. La sintassi del *repeat* è la seguente:

    repeat B **until** C

    La semantica è la seguente: eseguo una volta il blocco B e poi lo eseguo ancora fino a che la condizione C diventa vera.

- *For*. Siano x una variabile, n e m due numeri interi, e B un sottoprogramma. La sintassi del *for* è la seguente:

  for $x \leftarrow n$ **to** m **do** B **end for**

  La semantica è la seguente: x assume i valori da n a m in ordine crescente e per ogni valore assunto da x eseguo una volta il blocco B. Se sostituiamo il **to** con un **downto**, la variabile x assume i valori da n a m in ordine decrescente.

Inoltre, useremo le seguenti ulteriori istruzioni:

- *Ritorno*. Sia x una variabile o una costante. L'istruzione

  return x

  ha l'effetto di terminare la procedura in corso e restituire alla procedura chiamante il contenuto di x.

- *Errore*. Sia x una variabile o una costante. L'istruzione

  error x

  ha l'effetto di terminare l'intero programma e restituire al sistema operativo il contenuto di x.

- *Stampa*. Sia x una variabile o una costante. L'istruzione

  print x

  ha l'effetto di stampare il contenuto di x.

4 Strutture di dati

Per risolvere un problema, un algoritmo rappresenta i valori di ingresso, l'informazione intermedia e i valori d'uscita mediante strutture di dati. Una **struttura di dati** consiste di un insieme di attributi e un insieme di **operazioni**. Gli attributi sono valori che caratterizzano la struttura (ad esempio, dimensione, inizio, fine,...). Le operazioni sono procedure che manipolano la struttura (ricerca, inserimento, cancellazione, ...).

In questa sezione vedremo diverse strutture di dati per rappresentare **insiemi dinamici** di elementi o **oggetti**. Un insieme dinamico è un insieme che può variare la propria cardinalità, inserendo nuovi elementi o cancellando elementi presenti nell'insieme. Assumeremo che gli elementi dell'insieme possano essere totalmente ordinati. Siamo interessati alle seguenti operazioni su insiemi dinamici:

- **Inserimento**: l'operazione inserisce un nuovo elemento nell'insieme;
- **Cancellazione**: l'operazione cancella un elemento dall'insieme;
- **Ricerca**: l'operazione verifica la presenza di un elemento in un insieme;
- **Massimo**: l'operazione trova il massimo dell'insieme;
- **Minimo**: l'operazione trova il minimo dell'insieme;
- **Successore**: l'operazione trova, se esiste, il successore di un elemento dell'insieme;
- **Predecessore**: l'operazione trova, se esiste, il prececessore di un elemento dell'insieme.

Inserimento e cancellazione vengono dette **operazioni di modifica**, e le altre **operazioni di interrogazione**. Un **dizionario** è un insieme dinamico che supporta le operazioni di modifica e l'operazione di ricerca.

4.1 Vettori

La struttura di dati **vettore** (**array** in inglese) rappresenta un insieme di dimensione prefissata. Dunque rappresenta un insieme statico, cioè che non può modificare la sua cardinalità. Assumeremo che gli oggetti rappresentati siano semplicemente numeri. Sia n è la lunghezza della sequenza. Gli elementi del vettore sono indicizzati con i numeri naturali da 1 a n. La struttura di dati vettore possiede l'attributo $length(A)$ che contiene la dimensione del vettore A, e l'operazione $A[i]$ che ritorna l' i-esimo elemento del vettore A. L'accesso agli elementi del vettore è **diretto**: posso accedere all'elemento i-esimo con una sola operazione senza dover passare per gli elementi che lo precedono. Il costo dell'operazione $A[i]$ è dunque costante, indipendentemente dall'indice i. Posso solo leggere un elemento del vettore mediante l'istruzione $x \leftarrow A[i]$ e scrivere un elemento del vettore mediante l'istruzione $A[i] \leftarrow x$. Come detto, un vettore rappresenta un insieme statico. Ciò significa che non posso ne inserire nuovi elementi nel vettore, ne cancellare elementi dal vettore. Dunque sui vettori solo le operazione di interrogazione possono essere implementate.

Esercizio 4.1 *Sia A un vettore, i e j due indici di A. Scrivere una procedura $Exchange(A, i, j)$ che scambia gli elementi di A indicizzati con i e j.*

Soluzione

Algoritmo 1 Exchange(A,i,j)

Exchange(A,i,j)

1: $t \leftarrow A[i]$
2: $A[i] \leftarrow A[j]$
3: $A[j] \leftarrow t$

La complessità della procedura *Exchange* è costante, cioè $\Theta(1)$.

Esercizio 4.2 *Scrivere una procedura Exchange senza usare variabili temporanee.*

Soluzione

Algoritmo 2 Exchange(A,i,j)

Exchange(A,i,j)

$A[i] \leftarrow A[i] + A[j]$
$A[j] \leftarrow A[i] - A[j]$
$A[i] \leftarrow A[i] - A[j]$

La complessità della procedura *Exchange* è costante, cioè $\Theta(1)$.

Esercizio 4.3 *Scrivere una procedura ArrayEmpty(A) che verifica se il vettore A è vuoto.*

Soluzione

Algoritmo 3 ArrayEmpty(A)

ArrayEmpty(A)

```
if length(A) = 0 then
    return TRUE
else
    return FALSE
end if
```

La complessità della procedura *Exchange* è costante, cioè $\Theta(1)$.

Esercizio 4.4 *Sia A un vettore e x un numero intero. Scrivere una procedura $ArraySearch(A, x)$ che ritorna l'indice dell'elemento che contiene x, oppure* NIL *se tale elemento non esiste.*

Soluzione

Algoritmo 4 ArraySearch(A,x)

ArraySearch(A,x)

```
i ← 1
while i ≤ length(A)  and  A[i] ≠ x do
   i ← i + 1
end while
if i ≤ length(A) then
   return  i
else
   return  NIL
end if
```

La complessità della procedura $ArraySearch(A, x)$ è lineare nella dimensione del vettore A, cioè vale $\Theta(n)$, con $n = length(A)$.

Esercizio 4.5 *Scrivere una procedura $ArrayMax(A)$ che ritorna l'indice dell'elemento massimo di A, oppure* NIL *se il vettore A è vuoto.*

Soluzione

Algoritmo 5 ArrayMax(A)

ArrayMax(A)

```
if ArrayEmpty(A) then
   return NIL
end if
m ← 1
for i ← 2 to length(A) do
   if A[i] > A[m] then
      m ← i
   end if
end for
return  m
```

La complessità della procedura $ArrayMax(A)$ è $\Theta(n)$, con $n = length(A)$.

Esercizio 4.6 *Scrivere una procedura $ArrayMin(A)$ che ritorna l'indice dell'elemento minimo di A, oppure* NIL *se il vettore A è vuoto.*

Esercizio 4.7 *Dato un vettore A e un suo elemento x, l'elemento successore di x è il più piccolo elemento di A che è più grande di x. Sia A un vettore e i un suo indice. Scrivere una procedura $ArraySuccessor(A, i)$ che ritorna l'indice dell'elemento successore di $A[i]$, oppure* NIL *se $A[i]$ è il massimo di A.*

Soluzione

Algoritmo 6 ArraySuccessor(A,i)

ArraySuccessor(A,i)

```
s ← ArrayMax(A)
if A[i] = A[s] then
   return NIL
end if
for j ← 1 to length(A) do
   if A[j] > A[i]  and  A[j] < A[s] then
      s ← j
   end if
end for
return s
```

La complessità della procedura $ArraySuccessor(A, i)$ è $\Theta(n)$, con $n = length(A)$.

Esercizio 4.8 *Dato un vettore A e un suo elemento x, l'elemento predecessore di x è il più grande elemento di A che è più piccolo di x. Sia A un vettore e i un suo indice. Scrivere una procedura $ArrayPredecessor(A, i)$ che ritorna l'indice dell'elemento predecessore di $A[i]$, oppure* NIL *se $A[i]$ è il minimo di A.*

Esercizio 4.9 *Si scriva una procedura $SlowSort(A)$ che ordina in senso crescente il vettore A usando le procedure $ArrayMin$ e $ArraySuccessor$.*

Soluzione

Algoritmo 7 SlowSort(A)

SlowSort(A)

```
//Uso un vettore B tale che length(B) = length(A)
j ← ArrayMin(A)
for i ← 1 to length(B) do
   B[i] ← A[j]
   j ← ArraySuccessor(A, j)
end for
for i ← 1 to length(B) do
   A[i] ← B[i]
end for
```

La complessità della procedura $SlowSort(A)$ è $\Theta(n^2)$, con $n = length(A)$. Infatti, faccio n chiamate alla procedura $ArraySuccessor$, che ha complessità $\Theta(n)$. Dunque la complessità è quadratica.

Esercizio 4.10 *Si scriva una procedura SlowSort(A) che ordina in senso decrescente il vettore A usando le procedure ArrayMax e ArrayPredecessor.*

Esercizio 4.11 *Sia A un vettore ordinato in senso crescente di lunghezza n e x un numero. Scrivere una procedura iterativa ArrayBinarySearch(A, x) che, in tempo $O(\log n)$, ritorna l'indice dell'elemento che contiene x, oppure* NIL *se tale elemento non esiste.*

Soluzione

Algoritmo 8 ArrayBinarySearch(A,x)

ArrayBinarySearch(A,x)

```
l ← 1
r ← length(A)
repeat
    i ← (l + r) div 2
    if x < A[i] then
        r ← i − 1
    else
        l ← i + 1
    end if
until x = A[i] or l > r
if A[i] = x then
    return i
else
    return NIL
end if
```

Calcoliamo la complessità della procedura *ArrayBinarySearch*(A, x). Sia $n = length(A)$. Inizialmente abbiamo una sequenza di lunghezza n su cui cercare l'elemento x. Ogni iterazione del ciclo repeat divide a metà tale sequenza, e continua la ricerca su una delle due metà, tralasciando l'altra metà. Dopo i iterazioni, la dimensione della sequenza corrente risulta essere circa $n/2^i$. Nel caso peggiore, l'elemento cercato non è presente nel vettore. In tal caso, il ciclo repeat termina quando la lunghezza della sequenza corrente è inferiore a 1. Quindi il calcolo della complessità si riduce alla seguente questione: quante volte occorre suddividere a metà una sequenza di lunghezza n prima di ottenere una sequenza di lunghezza inferiore a 1? Ciò equivale a chiederci quante volte occorre moltiplicare 2 per se stesso prima di ottenere un numero maggiore o uguale a n. La risposta è $\lceil \log n \rceil$, in quanto $2^{\lceil \log n \rceil} \geq 2^{\log n} = n$. Dunque la complessità di *ArrayBinarySearch*(A, x) è $\Theta(\log n)$.

Esercizio 4.12 *Sia A un vettore. Si scriva una procedura* $ArrayReverse(A)$ *che inverte la sequenza contenuta in A.*

Soluzione

Algoritmo 9 ArrayReverse(A)

ArrayReverse(A)

```
for i ← 1 to length(A) div 2 do
    Exchange(A, i, length(A) − i + 1)
end for
```

La complessità della procedura $ArrayReverse(A)$ è $\Theta(n/2) = \Theta(n)$, con $n = length(A)$.

Esercizio 4.13 *Sia A un vettore. Si scriva una procedura* $ArrayPalindrome(A)$ *che verifica se la sequenza contenuta in A è palindrome.*

Soluzione

Algoritmo 10 ArrayPalindrome(A)

ArrayPalindrome(A)

```
for i ← 1 to length(A) div 2 do
    if A[i] ≠ A[length(A) − i + 1] then
        return FALSE
    end if
end for
return TRUE
```

La complessità della procedura $ArrayPalindrome(A)$ è $\Theta(n/2) = \Theta(n)$, con $n = length(A)$.

Esercizio 4.14 *Siano A, B e C vettori. Si scriva una procedura $ArrayConcat(A, B, C)$ che mette in C la concatenazione delle sequenze contenute in A e in B.*

Soluzione

Algoritmo 11 ArrayConcat(A,B,C)

ArrayConcat(A,B,C)

```
// length(C) = length(A) + length(B)
if length(A) ≥ length(B) then
   n ← length(A)
else
   n ← length(B)
end if
for i ← 1 to n do
   if i ≤ length(A) then
      C[i] ← A[i]
   end if
   if i ≤ length(B) then
      C[length(A) + i] ← B[i]
   end if
end for
```

La complessità della procedura $ArrayConcat(A, B, C)$ è $\Theta(max\{n, m\})$, con $n = length(A)$ e $m = length(B)$.

Esercizio 4.15 *Siano A e B vettori. Si scriva una procedura $ArrayCompare(A, B)$ che verifica se le sequenze contenute in A e in B sono uguali.*

Soluzione

Algoritmo 12 ArrayCompare(A,B)

ArrayCompare(A,B)

```
if length(A) ≠ length(B) then
   return FALSE
end if
for i ← 1 to length(A) do
   if A[i] ≠ B[i] then
      return FALSE
   end if
end for
return TRUE
```

Nel caso pessimo, i vettori A e B hanno la medesima lunghezza n. La complessità della procedura $ArrayCompare(A, B)$ è dunque $\Theta(n)$.

Vediamo alcuni esempi di procedure ricorsive. Una **procedura ricorsiva** è una procedura che richiama sè stessa. Le procedure ricorsive sono alternative rispetto alle procedure iterative. E' possibile mostrare che ogni procedura ricorsiva è equivalente ad una procedura iterativa e viceversa. In generale, le procedure ricorsive sono più facili da scrivere rispetto alle controparti iterative, ma sono anche meno efficienti.

Esercizio 4.16 *Sia* $n \geq 0$ *un numero intero. Si scriva una procedura ricorsiva* $Factorial(n)$ *che calcola* $n!$.

Soluzione

Algoritmo 13 Factorial(n)

Factorial(n)

```
if n = 0 then
    return 1
else
    return n * Factorial(n − 1)
end if
```

La complessità $C(n)$ della procedura $Factorial(n)$ è espressa dalla seguente equazione ricorsiva: $C(0) = 1$, e, per $n > 0$, $C(n) = C(n-1) + 1$. La soluzione è $C(n) = \Theta(n)$ (Esercizio 2.10, punto 2).

Esercizio 4.17 *Siano* $n \geq k \geq 0$ *due numeri interi. Si scriva una procedura ricorsiva* $Binomial(n, k)$ *che calcola il coefficiente binomiale* (n, k)*. Si sfrutti la definizione ricorsiva seguente:* $(n, n) = (n, 0) = 1$*, e, per* $n \neq k$ *e* $k > 0$, $(n, k) = (n-1, k) + (n-1, k-1)$.

Soluzione

Algoritmo 14 Binomial(n,k)

Binomial(n,k)

```
if (n = k) or (k = 0) then
    return 1
else
    return Binomial(n − 1, k) + Binomial(n − 1, k − 1)
end if
```

La complessità $C(n, k)$ della procedura $Binomial(n, k)$ è espressa dalla seguente equazione ricorsiva: $C(n, n) = C(n, 0) = 1$, e, per $n \neq k$ e $k > 0$, $C(n, k) = C(n-1, k) + C(n-1, k-1)$. La soluzione è $C(n, k) = \Omega(2^{min\{k, n-k\}})$ (Esercizio 2.11).

Esercizio 4.18 *Si scriva una versione iterativa e efficiente di* $Binomial(n, k)$*. Si sfrutti il fatto che* $(n, k) = n!/(k!(n-k)!)$*.*

Esercizio 4.19 *Sia* A *un vettore e* $i \leq j$ *due indici di* A*. Si scriva una procedura ricorsiva* $ArrayMax(A, i, j)$ *che restituisce l'indice dell'elemento massimo del sottovettore* $A[i, \ldots, j]$*, oppure* NIL *se tale sottovettore è vuoto.*

Soluzione

Algoritmo 15 ArrayMax(A,i,j)

ArrayMax(A,i,j)

```
if i > j then
  return NIL
else
  if i = j then
    return i
  else
    l ← (j − i + 1) div 2
    m1 ← ArrayMax(A, i, i + l − 1)
    m2 ← ArrayMax(A, i + l, j)
    if A[m1] > A[m2] then
      return m1
    else
      return m2
    end if
  end if
end if
```

La complessità $C(n)$ di $ArrayMax(A, i, j)$, ove $n = j - i + 1$, è espressa dalla seguente equazione ricorsiva: $C(0) = \Theta(1)$ e, per $n > 0$, $C(n) = 2C(n/2) + \Theta(1)$. La soluzione è $C(n) = \Theta(n)$ (Esercizio 2.9, punto 2).

