Appunti di “Algoritmi e Strutture Dati”

SiD

1. Concetti chiave ... 3
1.1. In-place ... 3
1.2. Tight ... 3
1.3. Stabilità ... 3
1.4. Algoritmo asintoticamente ottimale ... 3
1.5. Arrotondamenti ... 3
1.6. Complessità e limiti superiori ... 3
2. Equazioni ricorsive ... 4
2.1. Sostituzione ... 4
2.2. Iterazione ... 4
2.3. Albero delle chiamate ricorsive ... 4
3. Algoritmi di ordinamento per confronto ... 4
3.1. Ordinamento tramite INSERTIONSORT di un vettore 4
3.2. Ordinamento tramite SELECTIONSORT di un vettore 4
3.3. Ordinamento tramite BUBBLESORT di un vettore ... 5
3.4. Ordinamento tramite MERGESORT di un vettore ... 5
3.5. Ordinamento tramite HEAPSORT di un vettore tramite un heap ... 6
3.6. Ordinamento tramite QUICKSORT di un vettore ... 7
3.7. Riepilogo delle complessità ... 7
4. Heap ... 8
4.1. HEAPIFY ... 8
4.2. BUILDHEAP ... 9
4.3. PARTITION ... 9
4.4. HEAPMAXIMUM ... 10
4.5. HEAPEXTRACTMAX ... 10
4.6. HEAPINCREASEKEY ... 10
4.7. HEAPINSERT ... 10
4.8. Riepilogo delle complessità ... 10
5. Algoritmi di ordinamento in tempo lineare ... 11
5.1. Ordinamento tramite COUNTINGSORT ... 11
5.2. Ordinamento tramite RADIXSORT ... 11
5.3. Ordinamento tramite BUCKETSORT ... 12
6. Algoritmi e strutture dati di supporto agli algoritmi di ordinamento in tempo lineare ... 12
6.1. RANDOMIZEDSELECT ... 12
6.2. SELECT ... 13
6.3. Operazioni su stack LIFO ... 13
6.4. Operazioni su code FIFO ... 13
6.5. Operazioni sulle liste concatenate ... 13
7. Hashing ... 14
7.1. Tabella di indirizzamento diretto ... 14
7.2. Hash con chaining ... 14
Funzioni di hash ... 14