Esercizio 4.20 *Sia A un vettore ordinato in senso crescente di lunghezza n, x un numero e i, j due indici del vettore A. Scrivere una procedura ricorsiva $ArrayBinarySearch(A, i, j, x)$ che, in tempo $O(\log n)$, ritorna l'indice dell'elemento che contiene x cercando nel sottovettore $A[i, \ldots, j]$, oppure* NIL *se tale elemento non esiste.*

Soluzione

Algoritmo 16 ArrayBinarySearch(A,i,j,x)

ArrayBinarySearch(A,i,j,x)

```
if i > j then
   return NIL
end if
if i = j then
   if A[i] = x then
      return i
   else
      return NIL
   end if
end if
l ← (j − i + 1) div 2
if x = A[i + l] then
   return i + l
else
   if x < A[i + l] then
      return ArrayBinarySearch(A, i, i + l − 1, x)
   else
      return ArrayBinarySearch(A, i + l + 1, j, x)
   end if
end if
```

La complessità $C(n)$ di $ArrayBinarySearch(A, i, j, x)$, ove $n = j - i + 1$, è espressa dalla seguente equazione ricorsiva: $C(0) = \Theta(1)$ e, per $n > 0$, $C(n) = C(n/2) + \Theta(1)$. La soluzione è $C(n) = \Theta(\log n)$ (Esercizio 2.9, punto 1).

4.2 Pile

La struttura di dati **pila** (**stack** in inglese) rappresenta un insieme dinamico nel quale le operazioni di inserimento e cancellazione seguono una politica **LIFO** (**Last In First Out**). Ciò significa che l'oggetto cancellato è quello che è stato inserito per ultimo. Assumeremo che gli oggetti rappresentati siano numeri. La struttura di dati pila possiede un attributo $top(S)$ che contiene un puntatore (cioè un indice) alla cima della pila S e le seguenti quattro operazioni:

- $StackEmpty(S)$ che controlla se la pila S è vuota;
- $StackFull(S)$ che controlla se la pila S è piena;
- $Push(S, x)$ che inserisce l'oggetto x in cima alla pila S;
- $Pop(S)$ che rimuove l'oggetto in cima alla pila S;

Useremo i vettori per implementare la struttura di dati pila. Una pila di al più n elementi viene implementata da un vettore di lunghezza n. L'attributo $top(S)$ contiene l'indice dell'elemento in cima alla pila. Gli elementi contenuti nella pila sono $S[1], S[2], \ldots, S[top(S)]$, ove $S[1]$ è l'elemento che sta in fondo alla pila S, mentre $S[top(S)]$ è l'elemento che sta in cima alla pila S. Gli elementi $S[top(S)+1], S[top(S)+2], \ldots, S[n]$ sono liberi.

Algoritmo 17 StackEmpty(S)

StackEmpty(S)

```
if top(S) = 0 then
    return TRUE
else
    return FALSE
end if
```

Algoritmo 18 StackFull(S)

StackFull(S)

```
if top(S) = length(S) then
    return TRUE
else
    return FALSE
end if
```

Algoritmo 19 Push(S,x)

Push(S,x)

```
if StackFull(S) then
    error OVERFLOW
else
    top(S) ← top(S) + 1
    S[top(S)] ← x
end if
```

Algoritmo 20 Pop(S)

Pop(S)

```
if StackEmpty(S) then
    error UNDERFLOW
else
    x ← S[top[S]]
    top(S) ← top(S) − 1
    return x
end if
```

La complessità delle quattro procedure precedenti è costante, cioè $\Theta(1)$.

Esercizio 4.21 *Sia S una pila. Si scriva una procedura $StackReverse(S)$ che inverte la sequenza contenuta in S.*

Soluzione

Algoritmo 21 StackReverse(S)

StackReverse(S)

```
// Uso un vettore A tale che length(A) = length(S)
len ← top(S)
while  not StackEmpty(S) do
    A[i] ← Pop(S)
    i ← i + 1
end while
for i ← 1 to len do
    Push(S, A[i])
end for
```

La complessità di $StackReverse(S)$ è $\Theta(n)$, ove $n = top(S)$ è la dimensione della pila S.

Esercizio 4.22 *Siano R e S pile. Si scriva una procedura $StackCompare(R, S)$ che verifica se le sequenze contenute in R e in S sono uguali.*

Soluzione

Algoritmo 22 StackCompare(R,S)

StackCompare(R,S)

```
if top(R) ≠ top(S) then
   return FALSE
end if
while not StackEmpty(R) do
   if Pop(R) ≠ Pop(S) then
      return FALSE
   end if
end while
return TRUE
```

Nel caso pessimo, le pile R e S hanno la medesima dimensione n. La complessità di $StackCompare(R, S)$ è dunque $\Theta(n)$.

Esercizio 4.23 *Sia S una pila. Si scriva una procedura $StackPalindrome(S)$ che verifica se la sequenza contenuta in S è palindrome.*

Soluzione

Algoritmo 23 StackPalindrome(S)

StackPalindrome(S)

```
// Uso una pila vuota T
l ← top(S)
for i ← 1 to l div 2 do
   Push(T, Pop(S))
end for
//se l è dispari
if (l mod 2) = 1 then
   Pop(S)
end if
return StackCompare(S, T)
```

La complessità di $StackPalindrome(S)$ è $\Theta(n)$, ove $n = top(S)$ è la dimensione della pila S.

Esercizio 4.24 *Siano R e S pile. Si scriva una procedura $StackConcat(R, S)$ che mette in S la concatenazione delle sequenze contenute in R e in S.*

Soluzione

Algoritmo 24 StackConcat(R,S)

StackConcat(R,S)

```
StackReverse(R)
while not StackEmpty(R) do
    Push(S, Pop(R))
end while
```

La complessità di $StackConcat(R, S)$ è $\Theta(n)$, ove $n = top(R)$ è la dimensione della pila R.

4.3 Code

La struttura di dati **coda** (**queue** in inglese) rappresenta un insieme dinamico nel quale le operazioni di inserimento e cancellazione seguono una politica **FIFO** (**First In First Out**). Ciò significa che dire che l'oggetto cancellato è quello che è stato inserito per primo. Assumeremo che gli oggetti rappresentati siano numeri. La struttura di dati coda possiede un attributo $head(Q)$ che contiene un puntatore all'inizio della coda, un attributo $tail(Q)$ che contiene un puntatore alla fine della coda, e le seguenti quattro operazioni:

- $QueueEmpty(Q)$ che controlla se la coda Q è vuota;
- $QueueFull(Q)$ che controlla se la coda Q è piena;
- $Enqueue(Q, x)$ che inserisce l'oggetto x alla fine della coda Q;
- $Dequeue(S)$ che rimuove l'oggetto all'inizio della coda Q;

Useremo i vettori per implementare la struttura di dati coda. Una coda di al più $n-1$ elementi viene implementata da un vettore di lunghezza n. L'attributo $head(Q)$ contiene l'indice dell'elemento all'inizio della coda, e dunque punta alla locazione del prossimo elemento da cancellare, mentre l'attributo $tail(Q)$ contiene l'indice dell'elemento alla fine della coda incrementato di uno, dunque punta alla locazione in cui verrà inserito il prossimo elemento. La coda è circolare, cioè la locazione 1 segue immediatamente la locazione n. Dunque, se $head(Q) = k < n$, allora $head(Q) + 1 = k + 1$, mentre se $head(Q) = n$, allora $head(Q)+1 = 1$. Inoltre, se $head(Q) = k > 1$, allora $head(Q)-1 = k-1$, mentre se $head(Q) = 1$, allora $head(Q) - 1 = n$. Similmente per $tail(Q)$. Gli elementi contenuti nella coda sono $Q[head(Q)], Q[head(Q) + 1], \ldots, Q[tail(Q) - 1]$. Gli elementi $Q[tail(Q)], Q[tail(Q) + 1], \ldots, Q[head(Q) - 1]$ sono liberi. La coda Q è vuota se $head(Q) = tail(Q)$, mentre Q è piena se $head(Q) = tail(Q) + 1$.

Algoritmo 25 QueueEmpty(Q)

QueueEmpty(Q)

```
if head(Q) = tail(Q) then
    return TRUE
else
    return FALSE
end if
```

Algoritmo 26 QueueFull(Q)

QueueFull(Q)

```
if ((tail(Q) + 1) mod length(Q)) = head(Q) then
    return TRUE
else
    return FALSE
end if
```

Algoritmo 27 Enqueue(Q,x)

Enqueue(Q,x)

```
if QueueFull(Q) then
    error OVERFLOW
else
    Q[tail(Q)] ← x
    tail(Q) ← (tail(Q) + 1) mod length(Q)
end if
```

Algoritmo 28 Dequeue(Q)

Dequeue(Q)

```
if QueueEmpty(S) then
    error UNDERFLOW
else
    x ← Q[head(Q)]
    head(Q) ← (head(Q) + 1) mod length(Q)
    return x
end if
```

La complessità delle quattro procedure precedenti è costante, cioè $\Theta(1)$.

Esercizio 4.25 *Sia Q una coda. Si scriva una procedura $QueueLength(Q)$ che ritorna la lunghezza della coda Q.*

Soluzione

Algoritmo 29 QueueLength(Q)

QueueLength(Q)

if $head(Q) \leq tail(Q)$ **then**
    return $tail(Q) - head(Q)$
else
    return $length(Q) - head(Q) + tail(Q)$
end if

La complessità di $QueueLength(Q)$ è costante, cioè $\Theta(1)$.

Esercizio 4.26 *Siano P e Q code. Si scriva una procedura $QueueConcat(P, Q)$ che mette in P la concatenazione delle sequenze contenute in P e in Q.*

Soluzione

Algoritmo 30 QueueConcat(P,Q)

QueueConcat(P,Q)

while not $QueueEmpty(Q)$ **do**
    $Enqueue(P, Dequeue(Q))$
end while

La complessità di $QueueConcat(P, Q)$ è $\Theta(n)$, ove $n = QueueLength(Q)$ è la dimensione della coda Q.

Esercizio 4.27 *Siano P e Q pile. Si scriva una procedura $QueueCompare(P, Q)$ che verifica se le sequenze contenute in P e in Q sono uguali.*

Soluzione

Algoritmo 31 QueueCompare(P,Q)

QueueCompare(P,Q)

```
if QueueLength(P) ≠ QueueLength(Q) then
  return FALSE
end if
while not QueueEmpty(P) do
  if Dequeue(P) ≠ Dequeue(Q) then
    return FALSE
  end if
end while
return TRUE
```

Nel caso peggiore, le code P e Q hanno la stessa dimensione n. La complessità di $QueueCompare(P, Q)$ è dunque $\Theta(n)$.

Esercizio 4.28 *Sia Q una coda. Si scriva una procedura $QueueReverse(Q)$ che inverte la sequenza contenuta in Q.*

Soluzione

Algoritmo 32 QueueReverse(Q)

QueueReverse(Q)

```
// Uso una pila vuota S
while not QueueEmpty(Q) do
  Push(S, Dequeue(Q))
end while
while not StackEmpty(S) do
  Enqueue(Q, Pop(S))
end while
```

La complessità di $QueueReverse(Q)$ è $\Theta(n)$, ove $n = QueueLength(Q)$ è la dimensione della coda Q.

Esercizio 4.29 *Siano P una coda e Q una coda vuota. Si descriva l'effetto delle seguenti due procedure.*

Algoritmo 33 QueueReverse(P,Q)

QueueReverse(P,Q)

while **not** $QueueEmpty(P)$ **do**
  $Enqueue(Q, Dequeue(P))$
end while

Algoritmo 34 QueueReverse(P)

QueueReverse(P)

while **not** $QueueEmpty(P)$ **do**
  $Enqueue(P, Dequeue(P))$
end while

Soluzione

L'effetto della prima procedura è di copiare in Q la coda P. L'effetto della seconda è un ciclo infinito, qualora P non sia vuota.

Esercizio 4.30 *Sia Q una coda. Si scriva una procedura $QueuePalindrome(Q)$ che verifica se la sequenza contenuta in Q è palindrome.*

Soluzione

Algoritmo 35 QueuePalindrome(Q)

QueuePalindrome(Q)

// Uso una pila S vuota
$l \leftarrow QueueLength(Q)$
for $i \leftarrow 1$ **to** l div 2 **do**
  $Push(S, Dequeue(Q))$
end for
//se l è dispari
if $(l \ mod \ 2) = 1$ **then**
  $x \leftarrow Dequeue(Q)$
end if
while **not** $QueueEmpty(Q)$ **do**
  if $Dequeue(Q) \neq Pop(S)$ **then**
    return FALSE
  end if
end while
return TRUE

La complessità di $QueuePalindrome(Q)$ è $\Theta(n)$, ove $n = QueueLength(Q)$ è la dimensione della coda Q.

Esercizio 4.31 *Implementare la struttura di dati pila usando due code e la struttura di dati coda usando due pile.*

Soluzione

Data una pila S di dimensione n, uso due code P e Q di dimensione $n+1$ tali che una delle due code è sempre vuota, e l'altra contiene gli elementi della pila S, con la cima della pila che corrisponde alla fine della coda.

Algoritmo 36 StackEmpty(P,Q)

StackEmpty(P,Q)

```
if QueueEmpty(P) and QueueEmpty(Q) then
    return TRUE
else
    return FALSE
end if
```

Algoritmo 37 StackFull(P,Q)

StackFull(P,Q)

```
if QueueFull(P) or QueueFull(Q) then
    return TRUE
else
    return FALSE
end if
```

Algoritmo 38 Push(P,Q,x)

Push(P,Q,x)

```
if not QueueEmpty(P) then
    Enqueue(P, x)
else
    Enqueue(Q, x)
end if
```

Algoritmo 39 Pop(P,Q)

Pop(P,Q)

if **not** $QueueEmpty(P)$ **then**
  $len \leftarrow QueueLength(P)$
  for $i \leftarrow 1$ **to** $len - 1$ **do**
    $Enqueue(Q, Dequeue(P))$
  end for
  return $Dequeue(P)$
else
  $len \leftarrow QueueLength(Q)$
  for $i \leftarrow 1$ **to** $len - 1$ **do**
    $Enqueue(P, Dequeue(Q))$
  end for
  return $Dequeue(Q)$
end if

La complessità di $StackEmpty$, $StackFull$, e $Push$ è costante, mentre la complessità di Pop è $\Theta(n)$, ove n è la dimensione della pila.

Data una coda Q di dimensione $n+1$, uso due pile R e S di dimensione n tali che una delle due pile è sempre vuota, e l'altra contiene gli elementi della coda Q, con la cima della pila che corrisponde alla fine della coda.

Algoritmo 40 QueueEmpty(R,S)

QueueEmpty(R,S)

if $StackEmpty(R)$ **and** $StackEmpty(S)$ **then**
  return TRUE
else
  return FALSE
end if

Algoritmo 41 QueueFull(R,S)

QueueFull(R,S)

if $StackFull(R)$ **or** $StackFull(S)$ **then**
  return TRUE
else
  return FALSE
end if

Algoritmo 42 Enqueue(R,S,x)

Enqueue(R,S,x)

```
if not StackEmpty(R) then
    Push(R, x)
else
    Push(S, x)
end if
```

Algoritmo 43 Dequeue(R,S)

Dequeue(R,S)

```
if not StackEmpty(R) then
    StackReverse(R)
    x ← Pop(R)
    while not StackEmpty(R) do
        Push(S, Pop(R))
    end while
    return x
else
    StackReverse(S)
    x ← Pop(S)
    while not StackEmpty(S) do
        Push(R, Pop(S))
    end while
    return x
end if
```

La complessità di *QueueEmpty*, *QueueFull*, e *Enqueue* è costante, mentre la complessità di *Dequeue* è $\Theta(n)$, ove n è la dimensione della coda.

4.4 Liste

La struttura di dati **lista** (**list** in inglese) rappresenta un insieme dinamico che, a differenza della pila e della coda, non implementa particolari politiche di inserimento e cancellazione e inoltre non ha una capienza massima. Una lista possiede due differenze rispetto ad un vettore: posso inserire e cancellare oggetti, quindi modificare le dimensioni della lista. Inoltre, l'accesso agli oggetti è sequenziale, e non diretto. Ciò significa che se voglio accedere l'oggetto i-esimo della lista, devo prima passare per tutti gli oggetti che lo precedono, da 1 a $i-1$.

Gli oggetti rappresentati in una lista non sono semplicemente numeri, ma elementi strutturati. Inoltre, l'accesso agli oggetti in una lista avviene attraverso puntatori. Diamo una definizione generale di oggetto e di puntatore. Un **oggetto** è un'area di memoria suddivisa in sezioni chiamate **campi**. Per accedere ad un oggetto, devo conoscere l'indirizzo di memoria che contiene l'oggetto. Un **puntatore** è una variabile che contiene un indirizzo di memoria. Un puntatore inoltre può contenere la costante NIL. Se x è un puntatore ad un oggetto, e C è il nome di un campo, allora $C[x]$ è il contenuto del campo dell'oggetto puntato da x. $C[x]$ può essere usato come una qualsiasi variabile. Si noti che dopo l'assegnamento $y \leftarrow x$, y punta allo stesso oggetto puntato da x, e dunque $C[x] = C[y]$ per ogni campo C dell'oggetto puntato da x. La variabile y è detta **alias** di x. Inoltre, se x è l'unico puntatore ad un oggetto, cioè se non esistono alias di x, allora dopo l'istruzione $x \leftarrow$ NIL non abbiamo più accesso all'oggetto, e l'area di memoria che occupava rimane inutilizzata.

Useremo la stessa notazione per puntatore e oggetto puntato dal puntatore (a differenza del linguaggio di programmazione C). Sia x una variabile che contiene un indirizzo di memoria. La variabile x rappresenta un puntatore o un oggetto a seconda dell'uso che ne facciamo. Più precisamente, nell'assegnamento $y \leftarrow x$, x rappresenta un puntatore, e l'effetto dell'istruzione è quello di assegnare a y l'indirizzo di memoria contenuto in x. Viceversa, nell'assegnamento $z \leftarrow C[x]$, x rappresenta un oggetto (l'oggetto allocato all'indirizzo di memoria contenuto in x), e l'effetto dell'istruzione è quello di assegnare alla variabile z il contenuto del campo C dell'oggetto puntato da x.

Infine, una lista gestisce la memoria in modo **dinamico**. Quando mi serve un nuovo oggetto, faccio una **allocazione** della memoria necessaria per contenere l'oggetto, cioè chiedo al sistema operativo di riservarmi una porzione di memoria in modo che altri utenti non possano accedervi. Quando un oggetto non mi serve ulteriormente, faccio una **deallocazione** della memoria che conteneva l'oggetto, cioè libero la memoria allocata e la restituisco al sistema operativo in modo che altri utenti la possano utilizzare. La gestione dinamica della memoria si contrappone alla gestione **statica**, nella quale la memoria viene allocata all'inizio del programma e viene deallocata alla fine del programma. Non abbiamo quindi allocazioni o deallocazioni run-time, cioè *durante* l'esecuzione del programma. Una gestione dinamica della memoria è preferibile in quanto permette in ogni momento di allocare esattamente la memoria che serve. Nel nostro pseudocodice ci astraiamo dalle operazioni di allocazione e deallocazione della memoria.

Useremo un modello di lista chiamato **lista doppia** o **lista bidirezionale**.

Gli oggetti contenuti in una lista doppia hanno tre campi: una campo chiave *key*, un campo successore *next* e un campo predecessore *prev*. La lista è doppia perchè ogni oggetto è collegato in avanti e indietro al resto della lista. Vedremi in seguito altri modelli di lista. Possono inoltre esistere altri campi, chiamati *dati satellite*. Assumeremo che $key[x]$ contenga un numero. $next[x]$ è un puntatore all'oggetto che segue x, se tale oggetto esiste, oppure la costante NIL se x è l'ultimo oggetto della lista. Infine, $prev[x]$ è un puntatore all'oggetto che precede x, se tale oggetto esiste, oppure la costante NIL se x è il primo oggetto della lista (se x è l'unico oggetto della lista, entrambi $next[x]$ e $prev[x]$ sono NIL).

La struttura di dati lista possiede un attributo $head(L)$ che contiene un puntatore al primo oggetto della lista L, e le seguenti quattro operazioni:

- $ListEmpty(L)$ che controlla se la lista L è vuota;
- $ListSearch(L, k)$ che ritorna un puntatore all'oggetto con chiave k, se esiste, e NIL altrimenti;
- $ListInsert(L, x)$ che inserisce l'oggetto puntato da x in testa alla lista;
- $ListDelete(L, x)$ che cancella l'oggetto puntato da x.

Algoritmo 44 $ListEmpty(L)$

$ListEmpty(L)$

if $head(L) =$ NIL **then**
  return TRUE
else
  return FALSE
end if

Algoritmo 45 ListSearch(L, k)

ListSearch(L, k)

$x \leftarrow head(L)$
while $x \neq$ NIL **and** $key[x] \neq k$ **do**
  $x \leftarrow next[x]$
end while
return x

Algoritmo 46 ListInsert(L,x)

ListInsert(L,x)

$next[x] \leftarrow head(L)$
$prev[x] \leftarrow$ NIL
if $head(L) \neq$ NIL **then**
  $prev[head(L)] \leftarrow x$
end if
$head(L) \leftarrow x$

Algoritmo 47 ListDelete(L,x)

ListDelete(L,x)

if $prev[x] \neq$ NIL **then**
  $next[prev[x]] \leftarrow next[x]$
else
  $head(L) \leftarrow next[x]$
end if
if $next[x] \neq$ NIL **then**
  $prev[next[x]] \leftarrow prev[x]$
end if

Le procedure *ListEmpty*, *ListInsert*, e *ListDelete* hanno complessità costante, mentre $ListSearch(L, k)$ ha complessità $\Theta(n)$, con n la lunghezza della lista L.

Esercizio 4.32 *Sia L una lista. Si scriva una procedura* $ListLength(L)$ *che ritorna la lunghezza della lista L.*

Soluzione

Algoritmo 48 ListLength(L)

ListLength(L)

$x \leftarrow head(L)$
$len \leftarrow 0$
while $x \neq$ NIL **do**
  $x \leftarrow next[x]$
  $len \leftarrow len + 1$
end while
return len

La complessità di $ListLength(L)$ è $\Theta(n)$, con n è la lunghezza della lista L.

Esercizio 4.33 *Sia L una lista. Si scriva una procedura $ListReverse(L)$ che inverte la sequenza contenuta in L.*

Soluzione

Algoritmo 49 ListReverse(L)

ListReverse(L)

$y \leftarrow$ NIL
$x \leftarrow head(L)$
while $x \neq$ NIL **do**
    $t \leftarrow next[x]$
    $next[x] \leftarrow prev[x]$
    $prev[x] \leftarrow t$
    $y \leftarrow x$
    $x \leftarrow t$
end while
$head(L) \leftarrow y$

La complessità di $ListReverse(L)$ è $\Theta(n)$, con n è la lunghezza della lista L.

Esercizio 4.34 *Sia L una lista. Si scriva una procedura $ListPalindrome(L)$ che verifica se la sequenza contenuta in L è palindrome.*

Soluzione

Algoritmo 50 ListPalindrome(L)

ListPalindrome(L)

```
y ← NIL
x ← head(L)
while x ≠ NIL do
    y ← x
    x ← next[x]
end while
x ← head(L)
len ← ListLength(L)
for i ← 1 to len div 2 do
    if key[x] ≠ key[y] then
        return FALSE
    else
        x ← next[x]
        y ← prev[y]
    end if
end for
return TRUE
```

La complessità di $ListPalindrome(L)$ è $\Theta(n)$, con n è la lunghezza della lista L.

Esercizio 4.35 *Siano L e M due liste. Si scriva una procedura $ListConcat(L, M)$ che concatena le liste L e M e ritorna un puntatore alla lista concatenata.*

Soluzione

Algoritmo 51 ListConcat(L,M)

ListConcat(L,M)

```
if ListEmpty(L) then
    return head(M)
end if
if ListEmpty(M) then
    return head(L)
end if
x ← head(L)
while x ≠ NIL do
    y ← x
    x ← next[x]
end while
next[y] ← head(M)
prev[head(M)] ← y
return head(L)
```

La complessità di $ListConcat(L, M)$ è $\Theta(n)$, con n è la lunghezza della lista L.

Esercizio 4.36 *Siano L e M due liste. Si scriva una procedura $ListCompare(L, M)$ che confronta il contenuto di L e M.*

Soluzione

Algoritmo 52 ListCompare(L,M)

ListCompare(L,M)

```
x ← head(L)
y ← head(M)
while (x ≠ NIL) and (y ≠ NIL) do
    if key[x] ≠ key[y] then
        return FALSE
    else
        x ← next[x]
        y ← next[y]
    end if
end while
if (x = NIL) and (y = NIL) then
    return TRUE
else
    return FALSE
end if
```

La complessità di $ListCompare(L, M)$ è $\Theta(min\{n, m\})$, con n è la lunghezza della lista L e m è la lunghezza della lista M.

Esercizio 4.37 *Una* ***lista circolare*** *è una lista in cui il successore dell'ultimo oggetto della lista è il primo elemento della lista e il predecessore del primo oggetto della lista è l'ultimo oggetto della lista. Siano L e M due liste circolari. Si scriva una procedura* $CircularListConcat(L, M)$ *che concatena le liste L e M e ritorna un puntatore alla lista circolare concatenata.*

Soluzione

Algoritmo 53 CircularListConcat(L,M)

CircularListConcat(L,M)

if $ListEmpty(L)$ **then**
  return $head(M)$
end if
if $ListEmpty(M)$ **then**
  return $head(L)$
end if
$y \leftarrow prev[head(L)]$
$z \leftarrow prev[head(M)]$
$next[y] \leftarrow head(M)$
$prev[head(M)] \leftarrow y$
$prev[head(L)] \leftarrow z$
$next[z] \leftarrow head(L)$
return $head(L)$

La complessità di $CircularListConcat(L, M)$ è $\Theta(1)$.

Esercizio 4.38 *Una lista* ***singola*** *o* ***lista monodirezionale*** *è una lista in cui gli oggetti hanno due campi: il campo chiave key e il campo next che punta all'oggetto successivo (manca quindi il campo prev). Sia L una lista singola di lunghezza n. Si scriva una procedura $SingleListReverse(L)$ che inverte in tempo $O(n)$ la sequenza contenuta in L.*

Soluzione

Algoritmo 54 SingleListReverse(L)

SingleListReverse(L)

$y \leftarrow$ NIL
$x \leftarrow head(L)$
while $x \neq$ NIL **do**
    $z \leftarrow next[x]$
    $next[x] \leftarrow y$
    $y \leftarrow x$
    $x \leftarrow z$
end while
$head(L) \leftarrow y$

Esercizio 4.39 *Implementare le operazioni di ricerca, inserimento e cancellazione su liste circolari e su liste singole.*

Esercizio 4.40 *Implementare le procedure di inversione, palindrome, concatenazione e confronto su liste circolari e su liste singole.*

Esercizio 4.41 *Implementare le strutture di dati pila e coda usando le liste.*

Soluzione
Implemento una pila S usando una lista L mettendo l'elemento in cima alla pila in posizione di testa nella lista.

Algoritmo 55 StackEmpty(L)

StackEmpty(L)

1: **return** $ListEmpty(L)$

Algoritmo 56 StackFull(L)

StackFull(L)

1: **return** FALSE

Algoritmo 57 Push(L,x)

Push(L,x)

1: $ListInsert(L, x)$

Algoritmo 58 Pop(L)

Pop(L)

if $ListEmpty(L)$ **then**
  error UNDERFLOW
end if
$h \leftarrow head(L)$
$ListDelete(L, h)$
return h

Implemento una coda Q usando una lista circolare L mettendo l'elemento all'inizio della coda in posizione di testa nella lista.

Algoritmo 59 QueueEmpty(L)

QueueEmpty(L)

return $CircularListEmpty(L)$

Algoritmo 60 QueueFull(L)

QueueFull(L)

return FALSE

Algoritmo 61 Enqueue(L,x)

Enqueue(L,x)

$M \leftarrow \emptyset$
$CircularListInsert(M, x)$
$CircularListConcat(L, M)$

Algoritmo 62 Dequeue(L)

Dequeue(L)

if $CircularListEmpty(L)$ **then**
  error UNDERFLOW
end if
$h \leftarrow head(L)$
$CircularListDelete(L, h)$
return h

Tutte le operazioni hanno complessità costante. Si noti che, a differenza dell'implementazione su vettori, l'allocazione e la deallocazione della memoria avvengono dinamicamente. Questo è un vantaggio, in quanto uso solo la memoria che mi serve. Inoltre, non ho limiti massimi di capacità (a parte il limite della memoria fisica a disposizione sulla macchina).

4.5 Alberi binari di ricerca

La struttura di dati albero binario di ricerca permette di implementare in modo efficiente tutte le operazioni su un insieme dinamico. Un **albero binario** è una struttura matematica definita ricorsivamente su un insieme finito di **nodi** nel modo seguente:

- l'albero vuoto () è una albero binario;
- se x è un nodo, T_1 e T_2 sono alberi binari disgiunti che non contengono x, allora (x, T_1, T_2) è un albero binario.

Dato un albero $T = (x, T_1, T_2)$, il nodo x è la radice, T_1 è il **sottoalbero sinistro** di T, e T_2 è il **sottoalbero destro** di T. La radice di T_1, se esiste, è il **figlio sinistro** di x, mentre la radice di T_2, se esiste, è il **figlio destro** di x. Il nodo x viene detto **padre** dei propri figli. Un **nodo foglia** è un nodo privo di figli. Un **nodo interno** è un nodo con almeno un figlio. Un **cammino** di lunghezza n è una sequenza di nodi $\langle x_1, x_2, \ldots x_{n+1} \rangle$ tale che x_{i+1} è figlio di x_i per ogni $1 \leq i \leq n$. Un **antenato** di x è un qualsiasi nodo sull'unico cammino da x alla radice dell'albero. Diremo che y è **discendente** di x se x è un antenato di y. Si noti che un nodo è sia antenato che discendente di se stesso. La **profondità di un nodo** x è la lunghezza dell'unico cammino che parte dalla radice dell'albero e termina in x. L'**altezza di un nodo** x è la lunghezza del cammino più lungo che parte da x e termina in una foglia dell'albero. L'**altezza di un albero** è l'altezza della sua radice o, equivalentemente, è la massima profondità di un nodo dell'albero. Dato $i \geq 0$, il **livello** i-esimo di un albero è l'insieme dei nodi che hanno profondità i. Un albero è **pieno** se ogni nodo interno ha esattamente due figli. Un albero è **completo** se è pieno e ogni sua foglia ha la stessa profondità. Un albero è **quasi completo** se è completo fino al penultimo livello e i nodi dell'ultimo livello sono inseriti da sinistra a destra. Dato un albero T e un nodo x di T, useremo la notazione $T(x)$ per indicare il sottoalbero di T radicato in x. Una albero si dice **lineare** se ogni nodo interno ha esattamente un figlio.

Esercizio 4.42 *Qual è il numero di nodi di un albero completo di altezza h? Qual è l'altezza di un albero completo di n nodi? Qual è il minimo e il massimo numero di nodi di un albero quasi completo di altezza h? E di un albero pieno di altezza h? E di un albero di altezza h?*

Soluzione

Sia T un albero completo di altezza h e sia n il numero di nodi di T. Sommando i nodi per livelli, abbiamo che $n = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$. Dunque $n = \Theta(2^h)$. Passando al logaritmo, abbiamo che $h = \log(n+1) - 1 = \Theta(\log n)$.

Il minimo numero di nodi di un albero quasi completo di altezza h è 2^h (l'albero ha un solo nodo all'ultimo livello), il massimo è $2^{h+1} - 1$ (l'albero è completo). Cioè $2^h \leq n \leq 2^{h+1} - 1$ e dunque $n = \Theta(2^h)$ e $h = \Theta(\log n)$.

Il minimo numero di nodi di un albero pieno di altezza h è $2h + 1$ (ogni nodo interno ha almeno un figlio foglia), il massimo è $2^{h+1} - 1$ (l'albero è completo).

Cioè $2h+1 \leq n \leq 2^{h+1}-1$. Dunque $n = \Omega(h)$ e $n = O(2^h)$, e quindi $h = \Omega(\log n)$ e $h = O(n)$.

Il minimo numero di nodi di un albero di altezza h è $h+1$ (l'albero è lineare, cioè ogni nodo interno ha un unico figlio), il massimo è $2^{h+1} - 1$ (l'albero è completo). Cioè $h+1 \leq n \leq 2^{h+1} - 1$. Dunque $n = \Omega(h)$ e $n = O(2^h)$, e quindi $h = \Omega(\log n)$ e $h = O(n)$.

La struttura di dati **albero binario** (**binary tree** in inglese) rappresenta un insieme dinamico in forma di albero binario. Ogni elemento dell'insieme corrisponde ad un nodo dell'albero. Ogni nodo è rappresentato da un oggetto dotato di quattro campi: una chiave *key*, un puntatore al padre *p*, un puntatore al figlio sinistro $left$ e un puntatore al figlio destro $right$. La chiave identifica univocamente l'oggetto: per ogni $x \neq y$, $key[x] \neq key[y]$. Se il nodo x è la radice, allora $p[x] = \text{NIL}$, se x non ha un figlio sinistro, allora $left[x] = \text{NIL}$, se x non ha un figlio destro, allora $right[x] = \text{NIL}$. La struttura di dati albero binario possiede un attributo $root(T)$ che contiene un puntatore alla radice dell'albero T. La struttura di dati **albero binario di ricerca** (**binary search tree** in inglese) rappresenta un insieme dinamico in forma di albero binario le cui chiavi soddisfano la seguente **proprietà degli alberi binari di ricerca**:

> Se y è un nodo del sottoalbero sinistro di x, allora $key[y] \leq key[x]$.
> Se y è un nodo del sottoalbero destro di x, allora $key[x] \leq key[y]$.

La struttura di dati albero binario di ricerca possiede le seguenti otto operazioni:

- $TreeEmpty(T)$ che controlla se l'albero T è vuoto;
- $TreeSearch(x, k)$ che ritorna, se esiste, un puntatore all'oggetto con chiave k cercando nell'albero radicato in x, oppure NIL altrimenti;
- $TreeMin(x)$ che ritorna un puntatore all'oggetto con chiave minima dell'albero radicato in x, oppure NIL se l'albero radicato in x è vuoto;
- $TreeMax(x)$ che ritorna un puntatore all'oggetto con chiave massima dell'albero radicato in x, oppure NIL se l'albero radicato in x è vuoto;
- $TreeSuccessor(x)$ che ritorna un puntatore all'oggetto successore di x, oppure NIL se x è il massimo. Il successore di x è l'oggetto che ha la più piccola chiave tra i maggioranti, cioè tra quelli che hanno chiave maggiore o uguale della chiave di x;
- $TreePredecessor(x)$ che ritorna un puntatore all'oggetto predecessore di x, oppure NIL se x è il minimo. Il predecessore di x è l'oggetto che ha la più grande chiave tra i minoranti, ciè tra quelli che hanno chiave minore o uguale della chiave di x;
- $TreeInsert(T, x)$ che inserisce l'oggetto puntato da x nell'albero T mantenendo la proprietà degli alberi binari di ricerca;

- $TreeDelete(T, x)$ che cancella l'oggetto puntato da x dall'albero T mantenendo la proprietà degli alberi binari di ricerca.

Algoritmo 63 TreeEmpty(T)

TreeEmpty(T)

```
if root(T) = NIL then
    return TRUE
else
    return FALSE
end if
```

Algoritmo 64 TreeSearch(x,k)

TreeSearch(x,k)

```
if x = NIL then
    return NIL
end if
if k = key[x] then
    return x
end if
if k < key[x] then
    return TreeSearch(left[x], k)
else
    return TreeSearch(right[x], k)
end if
```

La procedura $TreeSearch(x, k)$ traccia un cammino che parte da x e termina sull'elemento con chiave k, se tale elemento esiste. Nel caso peggiore la profondità dell'elemento con chiave k è la massima profondità dell'albero radicato in x, cioè è l'altezza dell'albero radicato in x. Quindi la complessità della procedura di ricerca è $\Theta(h)$, ove h è l'altezza dell'albero.

Esercizio 4.43 *Sia scriva una versione iterativa di $TreeSearch$.*

Soluzione

La complessità della procedura $TreeSearch(x, k)$ è pari all'altezza h dell'albero radicato in x, dunque $\Theta(h)$.

Algoritmo 65 TreeSearch(x,k)

TreeSearch(x,k)

```
while (x ≠ NIL) and (key[x] ≠ k) do
    if k < key[x] then
        x ← left[x]
    else
        x ← right[x]
    end if
end while
return x
```

Proposizione 4.1 *Il minimo di un albero binario di ricerca è l'ultimo nodo del cammino più a sinistra che parte dalla radice. Il massimo di un albero binario di ricerca è l'ultimo nodo del cammino più a destra che parte dalla radice.*

Dimostrazione

Ragioniamo come segue. Siano r la radice, e x e y i figli sinistro e destro di r, qualora esistano. Per la proprietà degli alberi binari di ricerca, $key[x] \leq key[r] \leq key[z]$ per ogni $z \in T(y)$. Dunque il figlio sinistro x della radice è il minimo corrente. Siamo u e v i figli sinistro e destro di x, qualora esistano. Per la proprietà degli alberi binari di ricerca, $key[u] \leq key[x] \leq key[z]$ per ogni $z \in T(v)$, e dunque il figlio sinistro u di x è il minimo corrente. E così via fino a quando raggiungiamo l'ultimo nodo del cammino più a sinistra che parte dalla radice. Tale nodo sarà necessariamente il minimo. La dimostrazione per il massimo è simmetrica.

Algoritmo 66 TreeMin(x)

TreeMin(x)

```
if x = NIL then
    return NIL
end if
while left[x] ≠ NIL do
    x ← left[x]
end while
return x
```

Le procedure $TreeMin(x)$ e $TreeMax(x)$ hanno complessità pessima $\Theta(h)$, ove h è l'altezza dell'albero radicato in x.

Algoritmo 67 TreeMax(x)

TreeMax(x)

```
if x = NIL then
    return NIL
end if
while right[x] ≠ NIL do
    x ← right[x]
end while
return x
```

Esercizio 4.44 *Sia scriva una versione ricorsiva di $TreeMax$ e di $TreeMin$.*

Soluzione

Algoritmo 68 TreeMax(x)

TreeMax(x)

```
if x = NIL then
    return NIL
end if
if right[x] = NIL then
    return x
else
    TreeMax(right[x])
end if
```

Algoritmo 69 TreeMin(x)

TreeMin(x)

```
if x = NIL then
    return NIL
end if
if left[x] = NIL then
    return x
else
    TreeMin(left[x])
end if
```

Proposizione 4.2 *Il successore di un nodo x, se esiste, è il minimo del sottoalbero di destra del nodo x, se tale albero non è vuoto. Altrimenti è il più basso antenato di x il cui figlio sinistro è un antenato di x. Il predecessore di un nodo x, se esiste, è il massimo del sottoalbero di sinistra del nodo x, se tale albero non è vuoto. Altrimenti è il più basso antenato di x il cui figlio destro è un antenato di x.*

Dimostrazione

Dimostriamo la tesi per il successore. Il ragionamento per il predecessore è simmetrico. Useremo il seguente lemma: se x e y sono due nodi tali che uno è successore dell'altro, allora x e y si trovano sullo stesso cammino, cioè x è antenato di y o y è antenato di x. Infatti, supponiamo per assurdo che x e y non si trovino sullo stesso cammino. Allora esiste z tale che x appartiene al sottalbero sinistro di z e y appartiene al sottoalbero destro di z, o viceversa. Nel primo caso $key[x] \leq key[z] \leq key[y]$, e nel secondo $key[y] \leq key[z] \leq key[x]$, dunque ne y è successore di x e neppure x è successore di x. Questo contraddice l'ipotesi, dunque x e y appartengono allo stesso cammino.

Supponiamo che x abbia un sottoalbero destro non vuoto T_2. Sia T_1 il sottoalbero sinistro (eventualmente vuoto) di x. Per il lemma di sopra, il successore deve appartenere al cammino che va da x alla radice oppure a T_1 oppure a T_2. I nodi di T_1 sono minoranti di x e dunque possono essere esclusi. I nodi di T_2 sono maggioranti di x e quindi potenziali successori. Tutti i nodi sul cammino da x alla radice sono minoranti di x (se li incontro svoltano a sinistra sul cammino da x alla radice) o maggioranti di x *e* di tutti i nodi in T_2 (se li incontro svoltano a destra sul cammino da x alla radice). In entrambi i casi non possono diventare il successore di x, in quanto esso è il *minimo* dei maggioranti di x. Dunque il successore va cercato sull'albero T_2 e in particolare sarà il minimo di tale albero. Supponiamo ora che x non abbia un sottoalbero destro. Con un ragionamento simile al precedente il successore appartiene al cammino che va da x alla radice e in particolare corrisponderà al nodo più basso che incontro svoltando a destra su tale cammino.

Algoritmo 70 TreeSuccessor(x)

TreeSuccessor(x)

if $right[x] \neq$ NIL **then**
  return $TreeMin(right[x])$
end if
$y \leftarrow p[x]$
while $(y \neq$ NIL$)$ **and** $(x = right[y])$ **do**
  $x \leftarrow y$
  $y \leftarrow p[y]$
end while
return y

Le procedure $TreeSuccessor(x)$ e $TreePredecessor(x)$ hanno complessità pessima $\Theta(h)$, ove h è l'altezza dell'albero che contiene x.

Algoritmo 71 TreePredecessor(x)

TreePredecessor(x)

```
if left[x] ≠ NIL then
    return TreeMax(left[x])
end if
y ← p[x]
while (y ≠ NIL) and (x = left[y]) do
    x ← y
    y ← p[y]
end while
return y
```

La procedura $TreeInsert(T, z)$ inserisce il nodo z sempre come foglia. La sua complessità pessima è dunque $\Theta(h)$, ove h è l'altezza dell'albero T.

Esercizio 4.45 *1. Trovare una sequenza di* 15 *chiavi il cui inserimento con la procedura* TreeInsert *genera un albero completo.*

2. Trovare una sequenza di 15 *chiavi il cui inserimento con la procedura* TreeInsert *genera un albero lineare.*

3. Qual è il la complessità ottima e pessima di n inserimenti con la procedura TreeInsert*?*

Soluzione

1. La sequenza è $\langle 8, 4, 12, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15\rangle$. In generale, dato $n = 2^r - 1$, con $r \geq 1$, la sequenza di n chiavi il cui inserimento con la procedura *TreeInsert* genera un albero completo è ottenuta concatenando le r sequenze $S_{r-1}, S_{r-2}, \ldots, S_0$ tali che
$$S_i = \langle 2^i(1 + 2k) \, : \, k = 0, 1, \ldots, (n + 1)/2^{i+1} - 1\rangle.$$

2. La sequenza è $\langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\rangle$ oppure la sua inversa. In generale ogni sequenza di chiavi oridinate in qualche senso genera un albero lineare.

3. Nel caso pessimo, l'albero T risultante dopo gli n inserimenti è lineare, cioè la sequenza di inserimenti è ordinata. L'i-esimo inserimento avviene su un albero di altezza $i - 2$. La complessità di $TreeInsert$ è dell'ordine dell'altezza dell'albero in cui il nodo viene inserito, cioè $\Theta(i - 2) = \Theta(i)$. Dunque la complessità dell'inserimento delle n chiavi nell'albero T è pari a:
$$\sum_{i=1}^{n} \Theta(i) = \Theta(\sum_{i=1}^{n} i) = \Theta(n(n + 1)/2) = \Theta(n^2).$$

Algoritmo 72 TreeInsert(T,z)

TreeInsert(T,z)

```
// p[z] = left[z] = right[z] = NIL
y ← NIL
x ← root(T)
while x ≠ NIL do
   y ← x
   if key[z] < key[x] then
      x ← left[x]
   else
      x ← right[x]
   end if
end while
p[z] ← y
if y = NIL then
   root(T) ← z
else
   if key[z] < key[y] then
      left[y] ← z
   else
      right[y] ← z
   end if
end if
```

Nel caso ottimo, l'albero T risultante dopo gli n inserimenti è un albero completo fino al penultimo livello. In tal caso, dopo i inserimenti, con $1 \leq i \leq n$, l'altezza dell'albero corrente è pari a $\lfloor \log i \rfloor$. La complessità di $TreeInsert$ è dell'ordine dell'altezza dell'albero in cui il nodo viene inserito. Dunque la complessità dell'inserimento delle n chiavi nell'albero T è pari a

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

Vediamo infine la procedura di cancellazione. Ci servirà il seguente risultato.

Proposizione 4.3 *Se un nodo in un albero binario di ricerca ha due figli, allora il suo successore non ha figlio sinistro e il suo predecessore non ha figlio destro.*

Dimostrazione

Dimostriamo la proprietà per il successore. Sia x un nodo con entrambi i figli. Allora il suo successore y è il minimo del sottoalbero destro di x. Supponiamo per assurdo che y abbia un figlio sinistro z. Per la proprietà dell'albero binario di ricerca, $key[x] < key[z] < key[y]$, e quindi y non è il successore di x. Il ragionamento per il predecessore è simile. ■

Algoritmo 73 TreeDelete(T,z)

TreeDelete(T,z)

```
if (left[z] = NIL) or (right[z] = NIL) then
    y ← z
else
    y ← TreeSuccessor(z)
end if
if left[y] ≠ NIL then
    x ← left[y]
else
    x ← right[y]
end if
if x ≠ NIL then
    p[x] ← p[y]
end if
if p[y] = NIL then
    root(T) ← x
else
    if y = left[p[y]] then
        left[p[y]] ← x
    else
        right[p[y]] ← x
    end if
end if
if y ≠ z then
    key[z] ← key[y]
    // copia i dati satellite di y in z
end if
return y
```

La complessità di $TreeDelete(T, z)$ è costante se z non ha entrambi i figli, altrimenti è $\Theta(h)$, ove h è l'altezza dell'albero T.

Tutte le operazioni implementate sugli alberi binari di ricerca (a parte la procedura $TreeEmpty$) hanno dunque complessità $\Theta(h)$, ove h è l'altezza dell'albero. Un albero binario di ricerca può assumere una forma qualsiasi, e quindi, per l'esercizio 4.42, $h = \Omega(\log n)$ e $h = O(n)$. Nel caso ottimo, l'altezza dell'albero è logaritmica nel numero di nodi, e dunque la complessità ottima di tutte le operazioni risulta logaritmica. Nel caso pessimo, l'altezza dell'albero è lineare nel numero dei nodi, e dunque la complessità pessima di tutte le operazioni risulta lineare, non meglio delle operazioni su liste. Fortunatamente, la complessità media è logaritmica, come afferma il seguente teorema.

Teorema 4.1 *Supponiamo di avere n chiavi distinte a disposizione. Scegliamo casualmente una permutazione delle n chiavi e inseriamo la permutazione in un* *albero binario inzialmente vuoto usando la procedura* TreeInsert*. Allora l'altezza media dell'albero risultante è* $O(\log n)$.

4.5.1 Visite di alberi

In questa sezione vedremo come è possibile visitare i nodi di un albero. Possiamo farlo in due modi: in **profondità** e in **ampiezza**. Vi sono tre modi per visitare in profondità un albero binario: visita intermedia, visita anticipata e visita posticipata. Sia T una albero binario e x un nodo di T. L'**ordinamento intermedio**, detto anche **sinistra-radice-destra**, (**inorder** in inglese) dei nodi di T è definito nel seguente modo: $x < y$ se $x \in T(left[y])$ oppure $y \in T(right[x])$ oppure esiste $z \in T$ tale che $x \in T(left[z])$ e $y \in T(right[z])$. Una visita intermedia di un albero visita i suoi nodi in ordine intermedio. Sostanzialmente la visita parte dalla radice ed esplora in ordine intermedio tutti i nodi del sottoalbero di sinistra della radice. Poi la visita esplora la radice e infine esplora in ordine intermedio tutti i nodi del sottalbero destro della radice. Vediamo una procedura ricorsiva $TreeInOrderVisit(x)$ che stampa in ordine intermedio i nodi dell'albero radicato in x.

Algoritmo 74 TreeInOrderVisit(x)

TreeInOrderVisit(x)

if $x \neq$ NIL **then**
  $TreeInOrderVisit(left[x])$
  print $key[x]$
  $TreeInOrderVisit(right[x])$
end if

La complessità della procedura $TreeInOrderVisit(x)$ è $\Theta(n)$, ove n è il numero di nodi dell'albero radicato in x. Infatti, l'equazione ricorsiva che ne descrive la complessità è $C(0) = 1$ e, per $n > 0$, $C(n) = C(a) + C(b) + 1$, con $a, b \geq 0$, e $a+b = n-1$. Per l'esercizio 2.10, punto 1, la soluzione è $C(n) = \Theta(n)$.

Esercizio 4.46 *Si scriva versione iterativa di $TreeInOrderVisit(x)$.*

Soluzione

Algoritmo 75 TreeInOrderVisit(x)

TreeInOrderVisit(x)

```
S ← ∅
while x ≠ NIL do
    Push(S, x)
    x ← left[x]
end while
while not StackEmpty(S) do
    x ← Pop(S)
    print key[x]
    y ← right[x]
    while y ≠ NIL do
        Push(S, y)
        y ← left[y]
    end while
end while
```

La complessità della versione iterativa di $TreeInOrderVisit(x)$ rimane lineare nel numero di nodi dell'albero radicato in x. Sia n il numero di nodi dell'albero radicato in x. Ogni nodo dell'albero viene caricato e scaricato dalla pila S esattamente una volta. Dato che le operazioni $Push$ e Pop hanno costo costante, la complessità risulta $\Theta(n)$.

L'**ordinamento anticipato**, detto anche **radice-sinistra-destra** (oppure **preorder** in inglese) dei nodi di T è definito nel seguente modo: $x < y$ se $y \in T(x)$ oppure esiste $z \in T$ tale che $x \in T(left[z])$ e $y \in T(right[z])$. Una visita anticipata di un albero visita i suoi nodi in ordine anticipato. Sostanzialmente la visita esplora prima la radice, poi in ordine anticipato tutti i nodi del sottoalbero di sinistra della radice e infine esplora in ordine anticipato tutti i nodi del sottalbero destro della radice. Vediamo una procedura ricorsiva $TreePreOrderVisit(x)$ che stampa in ordine intermedio i nodi dell'albero radicato in x.

Vediamo una procedura ricorsiva che stampa in ordine anticipato i nodi dell'albero radicato in x.

Algoritmo 76 TreePreOrderVisit(x)

TreePreOrderVisit(x)

```
if x ≠ NIL then
    print key[x]
    TreeInOrderVisit(left[x])
    TreeInOrderVisit(right[x])
end if
```

Infine, l'**ordinamento posticipato**, detto anche **sinistra-destra-radice** (oppure **postorder** in inglese) dei nodi di T è definito nel seguente modo: $x < y$ se $x \in T(y)$ oppure esiste $z \in T$ tale che $x \in T(left[z])$ e $y \in T(right[z])$. Una visita postipata di un albero visita i suoi nodi in ordine posticipato. Sostanzialmente la visita esplora in ordine posticipato tutti i nodi del sottoalbero di sinistra della radice, poi in ordine posticipato tutti i nodi del sottalbero destro della radice ed infine la radice. Vediamo una procedura ricorsiva $TreePostOrderVisit(x)$ che stampa in ordine intermedio i nodi dell'albero radicato in x. Ecco una procedura ricorsiva che stampa in ordine posticipato i nodi dell'albero radicato in x.

Algoritmo 77 TreePostOrderVisit(x)

TreePostOrderVisit(x)

```
if x ≠ NIL then
    TreeInOrderVisit(left[x])
    TreeInOrderVisit(right[x])
    print key[x]
end if
```

Vediamo ora come visitare un albero in ampiezza. Sia T una albero binario. Per $i \geq 0$, definiamo $L_i(T)$ come l'i-esimo livello di T, cioè l'insieme dei nodi di T che hanno profondità i. L'**ordinamento per livelli** dei nodi di T è definito ricorsivamente nel seguente modo: $x < y$ se $x \in L_i(T)$, $y \in L_j(T)$ e $i < j$, oppure $x, y \in L_i(T)$ ed esiste $z \in L_{i-1}(T)$ tale che $x = left[z]$ e $y = right[z]$, oppure $x, y \in L_i(T)$ ed esistono $z_1, z_2 \in L_{i-1}(T)$ tale che $z_1 < z_2$ e $x \in T(z_1)$ e $y \in T(z_2)$. Una visita per livelli di un albero esplora in nodi dell'albero secondo l'ordinamento per livelli, cioè dal primo livello (quello della radice) all'ultimo livello (quello delle foglie). All'inteno di ciascun livello i nodi vengono visitati da sinistra a destra. Vediamo una procedura $TreeBreadthFirstVisit(x)$ che stampa in ordine per livelli i nodi dell'albero radicato in x.

Algoritmo 78 TreeBreadthFirstVisit(x)

TreeBreadthFirstVisit(x)

```
Q ← ∅
if x ≠ NIL then
  Enqueue(Q, x)
end if
while not QueueEmpty(Q) do
  y ← Dequeue(Q)
  print key[y]
  if left[y] ≠ NIL then
    Enqueue(Q, left[y])
  end if
  if right[y] ≠ NIL then
    Enqueue(Q, right[y])
  end if
end while
```

La complessità di $TreeBreadthFirstVisit(x)$ è lineare nel numero di nodi dell'albero radicato in x. Sia n il numero di nodi dell'albero radicato in x. Ogni nodo dell'albero viene inserito e cancellato dalla coda Q esattamente una volta. Dato che le operazioni *Enqueue* e *Dequeue* hanno costo costante, la complessità risulta $\Theta(n)$. Dunque la complessità di una visita (qualsiasi) in profondità di un albero equivale alla complessità di una visita in ampiezza dell'albero.

Esercizio 4.47 *Sia T un albero binario di ricerca con n nodi. Sia scriva una procedura $TreePrintSorted(T)$ di complessità $O(n)$ che stampa in ordine crescente in nodi di T.*

Soluzione

Prima soluzione.

Algoritmo 79 TreePrintSorted(T)

TreePrintSorted(T)

```
TreeInOrderVisit(root(T))
```

Seconda soluzione.

Algoritmo 80 TreePrintSorted(T)

TreePrintSorted(T)

```
if root[T] ≠ NIL then
   x ← TreeMin(root(T))
   while x ≠ NIL do
      print key[x]
      x ← TreeSuccessor(x)
   end while
end if
```

La prima procedura ha la complessità di $TreeInOrderVisit$, e dunque $\Theta(n)$. Per quanto rigurda la seconda procedura, si noti che durante l'esecuzione l'albero T viene visitato in ordine intermedio. Dunque ogni nodo x di T viene visitato al più tre volte: la prima quando arrivo al nodo x dal padre di x, la seconda quando arrivo al nodo x dal figlio sinistro di x, e la terza quando arrivo al nodo x dal figlio destro di x. Dunque la complessità è $\Theta(3n) = \Theta(n)$.

Esercizio 4.48 *Sia A un vettore. Si consideri la seguente procedura di ordinamento $TreeSort(A)$. La procedura consiste di due passi:*

1. *gli elementi di A vengono inseriti in un albero binario di ricerca T usando la procedura $TreeInsert$;*
2. *l'albero T viene visitato in ordine intermedio e gli elementi di T vengono reinseriti nel vettore A.*

Si calcoli la complessità ottima, media e pessima di $TreeSort$.

Soluzione

Sia n la lunghezza di A. Nel caso pessimo, l'albero T risultante dopo gli n inserimenti degli elementi di A è lineare e, per l'Esercizio 4.45, la complessità dell'inserimento degli n elementi di A nell'albero T è pari a $\Theta(n^2)$. La complessità pessima della visita intermedia di T è $\Theta(n)$. Dunque la complessità pessima di $TreeSort$ risulta $\Theta(n^2 + n) = \Theta(n^2)$.

Nel caso ottimo, l'albero T risultante dopo gli n inserimenti degli elementi di A è un albero completo fino al penultimo livello. In tal caso, per l'Esercizio 4.45, la complessità dell'inserimento degli n elementi di A nell'albero T è pari a $\Theta(n \log n)$. La complessità ottima della visita intermedia di T è $\Theta(n)$. Dunque la complessità ottima di $TreeSort$ risulta $\Theta(n \log n + n) = \Theta(n \log n)$.

Nel caso medio, per il Teorema 4.1, l'albero T risultante dopo gli n inserimenti degli elementi di A è un albero bilanciato con altezza logaritmica nel numero dei nodi. Dunque la complessità media è pari alla complessità ottima, cioè $\Theta(n \log n)$.

4.6 Tabelle hash

Una **tabella hash** (in inglese **hash table**) è una struttura di dati efficiente per implementare **dizionari**, cioè insiemi dinamici che supportano le operazioni di inserimento, cancellazione e ricerca. Sotto ragionevoli assunzioni, le tabelle hash implementano queste tre operazioni fondamentali con **complessità media costante** $\Theta(1)$. Dunque, nel caso medio, la ricerca su tabelle hash è molto più efficiente della ricerca su liste, il cui costo medio è lineare $\Theta(n)$, e batte in efficienza anche la ricerca su alberi binari di ricerca, la cui complessità media è logaritmica $\Theta(\log n)$. Le altre operazioni, come massimo, minimo, successore e predecessore, sono invece inefficienti su tabelle hash. Un tipico esempio di applicazione delle tabelle hash è la tabella dei simboli che il compilatore mantiene, nella quale vengono inserite stringhe che corrispondono agli identificatori usati nel programma.

L'idea principale delle tabelle hash è di usare un **accesso** (o **indirizzamento**) **diretto** agli elementi dell'insieme. Dunque una tabella hash è una generalizzazione di un vettore, nel quale l'accesso ad un qualsiasi elemento ha costo costante $\Theta(1)$.

4.6.1 Tabelle ad indirizzamento diretto

Supponiamo di voler rappresentare un insieme dinamico di oggetti in cui ogni oggetto ha un campo chiave e altri campi contenenti dati satellite. In questa sezione, il campo chiave identifica univocamente un oggetto, cioè due oggetti sono diversi se e soltanto se hanno chiave diversa. Sia $U = \{0, 1, \ldots, u - 1\}$ l'universo (o spazio) di tutte le possibili chiavi. Se U non è molto grande, possiamo usare una tabella ad indirizzamento diretto, cioè un vettore $T[0 \ldots u-1]$, per rappresentare un insieme dinamico di oggetti le cui chiavi appartengono ad U. Ogni posizione i del vettore T contiene un puntatore all'oggetto con chiave i, se tale oggetto è presente in tabella, oppure NIL, se tale oggetto non è presente in tabella. Dunque le operazioni di inserimento, cancellazione e ricerca possono essere implementate con complessità costante come segue:

Algoritmo 81 DirectAddressInsert(T,x)

DirectAddressInsert(T,x)

1: $T[key[x]] \leftarrow x$

Algoritmo 82 DirectAddressDelete(T,x)

DirectAddressDelete(T,x)

1: $T[key[x]] \leftarrow$ NIL

Algoritmo 83 DirectAddressSearch(T,k)

DirectAddressSearch(T,k)

```
1: return T[k]
```

Questa soluzione ha però alcuni inconvenienti. La dimensione della tabella è pari alla cardinalità u dello spazio U delle chiavi. Se u è molto grande, è costoso e talvolta impossibile creare un vettore di dimensione u. Inoltre, lo spazio allocato è indipendente dal numero di elementi effettivamente inseriti nel vettore. Questo porta ad uno spreco di memoria qualora il numero di elementi inseriti nella tabella sia di molto inferiore alla dimensione della tabella. Infine, l'informazione sulla chiave degli oggetti è ridondante, in quanto corrisponde sia all'indice del vettore che al campo chiave dell'oggetto.

Esercizio 4.49 *Implementare le operazioni di minimo, massimo, successore e predecessore su tabelle ad indirizzamento diretto. Si calcolino le complessità ottima e pessima delle procedure. Si congetturi inoltre la loro complessità media.*

Esercizio 4.50 *Si modifichi opportunamente la struttura di dati tabella ad indirizzamento diretto in modo che possa contenere più oggetti con la medesima chiave (ma con dati satellite diversi). Le operazioni di inserimento, cancellazione e ricerca debbono avere costo costante.*

4.6.2 Tabelle hash

Sia $K \subseteq U$ l'insieme delle chiavi effettivamente contenute nel dizionario e siano $n = |K|$ e $u = |U|$. Quando u è molto grande e n è modesto rispetto a u, è preferibile usare una tabella hash piuttosto di una tabella ad indirizzamento diretto per rappresentare K. Sia m la dimensione di una **tabella hash** $T[0 \ldots m-1]$. Useremo una **funzione hash** $h : U \rightarrow \{0, 1, \ldots, m-1\}$ per mappare ogni chiave k nell'universo U in una posizione o **slot** $h(k)$ di T. Diremo che $h(k)$ è il **valore di hash** di k. La tabella hash $T[0 \ldots m-1]$ è un vettore che contiene in posizione i un puntatore all'oggetto x tale che $h(key[x]) = i$. Ci aspettiamo che $m << u$. Dunque sono possibili **collisioni**, cioè chiavi diverse k_1 e k_2 tali che $h(k_1) = h(k_2)$. In questo caso sorge il problema di gestire la collisione in qualche modo. Ci aspettiamo che le collisioni siano rare quando n è piccolo rispetto a m, e diventino più frequenti quando n si avvicina a m. Quando $n \geq m$ le collisioni sono inevitabili. Una buona funzione hash è una funzione che distribuisce le chiavi in modo uniforme sulle posizioni della tabella e quindi minimizza le collisioni quando possibile.

Facciamo due esempi. Il dizionario della lingua inglese contiene parole su un alfabeto di 26 caratteri. Supponendo che una parola non abbia lunghezza superiore a 15, abbiamo 26^{15} possibili parole (chiavi) nell'universo. Di queste, solo una minima parte hanno un senso, e quindi corrispondono a parole (chiavi) effettivamente contenute nel dizionario. Secondo esempio. Una tabella dei simboli contiene gli identificatori usati in un programma. Supponendo che un identificatore sia una stringa alfanumerica di al più 8 caratteri, vi sono 36^8 possibili chiavi, mentre gli identificatori usati in un programma sono generalmente molto meno. In entrambi i casi è preferibile usare una tabella hash piuttosto di una tabella ad indirizzamento diretto. Solitamente non si conosce il numero n di chiavi che verranno effettivamente inserite nel dizionario. Ma questo numero può essere stimato. Una volta stimato n, la dimensione m della tabella hash deve essere proporzionalmente a n. Nell'esempio della tabella dei simboli, se assumiamo di usare circa $n = 1500$ identificatori nei nostri programmi, allora potremmo tenere una tabella hash di dimensione $m = 500$, in modo che, in media, abbiamo 3 collisioni per chiave.

Due sono le questioni da discutere: cosa caratterizza una buona funzione hash e come gestire le collisioni, inevitabili quando $n \geq m$. Esaminiamo queste due questioni separatamente.

Funzioni hash

Una **buona funzione hash** deve (1) essere facile da calcolare (complessità costante), (2) soddisfare il requisito di **uniformità semplice**: ogni chiave ha la medesima probabilità di vedersi assegnato un qualsiasi valore di hash ammissibile, indipendentemente dagli altri valori di hash già assegnati. Il termine inglese hash significa tra l'altro *incasinare* e denota anche il nome che viene dato al polpettone, cioè al piatto di carne avanzata dal giorno prima e tritata assieme alle verdure. Il requisito di uniformità semplice è difficile da verificare perché in generale le chiavi non vengono estratte in modo casuale e non si conosce la distribuzione di probabilità con la quale le chiavi vengono generate. Si pensi all'esempio della tabella dei simboli. Non possiamo aspettarci che ogni identificatore abbia la medesima probabilità di comparire in un programma. Ad esempio *length* ha ragionevolmente più probabilità di essere usato rispetto a *htgnel*. Esistono inoltre correlazioni tra gli identificatori scelti. Ad esempio simboli solitamente usati nello stesso programma sono *pt* e *pts*, oppure x e y, o ancora n e m. Una buona funzione hash deve tenere in considerazione questa informazione qualitativa e evitare che chiavi correlate vadano a collidere.

Solitamente per generare una buona funzione hash si procede in maniera euristica, cioè accontentandoci di funzioni che si avvicinano all'ipotesi di uniformità semplice. Assumiamo che l'universo delle chiavi sia l'insieme dei numeri naturali. Ogni stringa può essere convertita in modo univoco in un numero naturale: ad esempio *pt* corrisponde alla coppia $(112, 116)$, ove 112 è il codice ASCII di p e 116 quello di t. La coppia $(112, 116)$ tradotta in base 128 è il numero $112 \cdot 128 + 115 = 14452$. Il **metodo della divisione** per generare funzioni hash consiste nell'associare alla chiave k il valore di hash $h(k) = k \bmod m$. Occorre evitare che m sia una potenza di 2. Se $m = 2^p$, allora $k \bmod m$ corrisponde ai p bit meno significativi di k. Questo limita la casualità della funzione hash, in quanto essa è funzione di una porzione (di dimensione logaritmica) della chiave. Una buona scelta è un numero primo non troppo vicino ad una potenza di due. Il **metodo della moltiplicazione** per generare funzioni hash è il seguente. Sia $0 < A < 1$ una costante. Data una chiave k, il valore di hash

$h(k) = \lfloor m \cdot (kA - \lfloor kA \rfloor) \rfloor$. Questo metodo ha il vantaggio che il valore di m non è critico. Possiamo scegliere m come una potenza di 2 per facilitare il calcolo della funzione hash. Per quanto riguarda il valore di A, viene suggerito un valore prossimo a $(\sqrt{5} - 1)/2$.

Vi sono principalmente due modi per risolvere le collisioni: risoluzione mediante concatenazione e risoluzione mediante indirizzamento aperto.

Risoluzione delle collisioni mediante concatenazione

Con questo metodo, una tabella hash $T[0 \dots m-1]$ contiene in posizione i un puntatore alla testa di una lista di oggetti x per cui $h(key[x]) = i$. Il puntatore è NIL se la lista è vuota. Dunque risolvo le collisioni concatenando in una lista tutti gli oggetti con medesimo valore di hash. Supponiamo di adottare per le liste un modello bidirezionale, senza l'attributo *head* (la testa della lista è già contenuta nella tabella T). Le operazioni di inserimento, cancellazione e ricerca sono dunque le seguenti:

Algoritmo 84 ChainedHashInsert(T,x)

ChainedHashInsert(T,x)

$i \leftarrow h(key[x])$
$next[x] \leftarrow T[i]$
$prev[x] \leftarrow$ NIL
if $T[i] \neq$ NIL **then**
    $prev[T[i]] \leftarrow x$
end if
$T[i] \leftarrow x$

Algoritmo 85 ChainedHashDelete(T,x)

ChainedHashDelete(T,x)

$i \leftarrow h(key[x])$
if $prev[x] \neq$ NIL **then**
    $next[prev[x]] \leftarrow next[x]$
else
    $T[i] \leftarrow next[x]$
end if
if $next[x] \neq$ NIL **then**
    $prev[next[x]] \leftarrow prev[x]$
end if

Vediamo ora di analizzare la complessità delle procedure introdotte. La complessità dipende dal tipo di funzione hash h che abbiamo usato. Supponiamo inizialmente che h verifichi solo la prima condizione di una buona funzione hash, cioè il costo di calcolo di h è costante $\Theta(1)$. L'inserimento e la cancellazione hanno quindi complessità costante $\Theta(1)$. Si noti che nell'inserimento

Algoritmo 86 ChainedHashSearch(T,k)

ChainedHashSearch(T,k)

```
x ← T[h(k)]
while x ≠ NIL and key[x] ≠ k do
    x ← next[x]
end while
return x
```

non controlliamo se l'oggetto da inserire ha una chiave già presente in tabella. Se volessimo fare questo controllo, dovremmo prima fare una ricerca, e poi eventualmente fare l'inserimento. In questo caso l'inserimento non avrebbe più una complessità costante. La complessità della ricerca è proporzionale alla lunghezza della lista in cui cerco. Il caso pessimo si ha quando tutti gli n oggetti presenti in tabella sono concatenati in una unica lista. Ad esempio, se poco saggiamente usiamo la funzione hash $h(k) = 0$, allora tutti gli inserimenti verranno effettuati nella lista $T(0)$. In questo caso la complessità pessima della ricerca è $\Theta(n)$, e quella media è $\Theta(n/2) = \Theta(n)$, come avviene per le liste.

In genere, useremo delle funzioni hash più intelligenti della funzione costante 0. Questo ci garantirà un costo inferiore, almeno nel caso medio. Assumiamo di usare una buona funzione hash h, cioè una funzione che si calcola con costo costante e che soddisfa l'ipotesi di uniformità semplice. Il caso pessimo non cambia: potremmo sventuratamente inserire nella tabella n chiavi che collidono, generando così una unica lista di concatenazione di lunghezza n. La ricerca dunque avrebbe complessità lineare $\Theta(n)$. Questo è però un caso eccezionale, come dimostreremo ora. In media, le cose vanno meglio. Definiamo il **fattore di carico** della tabella di hash come $\alpha = n/m$. Si noti che α è un numero *razionale* maggiore o uguale a 0. In particolare il fattore di carico è nullo quando $n = 0$ e il fattore di carico è uguale a 1 quando $n = m$. Dato che h soddisfa l'ipotesi di uniformità semplice, α corrisponde alla lunghezza media di una qualsiasi lista di concatenazione. Distinguiamo i casi di ricerca con insuccesso (cerchiamo una chiave che non c'è) e di ricerca con successo (cerchiamo una chiave esistente). In generale, ci aspettiamo che il costo medio di una ricerca con successo sia minore del costo di una ricerca con insuccesso. Infatti, in media, la ricerca con successo termina trovando l'elemento cercato prima della fine della lista da scandire.

Teorema 4.2 *In una tabella hash con risoluzione delle collisioni mediante concatenazione, adottando una buona funzione hash il costo medio della ricerca con insuccesso è* $\Theta(1 + \alpha)$.

Dimostrazione

La ricerca con insuccesso consiste nel calcolo di un valore della funzione hash per individuare la lista di concatenazione in cui cercare e nella scansione totale di tale lista. Il costo del calcolo della funzione hash è $\Theta(1)$. La lunghezza media di una lista di concatenazione è α, e dunque il costo medio di una ricerca con insuccesso è pari a $\Theta(1 + \alpha)$.

Teorema 4.3 *In una tabella hash con risoluzione delle collisioni mediante concatenazione, adottando una buona funzione hash il costo medio della ricerca con successo è* $\Theta(1 + \alpha)$.

Dimostrazione

La ricerca con successo consiste nel calcolo di un valore della funzione hash per individuare la lista di concatenazione in cui cercare e nella scansione di tale lista fino a trovare l'elemento cercato. Mediamente dovrò scandire metà della lista prima di trovare l'elemento cercato. Il costo del calcolo della funzione hash è $\Theta(1)$. La lunghezza media di una lista di concatenazione è α. Dunque il costo medio di una ricerca con successo è pari a $\Theta(1 + \alpha/2) = \Theta(1 + \alpha)$.

Dunque, asintoticamente, cercare con successo costa come cercare con insuccesso. Se $n = O(m)$, allora $\alpha = n/m = O(m)/m = O(1)$ e il costo medio della ricerca risulta costante $\Theta(1)$. In questo caso tutte le operazioni sul dizionario hanno complessità media costante.

Risoluzione delle collisioni mediante indirizzamento aperto

Con questa tecnica, la collisione viene risolta utilizzando le posizioni libere della tabella hash, se ne esistono, per contenere gli oggetti che collidono. Di conseguenza, tutti gli oggetti sono archiviati all'interno della tabella hash e il fattore di carico $\alpha = n/m$ è sempre minore o uguale a 1. La funzione hash in questo caso determina una sequenza di m posizioni distinte della tabella hash che debbono essere sondate al fine di inserire o cercare un elemento nella tabella. Quindi una funzione hash ora è una funzione binaria $h : U \times \{0, 1, \ldots m-1\} \rightarrow \{0, 1, \ldots m-1\}$ tale che, per ogni chiave $k \in U$, la **sequenza di scansione** $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$ è una permutazione di $\langle 0, 1, \ldots, m-1 \rangle$. L'inserimento di un elemento visita la sequenza di scansione fino a trovare eventualmente una posizione libera dove mettere l'elemento. Se non esistono posizioni libere, allora l'inserimento non è possibile. La ricerca esplora la sequenza di scansione fino a che trova l'elemento cercato oppure trova una posizione libera. Infatti, l'elemento cercato non potrebbe essere stato inserito dopo la prima posizione libera.

Algoritmo 87 OpenAdressingInsert(T,x)

OpenAdressingInsert(T,x)

```
i ← 0
k ← key[x]
repeat
    j ← h(k, i)
    if T[j] = NIL then
        T[j] ← x
    else
        i ← i + 1
    end if
until i = m
error OVERFLOW
```

Algoritmo 88 OpenAdressingSearch(T,k)

OpenAdressingSearch(T,k)

```
i ← 0
repeat
    j ← h(k, i)
    if key[T[j]] = k then
        return T[j]
    else
        i ← i + 1
    end if
until T[j] = NIL  or  i = m
return NIL
```

Si noti che la procedura di ricerca torna, in caso di successo, la posizione j dell'oggetto trovato $T[j]$, e non l'oggetto stesso. Per cancellare un elemento, non posso semplicemente mettere a NIL la corrispondente posizione nella tabella. Se facessi in questo modo, la procedura di ricerca non sarebbe più corretta, perché potrebbe terminare con insuccesso anche in presenza dell'elemento cercato. Una soluzione è marcare un elemento cancellato con la costante DELETED diversa da NIL e modificare la procedura di inserimento in modo che tratti gli elementi cancellati (marchiati DELETED) come se fossero liberi (marchiati NIL). La procedura di ricerca rimane invariata, perché non distingue un elemento pieno da uno cancellato. Questa implementazione ha lo svantaggio che il tempo speso dalla ricerca non è più proporzionale al fattore di carico.

Esercizio 4.51 *Si implementi l'operazione di cancellazione usando la costante* DELETED *per indicare che un elemento è stato cancellato. Si individui un caso in cui il fattore di carico è nullo ma la ricerca impiega tempo* $\Theta(m)$ *per cercare una qualsiasi chiave.*

Quando gli oggetti non contengono dati satellite, posso identificare la chiave k con l'oggetto x con chiave k, e dunque inserire direttamente le chiavi nella tabella, risparmiando lo spazio usato per i puntatori.

Vediamo quali sono le caratteristiche di una **buona funzione hash binaria**. Sia $h : U \times \{0, 1, \ldots m - 1\} \rightarrow \{0, 1, \ldots m - 1\}$ una funzione hash binaria. La funzione h associa ad ogni chiave k nell'universo la sequenza di scansione $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1)\rangle$ che è una permutazione di $\langle 0, 1, \ldots, m-1\rangle$. Una buona funzione hash deve: (1) essere facile da calcolare (complessità costante), (2) soddisfare il requisito di **uniformità**: ogni chiave ha la medesima probabilità di vedersi assegnata una qualsiasi delle $m!$ sequenze di scansione. Come nel caso di funzioni hash unarie, è difficile ottenere buone funzioni hash, e quindi ci accontentiamo di euristiche per generare funzioni hash soddisfacenti. Tre euristiche sono le seguenti:

1. **Scansione lineare**: data una funzione hash unaria $h' : U \rightarrow \{0, 1, \ldots m-1\}$, questo metodo definisce la funzione hash binaria come

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

2. **Scansione quadratica**: data una funzione hash unaria $h' : U \rightarrow \{0, 1, \ldots m-1\}$, e due costanti c_1 e $c_2 \neq 0$, questo metodo definisce la funzione hash binaria come

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

3. **Hashing doppio**: date due funzioni di hash unarie h_1 e h_2, questo metodo definisce la funzione hash binaria come

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m.$$

In sostanza, la differenza tra i tre metodi è la seguente. Tutti e tre i metodi fanno partire la sequenza di scansione di una chiave k da un valore determinato da una funzione hash ausiliaria. La differenza consiste nel passo, o spiazzamento, che determina la distanza tra due consecutive posizioni nella sequenza di scansione. Nel primo caso il passo è lineare, nel secondo è quadratico, nel terzo è casuale. Si noti che il numero di sequenze di scansione generate dai primi due metodi è m, mentre è m^2 nel terzo caso. Infatti, nei primi due casi, ogni sequenza è completamente determinata dal suo primo elemento, cioè da $h'(k)$. Dato che $h'(k)$ può assumere valori da 0 a $m - 1$, le possibili sequenze di scansione sono m. Nel terzo caso una sequenza viene identificata da $h_1(k)$ e da $h_2(k)$, ognuno dei quali può assumere valori da 0 a $m - 1$. Dunque ci sono m^2 possibili sequenze. In entrambi i casi, il numero di sequenze di scansione è inferiore al numero ideale di $m!$. Il metodo di scansione lineare soffre del **problema della accumulazione**: le posizioni occupate si accumulano in lunghi tratti contigui, aumentando il tempo medio di ricerca e di inserimento. Il metodo di scansione quadratica soffre dello stesso problema ma in forma minore, in quanto lo spiazzamento è quadratico. Il metodo di hashing doppio non ha questo problema perché il passo è casuale. Queste considerazioni fanno preferire il metodo di hashing doppio ai primi due.

Analizziamo ora la complessità delle procedure di inserimento e di ricerca con il metodo di indirizzamento aperto. Il caso pessimo per la procedura di inserimento si ha quando la tabella è piena, cioè $n = m$. In tal caso, l'inserimento non è possibile, e la procedura di inserimento esegue $\Theta(n)$ passi. Il caso pessimo per la procedura di ricerca si ha quando la tabella è piena, cioè $n = m$, e la ricerca non ha successo. In tal caso, la procedura di ricerca esegue $\Theta(n)$ passi. Quindi la complessità pessima equivale alla complessità pessima con il metodo di concatenazione.

Nel caso medio la complessità è la seguente.

Teorema 4.4 *In una tabella hash con risoluzione delle collisioni mediante indirizzamento aperto e con fattore di carico $\alpha < 1$, adottando una buona funzione hash binaria il costo medio della ricerca con insuccesso è al più $1/(1-\alpha)$.*

Dimostrazione

Dato che il costo di calcolo della funzione hash è costante, è sufficiente contare il numero medio di ispezioni di posizioni della tabella hash in caso di ricerca con insuccesso. Nell'ipotesi di uniformità, la probabilità di trovare una posizione occupata nella tabella hash è pari al fattore di carico $\alpha = n/m$. Nel caso di ricerca con insuccesso, una ispezione viene sempre fatta. Ne faccio una seconda qualora la prima posizione ispezionata sia occupata, cioè con probabilità α. Ne faccio una terza qualora le prime due posizioni ispezionate siano occupate. Assumendo l'ipotesi di uniformità, questo avviene con probabilità $\alpha \cdot \alpha = \alpha^2$. E così via fino a quando tutta la tabella è stata ispezionata. Dunque il numero medio di posizioni ispezionate sarà

$$1 + \alpha + \alpha^2 + \dots \alpha^{m-1} \leq \sum_{i=0}^{\infty} \alpha^i = 1/(1-\alpha).$$

Teorema 4.5 *In una tabella hash con risoluzione delle collisioni mediante indirizzamento aperto e con fattore di carico $\alpha < 1$, adottando una buona funzione hash binaria il costo medio della ricerca con successo è al più $(1/\alpha)\ln(1/(1-\alpha))$.*

Dimostrazione

Dato che il costo di calcolo della funzione hash è costante, è sufficiente contare il numero medio di ispezioni di posizioni della tabella hash in caso di ricerca con successo. Il numero di ispezioni di una ricerca con successo della chiave k corrisponde al numero di ispezioni di una ricerca con insuccesso dopo che la chiave k è stata cancellata. Sia β il fattore di carico della ricerca con insuccesso sulla tabella priva della chiave k e sia α il fattore di carico della ricerca con successo sulla tabella con chiave k. Fissato β, per il Teorema 4.4, il costo della ricerca con insuccesso è $1/(1-\beta)$. Il costo della ricerca con successo è la media del costo della ricerca con insuccesso al variare di β tra 0 e α. Dato che β è un razionale e la funzione $1/(1-\beta)$ è crescente, tale media risulta essere minore o uguale a:

$$(1/\alpha)\int_0^\alpha 1/(1-\beta)d\beta = (1/\alpha)[\ln(1/(1-\beta))]_0^\alpha = (1/\alpha)\ln(1/(1-\alpha)).$$

Infine, dato che un inserimento corrisponde ad una ricerca con insuccesso più una operazione di assegnamento, abbiamo il seguente corollario.

Corollario 4.1 *In una tabella hash con risoluzione delle collisioni mediante indirizzamento aperto e con fattore di carico $\alpha < 1$, adottando una buona funzione hash binaria il costo medio dell'inserimento è al più $1/(1-\alpha)$.*

Esercizio 4.52 *Si confrontino le complessità della ricerca con metodo di concatenazione e della ricerca con metodo di indirizzamento diretto qualora il fattore di carico α sia inferiore a 1. Dire per quali valori di α conviene usare il primo metodo rispetto al secondo.*

Soluzione

Si dimostra analiticamente che per ogni $0 < \alpha < 1$,

$\frac{1}{1-\alpha} > 1 + \alpha$

$\frac{1}{\alpha}\ln\frac{1}{1-\alpha} > 1 + \frac{\alpha}{2}$

Dunque, in ogni caso, la complessità del metodo con indirizzamento aperto è maggiore della complessità del metodo con concatenazione. Le due complessità sono vicine se il fattore di carico è basso. Inoltre, abbiamo una conferma del fatto che cercare con insuccesso costa mediamente più che cercare con successo. Infatti, per ogni $0 < \alpha < 1$:

$\frac{1}{1-\alpha} > \frac{1}{\alpha}\ln\frac{1}{1-\alpha}$

$1 + \alpha > 1 + \frac{\alpha}{2}$

In sostanza, il metodo con indirizzamento aperto è da preferire al metodo con concatenazione quando tutte e tre queste condizioni si verificano (1) non prevediamo di cancellare elementi (2) il fattore di carico rimane modesto (3) non ci sono dati satellite. In tal caso infatti l'assenza di dati satellite ci permette di archiviare gli oggetti diretamente nella tabella risparmiando lo spazio per i puntatori. La memoria risparmiata può essere impiegata per allungare la tabella e quindi migliorare le prestazioni del dizionario.

4.7 Heap

La struttura di dati **heap** rappresenta un insieme dinamico mediante un albero binario quasi completo le cui chiavi soddisfano la seguente **proprietà degli heap**: la chiave di ogni nodo è maggiore o uguale della chiave dei propri figli. La struttura di dati heap possiede un attributo $heapsize(A)$ che contiene il numero degli elementi inseriti nello heap A, e le seguenti operazioni:

- $HeapEmpty(A)$, che verifica se lo heap A è vuoto;
- $HeapFull(A)$, che verifica se lo heap A è pieno;
- $HeapBuild(A)$, che costruisce uno heap a partiere da un vettore A qualsiasi;
- $HeapMax(A)$ che ritorna l'elemento massimo dello heap A;
- $HeapInsert(A, x)$ che inserisce nello heap A un nuovo elemento x;
- $HeapDelete(A, i)$ che rimuove dallo heap A l'elemento con indice i;
- $HeapModofyKey(A, i, k)$ che modifica la chiave dell'elemento con indice i nello heap A facendogli assumere il valore di k.

Implementiamo uno heap usando i vettori inserendo gli elementi per livelli, da sinistra a destra, partendo dalla radice. Dunque la radice occupa la posizione 1 nel vettore, e, dato un nodo in posizione i, l'indice del nodo padre (se esiste) è $\lfloor i/2 \rfloor$, l'indice del figlio sinistro (se esiste) è $2i$ e l'indice del figlio destro (se esiste) è $2i+1$. L'attributo $heapsize(A) \leq length(A)$ contiene l'indice dell'ultimo elemento dello heap. Dunque gli elementi del vettore A che corrispondono ad elemento dello heap sono $A[1], A[2], \ldots A[heapsize(A)]$, mentre le posizioni $heapsize(A)+1, heapsize(A)+2, \ldots, length(A)$ del vettore A sono libere. Sia A uno heap di dimensione $n = heapsize(A)$. Si noti che i nodi interni dello heap hanno indice da 1 a $\lfloor n/2 \rfloor$, mentre le foglie dello heap hanno indice da $\lfloor n/2 \rfloor + 1$ fino a n. Infatti, $2\lfloor n/2 \rfloor \leq n$, dunque il nodo $\lfloor n/2 \rfloor$ ha un figlio sinistro e dunque non è una foglia. Invece $2(\lfloor n/2 \rfloor + 1) > n$, dunque il nodo $\lfloor n/2 \rfloor + 1$ non ha figli, cioè è una foglia. Dunque in uno heap ci sono $\lfloor n/2 \rfloor$ nodi interni e $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$ foglie. Inoltre, è facile verificare che i nodi ad altezza h in uno heap sono al massimo $\lceil n/2^{h+1} \rceil$.

Esercizio 4.53 *Qual è il numero di nodi di uno heap di altezza h? Qual è l'altezza di uno heap di n nodi?*

Soluzione

Uno heap è un albero quasi completo. Sia A uno heap di altezza h e sia n il numero di nodi di A. Il minimo numero di nodi si ha quando A possiede una unica foglia a livello h. In tal caso A ha un nodo in più rispetto ad un albero completo di altezza $h-1$. Dato che un albero completo di altezza h possiede $2^{h+1}-1$ nodi, il minimo numero di nodi di uno heap di altezza h è

2^h. Il massimo numero di nodi si ha quando A è un albero completo. In tal caso, il numero di nodi è $2^{h+1} - 1$. Dunque $2^h \leq n \leq 2^{h+1} - 1$ e $n = \Theta(2^h)$. Passando al logaritmo otteniamo che l'altezza h di uno heap di n nodi è tale che $\log(n+1) - 1 \leq h \leq \log n$, cioè $h = \lfloor \log n \rfloor$. Dunque $h = \Theta(\log n)$.

Useremo spesso le seguenti procedure:

Algoritmo 89 Parent(i)

Parent(i)

1: **return** i *div* 2

Algoritmo 90 Left(i)

Left(i)

1: **return** $2i$

Algoritmo 91 Right(i)

Right(i)

1: **return** $2i + 1$

Inoltre, faremo uso della seguente procedura $Heapify(A, i)$. Tale procedura assume che i sottoalberi sinistro e destro del nodo i siano degli heap, ma $A[i]$ può essere più piccolo dei proprio figli, quindi violando la proprietà degli heap. La procedura fa scivolare giù l'elemento $A[i]$ lungo l'albero in modo che l'albero con radice i diventi uno heap.

La procedura $Heapify(A, i)$ scende lungo un cammino che parte dal nodo i dello heap A. Nel caso peggiore, tale cammino è il più lungo dell'albero, cioè ha lunghezza pari all'altezza dell'albero radicato in i. Dunque la complessità di $Heapify(A, i)$ è $\Theta(h)$, con h l'altezza dell'albero con radice i. L'esercizio 4.53 mostra che l'altezza di uno heap è logaritmica nel numero di nodi. Dunque la complessità di $Heapify(A, i)$ è $\Theta(\log n)$, ove n è il numero di nodi dell'albero radicato in i.

Algoritmo 92 Heapify(A,i)

Heapify(A,i)

```
l ← Left(i)
r ← Right(i)
if (l ≤ heapsize(A)) and (A[l] > A[i]) then
    largest ← l
else
    largest ← i
end if
if (r ≤ heapsize(A)) and (A[r] ≥ A[largest]) then
    largest ← r
end if
if largest ≠ i then
    Exchange(A, i, largest)
    Heapify(A, largest)
end if
```

Vediamo ora le operazioni sugli heap.

Algoritmo 93 HeapEmpty(A)

HeapEmpty(A)

```
if heapsize(A) = 0 then
    return TRUE
else
    return FALSE
end if
```

La complessità di *HeapEmpty* è $\Theta(1)$.

Algoritmo 94 HeapFull(A)

HeapFull(A)

```
if heapsize(A) = length(A) then
    return TRUE
else
    return FALSE
end if
```

La complessità di *HeapFull* è $\Theta(1)$.

Algoritmo 95 HeapBuild(A)

HeapBuild(A)

$heapsize(A) \leftarrow length(A)$
for $i \leftarrow length(A)$ div 2 **downto** 1 **do**
  $Heapify(A, i)$
end for

La procedura $Heapify$ all'interno della procedura $HeapBuild$ viene chiamata solo sui nodi interni, cioè sui nodi da $\lfloor n/2 \rfloor$ giù fino a 1. Infatti le foglie sono heap, in quanto non posseggono figli.

Vediamo ora la complessità di $HeapBuild(A, i)$. Il ciclo for viene ripetuto $\lfloor n/2 \rfloor$ volte. Il costo della chiamata $Heapify(A, i)$ all'interno del ciclo è al più pari all'altezza del nodo i. Dato uno heap di n nodi, l'altezza dello heap è $\lfloor \log n \rfloor$ (Esercizio 4.53). Quindi il costo di $HeapBuild(A, i)$ è $O(\lfloor n/2 \rfloor \lfloor \log n \rfloor) = O(n \log n)$. Questo limite superiore è corretto ma può essere migliorato nel seguente modo. Dato uno heap di n nodi, il numero di nodi di altezza h è al più $\lceil n/2^{h+1} \rceil$. Sommando le complessità di $Heapify$, livello per livello, a partire dalle foglie fino alla radice, abbiamo che la complessità di $HeapBuild$ risulta

$$\sum_{h=0}^{\lfloor \log n \rfloor} \lceil n/2^{h+1} \rceil O(h) = O(n \sum_{h=0}^{\lfloor \log n \rfloor} h/2^h).$$

Dato che $\sum_{h=0}^{\infty} h/2^h = 2$, abbiamo che

$$O(n \sum_{h=0}^{\lfloor \log n \rfloor} h/2^h) = O(2n) = O(n).$$

E' immediato vedere che la complessità di $HeapBuild$ è inferiormente limitata da $\lfloor n/2 \rfloor$, dunque è $\Omega(n)$. Possiamo dunque concludere che la complessità di $HeapBuild$ è $\Theta(n)$.

Algoritmo 96 HeapMax(A)

HeapMax(A)

if $HeapEmpty(A)$ **then**
  return NIL
end if
return $A[1]$

La complessità di $HeapMax$ è $\Theta(1)$.

Algoritmo 97 HeapInsert(A,x)

HeapInsert(A,x)

```
if HeapFull(A) then
    error OVERFLOW
end if
heapsize(A) ← heapsize(A) + 1
A[heapsize(A)] ← x
i ← heapsize(A)
while (i > 1) and (A[Parent(i)] < A[i]) do
    Exchange(A, i, Parent(i))
    i ← Parent(i)
end while
```

La procedura inserisce il nuovo elemento come foglia e poi lo fa scalare verso l'alto fino, nel caso peggiore, alla radice. Dunque la complessità di $HeapInsert(A, x)$ è $\Theta(h)$, con h l'altezza dello heap A, oppure $\Theta(\log n)$, ove n è il numero di nodi dello heap A.

Algoritmo 98 HeapDelete(A,i)

HeapDelete(A,i)

```
if HeapEmpty(A) then
    error UNDERFLOW
end if
old ← A[i]
new ← A[heapsize(A)]
A[i] ← new
heapsize(A) ← heapsize(A) − 1
if new < old then
    Heapify(A, i)
else
    while (i > 1) and (A[Parent(i)] < A[i]) do
        Exchange(A, i, Parent(i))
        i ← Parent(i)
    end while
end if
```

La procedura mette al posto del nodo i l'ultima foglia dello heap e poi, a seconda dei casi, la fa scendere giù verso le foglie o salire sù verso la radice. Dunque la complessità di $HeapDelete(A, i)$ è $\Theta(h)$, con h l'altezza dello heap A, oppure $\Theta(\log n)$, ove n è il numero di nodi dello heap A.

Algoritmo 99 HeapModifyKey(A,i,k)

HeapModifyKey(A,i,k)

$old \leftarrow A[i]$
$new \leftarrow k$
$A[i] \leftarrow new$
if $new < old$ **then**
  $Heapify(A, i)$
else
  while $(i > 1)$ **and** $(A[Parent(i)] < A[i])$ **do**
    $Exchange(A, i, Parent(i))$
    $i \leftarrow Parent(i)$
  end while
end if

La procedura modifica la chiave del nodo i e poi, a seconda dei casi, la fa scendere giù verso le foglie o salire sù verso la radice. Dunque la complessità di $HeapModifyKey(A, i, k)$ è $\Theta(h)$, con h l'altezza dello heap A, oppure $\Theta(\log n)$, ove n è il numero di nodi dello heap A.

Esercizio 4.54 *Si scrivano le seguenti procedure e si calcolino le relative complessità:*

- $HeapMin(A)$ *che ritorna l'indice del minimo di A;*
- $HeapSearch(A, i, k)$ *che ritorna l'indice dell'elemento con chiave k cercando nel sottoalbero di A radicato in i, oppure NIL se tale elemento non esiste.*

Soluzione

Algoritmo 100 HeapMin(A)

HeapMin(A)

return $ArrayMin(A, (heapsize(A)\ div\ 2) + 1, heapsize(A))$

La complessità di $HeapMin(A)$ è $\Theta(n/2) = \Theta(n)$, ove $n = heapsize(A)$.

Algoritmo 101 HeapSearch(A,i,k)

HeapSearch(A,i,k)

if $(i > heapsize(A))$ **or** $(A[i] < k)$ **then**
  return NIL
end if
if $A[i] = k$ **then**
  return i
end if
$l \leftarrow HeapSearch(A, Left(i), k)$
if $l \neq$ NIL **then**
  return l
else
  return $HeapSearch(A, Right(i), k)$
end if

La complessità della procedura $HeapSearch(A, i, k)$ è $\Theta(n)$, ove n è il numero di nodi dell'albero radicato in i. Infatti, l'equazione ricorsiva che ne descrive la complessità è $C(0) = 1$ e, per $n > 0$, $C(n) = C(a) + C(b) + 1$, con $a, b \geq 0$, e $a+b = n-1$. Per l'esercizio 2.10, punto 1, la soluzione è $C(n) = \Theta(n)$.

Esercizio 4.55 *Implementare le procedure $HeapInsert$ e $HeapDelete$ usando $ModifyKey$.*

Soluzione

Algoritmo 102 HeapInsert(A,x)

HeapInsert(A,x)

if $HeapFull(A)$ **then**
  error OVERFLOW
end if
$heapsize(A) \leftarrow heapsize(A) + 1$
$A[heapsize(A)] \leftarrow -\infty$
$HeapModifyKey(A, heapsize(A), x)$

Algoritmo 103 HeapDelete(A,i)

HeapDelete(A,i)

if $HeapEmpty(A)$ **then**
    error UNDERFLOW
end if
$new \leftarrow A[heapsize(A)]$
$heapsize(A) \leftarrow heapsize(A) - 1$
$HeapModifyKey(A, i, new)$

4.7.1 Code con priorità

Una **coda con priorità** (priority queue, in inglese) è una coda i cui oggetti hanno assegnata una priorità (che denoteremo con un numero). Gli oggetti vengono inseriti in coda con una priorità assegnata. L'elemento cancellato dalla coda non è il primo ad essere entrato, ma quello con priorità massima. Inoltre, la priorità di un oggetto può essere modificata in più o in meno. Ad esempio, un sistema operativo multitasking può utilizzare una coda con priorità per selezionare il prossimo processo da mandare in esecuzione. La priorità di un processo può essere aumentata per evitare fenomeni di *starvation*, cioè situazioni in cui un processo è pronto ad eseguire ma a causa della sua bassa priorità rispetto agli altri processi in coda non viene mai selezionato. Implementiamo una coda con priorità usando uno heap.

Algoritmo 104 PriorityEnqueue(A,x)

PriorityEnqueue(A,x)

$HeapInsert(A, x)$

Algoritmo 105 PriorityDequeue(A)

PriorityDequeue(A)

$max \leftarrow HeapMax(A)$
$HeapDelete(A, 1)$
return max

Algoritmo 106 ModifyPriority(A,i,k)

ModifyPriority(A,i,k)

1: $HeapModifyKey(A, i, k)$

Queste operazioni hanno complessità logaritmica nel numero di oggetti presenti in coda.

5 Algoritmi di ordinamento e di selezione

In questa sezione verranno studiati algoritmi che risolvono il problema dell'ordinamento di un vettore (Esempio 2.1) e un algoritmo che risolve il problema della selezione. Un algoritmo di **ordinamento per confronto** utilizza il confronto tra elementi del vettore per determinare l'ordine relativo degli elementi del vettore. Dati due elementi a_i e a_j del vettore, un confronto è un test del tipo $a_i \leq a_j$. Un algoritmo di ordinamento di un vettore A viene detto **in-place** se gli elementi di A vengono riarrangiati all'interno del vettore A, e, in ogni istante, solo un numero costante di elementi vengono tenuti fuori da A.

Teorema 5.1 *La complessità pessima del problema dell'ordinamento, qualora ci riduciamo a considerare solo algoritmi per confronto, è* $\Omega(n \log n)$.

Dimostrazione
Un *albero di decisione* è un albero pieno che rappresenta i confronti effettuati da un generico algoritmo di ordinamento per confronto su una sequenza di ingresso di una certa dimensione. Sia $\langle a_1, \ldots, a_n \rangle$ una sequenza di ingresso di dimensione n. Ogni nodo interno dell'albero di decisione è etichettato con un confronto $a_i \leq a_j$, ove $1 \leq i \neq j \leq n$; il sottoalbero di sinistra del nodo etichettato con $a_i \leq a_j$ è l'albero di decisione per un sequenza in cui $a_i \leq a_j$, mentre il sottoalbero di destra del nodo etichettato con $a_i \leq a_j$ è l'albero di decisione per un sequenza in cui $a_i > a_j$. Le foglie sono etichettate con permutazioni $\langle a'_1, \ldots, a'_n \rangle$ della sequenza di ingresso. Un cammino dalla radice ad una foglia rappresenta dunque una possibile esecuzione di un algoritmo di ordinamento per confronto. Un algoritmo di ordinamento corretto deve essere in grado di generare ogni possibile permutazione dell'input. Dunque tutte le permutazioni della sequenza iniziale devono comparire come foglia. La complessità pessima di un qualsiasi algoritmo di ordinamento per confronto è equivalente al numero di confronti effettuati per risolvere il problema. Tale numero è la lunghezza del cammino più lungo che parte dalla radice e arriva ad una foglia nell'albero di decisione, cioè è l'altezza dell'albero di decisione. Un albero pieno di altezza h ha al più 2^h foglie. Dato che un albero di decisione è un albero pieno con $n!$ foglie, abbiamo che $n! \leq 2^h$ e dunque $h \geq \log(n!) = \Omega(n \log n)$ (Esercizio 2.7). Dunque la complessità pessima di un algoritmo di ordinamento per confronto è $\Omega(n \log n)$.

Si noti che il cammino minimo sull'albero di decisione è lungo $n - 1$ e corrisponde a $n - 1$ confronti. Dunque la **complessità minima** del problema dell'ordinamento è lineare $\Omega(n)$. Si dimostra che la **complessità media** del problema dell'ordinamento è $\Omega(n \log n)$.

5.1 InsertionSort

InsertionSort è un algoritmo di ordinamento iterativo, per confronto, e in-place. Questo algoritmo è molto semplice anche se non efficience e quindi viene spesso impiegato per ordinare vettori di lunghezza limitata.

Algoritmo 107 InsertionSort(A)

InsertionSort(A)

for $j \leftarrow 2$ **to** $length(A)$ **do**
    $key \leftarrow A[j]$
    $i \leftarrow j - 1$
    while $(i > 0)$ **and** $(key < A[i])$ **do**
        $A[i+1] \leftarrow A[i]$
        $i \leftarrow i - 1$
    end while
    $A[i+1] \leftarrow key$
end for

La procedura ordina il vettore A in senso crescente. Infatti, ogni iterazione del ciclo **for** fa scivolare l'elemento $A[j]$ nella giusta posizione all'interno del sottovettore $A[1, \ldots, j]$. Ne deriva che al termine di ogni iterazione del ciclo **for**, il sottovettore $A[1, \ldots, j]$ contiene gli elementi che originariamente si trovavano in $A[1, \ldots, j]$ ma ordinati in senso crescente. Alla fine dell'ultima iterazione $j = length(A)$, e dunque il vettore $A[1, \ldots, length(A)]$ è ordinato in senso crescente.

Sia $n = length(A)$. Il caso ottimo si ha quando A è inizialmente ordinato in senso crescente. In tal caso, il ciclo **while** non viene mai eseguito, e dunque ogni iterazione del ciclo **for** ha complessità costante. Dato che ci sono $n-1$ iterazioni del ciclo **for**, la complessità ottima risulta $(n-1)\Theta(1) = \Theta(n-1) = \Theta(n)$.

Il caso pessimo si ha quando A è inizialmente ordinato in senso decrescente. In tal caso, il ciclo **while** viene sempre eseguito fino all'ultimo, cioè da $i = j - 1$ fino a $i = 0$. Dunque ogni iterazione del ciclo **for** ha complessità $\Theta(j)$. Dato che il ciclo **for** itera da $j = 2$ fino a $j = n$, la complessità pessima risulta $\sum_{j=2}^{n} \Theta(j) = \Theta(\sum_{j=2}^{n} j) = \Theta(n(n+1)/2 - 1) = \Theta(n^2)$.

5.2 HeapSort

HeapSort è un algoritmo di ordinamento iterativo, per confronto, e in-place. *HeapSort* usa la struttura di dati heap (Sezione 4.7).

La complessità pessima di *HeapSort* è $O(n) + (n-1)O(\log n) = O(n \log n)$, ove $n = length(A)$. Se il vettore A è inizialmente ordinato (in senso crescente o decrescente), la complessità di *HeapSort*(A) rimane $n \log n$. Dunque la complessità pessima di *HeapSort* è $\Theta(n \log n)$. E' possibile mostrare che la complessità ottima di *HeapSort* rimane $\Theta(n \log n)$. Di conseguenza la complessità media di *HeapSort* risulta $\Theta(n \log n)$. Dunque *HeapSort* non distingue tra caso ottimo, medio e pessimo.

Algoritmo 108 HeapSort(A)

HeapSort(A)

```
HeapBuild(A)
for i ← length(A) downto 2 do
   Exchange(A, 1, i)
   heapsize(A) ← heapsize(A) − 1
   Heapify(A, 1)
end for
```

5.3 QuickSort

Presentiamo ora *QuickSort*, un algoritmo di ordinamento ricorsivo, per confronto, e in-place. *QuickSort* usa una procedura ausiliaria $Partition(A, p, r)$ che partiziona il vettore A in due sottovettori $A[p, \ldots, q - 1]$ e $A[q + 1, \ldots, r]$, per qualche $p \leq q \leq r$, tali che, per ogni $p \leq i \leq q - 1$, $A[i] \leq A[q]$, e, per ogni $q + 1 \leq i \leq r$, $A[q] \leq A[i]$. L'elemento $A[q]$ viene detto *pivot* e la procedura ritorna q.

Algoritmo 109 Partition(A,p,r)

Partition(A,p,r)

```
x ← A[r]
i ← p − 1
for j ← p to r − 1 do
   if A[j] ≤ x then
      i ← i + 1
      Exchange(A, i, j)
   end if
end for
Exchange(A, i + 1, r)
return i + 1
```

Si noti che se A è ordinato (in qualche senso), la procedura $Partition(A, p, r)$ ritorna uno degli estremi (p o r). La complessità di $Partition(A, p, r)$ è $\Theta(n)$, ove $n = r - p + 1$ è la dimensione del sottovettore $A[p, \ldots, r]$.

Algoritmo 110 QuickSort(A,p,r)

QuickSort(A,p,r)

```
if p < r then
   q ← Partition(A, p, r)
   QuickSort(A, p, q − 1)
   QuickSort(A, q + 1, r)
end if
```

Nel caso pessimo la procedura $Partition(A, p, r)$ ritorna un estremo (p o r), cioè il vettore A viene suddiviso in un sottovettore di dimensione $n-1$ e uno di dimensione 0, ove $n = r-p+1$. Dunque la complessità pessima è espressa dalla seguente equazione ricorsiva: $C(0) = \Theta(1)$ e $C(n) = C(n-1) + C(0) + n = C(n-1) + (n+1)$, per $n > 0$. Dunque $C(n) = \Theta(n^2)$ (Esercizio 2.10, punto 3). Quindi la complessità pessima di $QuickSort$ è quadratica.

Nel caso ottimo la procedura $Partition(A, p, r)$ ritorna un indice equidistante da p e r, cioé il vettore A viene suddiviso in un sottovettore di dimensione $\lfloor n/2 \rfloor$ e uno di dimensione $\lceil n/2 \rceil - 1$, ove $n = r - p + 1$. Dunque la complessità ottima è espressa dalla seguente equazione ricorsiva: $C(0) = \Theta(1)$ e $C(n) = 2C(n/2) + \Theta(n)$, per $n > 0$, la cui soluzione è $C(n) = \Theta(n \log n)$ (Esercizio 2.9, punto 3).

Infine, è possibile mostrare che, in media, $Partition(A, p, r)$ suddivide in maniera bilanciata il vettore A, e dunque la complessità media risulta essere uguale alla complessità ottima, cioé $\Theta(n \log n)$.

5.4 MergeSort

$MergeSort$ è un algoritmo di ordinamento ricorsivo, per confronto, e non in-place. $MergeSort$ usa una procedura ausiliaria $Merge(A, p, q, r)$ che presuppone che i sottovettori $A[p, \ldots, q]$ e $A[q + 1, \ldots, r]$ siamo ordinati (in senso crescente). Il compito di $Merge(A, p, q, r)$ è quello di odinare il vettore $A[p, \ldots, r]$.

Algoritmo 111 Merge(A, p, q, r)

Merge(A, p, q, r)

$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
for $i \leftarrow 1$ **to** n_1 **do**
  $L[i] \leftarrow A[p + i - 1]$
end for
for $j \leftarrow 1$ **to** n_2 **do**
  $R[j] \leftarrow A[q + j]$
end for
$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
for $k \leftarrow p$ **to** r **do**
  if $L[i] \leq R[j]$ **then**
    $A[k] \leftarrow L[i]$
    $i \leftarrow i + 1$
  else
    $A[k] \leftarrow R[j]$
    $j \leftarrow j + 1$
  end if
end for

La complessità di $Merge(A, p, q, r)$ è $\Theta(n)$, over $n = r - p + 1$ è la dimensione del sottovettore $A[p, \ldots, r]$.

Algoritmo 112 MergeSort(A, p, r)

MergeSort(A, p, r)

if $p < r$ **then**
  $q \leftarrow (p + r)\ div\ 2$
  $MergeSort(A, p, q)$
  $MergeSort(A, q + 1, r)$
  $Merge(A, p, q, r)$
end if

La complessità pessima di $MergeSort(A, p, r)$ è espressa dalla seguente equazione ricorsiva: $C(1) = \Theta(1)$ e $C(n) = 2C(n/2) + \Theta(n)$, per $n > 1$. La soluzione è $C(n) = \Theta(n \log n)$ (Esercizio 2.9, punto 3). E' immediato vedere che $MergeSort$ non distingue tra caso pessimo, medio e ottimo, dunque la complessità ottima e la complessità media di $MergeSort$ risultano $\Theta(n \log n)$.

Esercizio 5.1 *La Signora Edda è giunta al seguente paradosso. Ella sostiene di aver appena scoperto il seguente algoritmo di ordinamento per confronto con complessità pessima lineare: dato un albero binario di ricerca, si visiti tale albero* *in ordine intermedio (sinistra-radice-destra), salvando le chiavi dei nodi visitati. Al termine della visita, le chiavi saranno state salvate in ordine crescente. Tale algoritmo ha complessità pessima lineare $\Theta(n)$, ove n è il numero di nodi dell'albero. D'altronte, ella è a conoscenza dell'esistenza di un teorema che afferma che il problema dell'ordinamento per confronto ha complessità pessima pari a $\Theta(n \log n)$, ove n è il numero di elementi da ordinare. Dunque, non possono esistere algoritmi di ordinamento per confronto con complessità pessima lineare. Dove sbaglia la Signora Edda?*

Soluzione

La Signora Edda sbaglia in quanto l'algoritmo che lei sostiene di aver trovato risolve solo parzialmente il problema dell'ordinamento. Infatti, il problema dell'ordinamento consiste nel trovare una permutazione ordinata di una sequenza fornita in ingresso. L'input del problema dell'ordinamento è dunque una sequenza, e non un albero binario di ricerca, in cui le chiavi hanno una certa organizzazione (secondo la proprietà degli alberi binari di ricerca). Dunque l'algoritmo è parziale in quanto manca un passo iniziale che costruisce un albero binario di ricerca a partire da una sequenza. Questo passo ha complessità pessima pari a $\Theta(n^2)$, ove n è la lunghezza della sequenza, come mostrato nell'Esercizio 4.45.

Esercizio 5.2 *Si dica se il seguente algoritmo risolve il problema dell'ordinamento e si calcoli la complessità computazionale ottima e pessima.*

Algoritmo 113 BubbleSort(A)

BubbleSort(A)

for $i \leftarrow 1$ **to** $length(A)$ **do**
    for $j \leftarrow length(A)$ **downto** $i+1$ **do**
        if $A[j-1] > A[j]$ **then**
            $Exchange(A, j, j-1)$
        end if
    end for
end for

Soluzione

L'algoritmo risolve il problema dell'ordinamento: dopo un passo del ciclo più esterno, il minimo di $A[1, \ldots, length(A)]$ viene posto in prima posizione, dopo due passi, il minimo di $A[2, \ldots, length(A)]$ viene posto in seconda posizione, e così via. In generale, dopo k passi, il vettore $A[1, \ldots, k]$ contiene i primi k elementi di A ordinati in senso crescente. Dunque, dopo $k = length(A)$ passi, cioè al termine della procedura, il vettore $A[1, \ldots, length(A)]$ contiene gli elementi originariamente contenuti in A ordinati in senso crescente.

Non esiste differenza significativa tra complessità ottima e pessima, in quanto la procedura *Exchange* ha complessità costante. Sia $n = length(A)$. Il ciclo più esterno itera da 1 a n. Alla k-esima iterazione del ciclo più esterno, il ciclo interno compie $n-k$ iterazioni di complessità costante. Dunque la complessità ottima e pessima risulta $\sum_{i=1}^{n-1} i = (n-1)n/2 = \Theta(n^2)$.

5.5 Ordinamento in tempo lineare

Vediamo di seguito un algoritmo di ordinamento che non si basa sui confronti. L'algoritmo $CountingSort(A, k)$ assume che gli elementi di A siano numeri interi compresi nell'intervallo $[0, k]$.

Algoritmo 114 CountingSort(A, k)

CountingSort(A, k)

1: // Uso due vettori $B[1, \dots, length(A)]$ e $C[0, \dots, k]$
2: **for** $i \leftarrow 0$ **to** k **do**
3:     $C[i] \leftarrow 0$
4: **end for**
5: **for** $i \leftarrow 1$ **to** $length(A)$ **do**
6:     $C[A[i]] \leftarrow C[A[i]] + 1$
7: **end for**
8: **for** $i \leftarrow 1$ **to** k **do**
9:     $C[i] \leftarrow C[i-1] + C[i]$
10: **end for**
11: **for** $i \leftarrow length(A)$ **downto** 1 **do**
12:     $B[C[A[i]]] \leftarrow A[i]$
13:     $C[A[i]] \leftarrow C[A[i]] - 1$
14: **end for**
15: **for** $i \leftarrow 1$ **to** $length(A)$ **do**
16:     $A[i] \leftarrow B[i]$
17: **end for**

Si noti che dopo il secondo ciclo for, cioè alla linea 7, l'elemento $C[i]$ contiene il numero di elementi di A che sono uguali a i. Dopo il terzo ciclo for, cioè alla linea 10, l'elemento $C[i]$ contiene il numero di elementi di A che sono minori o uguali a i. Dunque $C[A[i]]$ è la posizione di $A[i]$ nel vettore ordinato. Se $n = length(A)$, la complessità di $CountingSort(A, k)$ è $\Theta(n+k)$. Dunque qualora $k = O(n)$, $CountingSort(A, k)$ ha complessità lineare $\Theta(n)$.

5.6 Algoritmi di selezione

Sia S una sequenza di n numeri distinti. Dato $1 \leq i \leq n$ e un elemento x di S, diremo che x è l'elemento di **ordine** i di S se esistono $i-1$ elementi in S più piccoli di x. In particolare, l'elemento di ordine 1 è detto minimo di S, l'elemento di ordine n è detto massimo di S, e l'elemento di ordine $\lfloor (n+1)/2 \rfloor$ è detto **mediana** di S.

Esempio 5.1 (*Problema della selezione*)

Il problema della selezione è il seguente: dati in ingresso una sequenza S di n numeri distinti e un indice $1 \leq i \leq n$, fornire in uscita l'elemento di ordine i di S.

Il problema della selezione può essere risolto con complessità $\Theta(n \log n)$ ordinando gli elementi della sequenza e ritornando l'i-esimo elemento della sequenza ordinata. Questa soluzione è inefficiente, in quanto la soluzione del problema della selezione non richiede l'ordinamento dell'intera sequenza, ma, come vedremo, solo un opportuno partizionamento della stessa.

Vediamo un algoritmo che risolve il problema della selezione in tempo lineare nella dimensione dell'input. Sia S una sequenza di n numeri distinti e $1 \leq i \leq n$. L'algoritmo *Select* consiste dei seguenti cinque passi.

1. Dividi gli n elementi di S in $\lfloor n/5 \rfloor$ gruppi di 5 elementi ciascuno e un eventuale gruppo di n *mod* 5 elementi;
2. trova la mediana di ogni singolo gruppo ordinando gli elementi di ciascun gruppi usando $MergeSort$;
3. usa *Select* ricorsivamente per trovare la mediana x delle mediane dei singoli gruppi;
4. partiziona la sequenza S attorno alla mediana delle mediane x usando (una versione modificata di) *Partition*. Sia k la posizione di x dopo la partizione;
5. se $i = k$, allora ritorna x. Altrimenti, usa *Select* ricorsivamente per trovare l'elemento di ordine i sulla metà di sinistra della partizione, qualora $i < k$, oppure per trovare l'elemento di ordine $i - k$ sulla metà destra della partizione, qualora $i > k$.

Esercizio 5.3 *Si usi la procedura Select per implementare una versione di QuickSort con complessità pessima pari a $\Theta(n \log n)$.*

Soluzione

L'idea è quella di partizionare il vettore attorno alla mediana, invece che attorno ad un generico pivot. Scriviamo innanzitutto la seguente procedura che ci calcola la mediana:

Algoritmo 115 Median(A,p,r)

Median(A,p,r)

1: $n \leftarrow r - p + 1$
2: **return** *Select*$(A, p, r, (n + 1)$ *div* $2)$

Usando $Median$ implementiamo una versione accelerata di $QuickSort$:

Algoritmo 116 QuickSort(A,p,r)

QuickSort(A,p,r)

if $p < r$ **then**
  $x \leftarrow Median(A, p, r)$
  $i \leftarrow ArraySearch(A, p, r, x)$
  $Exchange(A, i, r)$
  $q \leftarrow Partition(A, p, r)$
  $QuickSort(A, p, q - 1)$
  $QuickSort(A, q + 1, r)$
end if

L'equazione di complessità nel caso pessimo risulta $C(n) = 2C(n/2) + \Theta(n)$, e dunque la complessità pessima vale $C(n) = \Theta(n \log n)$.

Di seguito implementiamo $Select(A, p, r, i)$ che trova l'elemento di ordine i cercando in $A[p, \ldots, r]$. E' possibile mostrare che la complessità di $Select(A, p, r, i)$ è $\Theta(n)$, con $n = r - p + 1$.

Algoritmo 117 Select(A,p,r,i)

Select(A,p,r,i)

```
// Uso un vettore B[1, ..., ⌈length(A)/5⌉]
// p ≤ r, 1 ≤ i ≤ r − p + 1
// Inizio Parte I
if p = r then
   return A[p]
end if
n ← r − p + 1
for j ← 1 to n div 5 do
   MergeSort(A, p + 5(j − 1), p + 5j − 1)
end for
if (n mod 5) ≠ 0 then
   MergeSort(A, p + 5j, r)
end if
// Inizio Parte II
for j ← 1 to n div 5 do
   B[j] ← A[p + (j − 1)5 + 2]
end for
if (n mod 5) ≠ 0 then
   q ← p + 5j
   B[j + 1] ← A[q + (r − q) div 2]
end if
// Inizio Parte III
m ← ⌈n/5⌉
x ← Select(B, 1, m, (m + 1) div 2)
// Inizio Parte IV
j ← ArraySearch(A, p, r, x)
Exchange(A, j, r)
k ← Partition(A, p, r)
// Inizio Parte V
if i = k then
   return x
else
   if i < k then
      Select(A, p, k − 1, i)
   else
      Select(A, k + 1, r, i − k)
   end if
end if
```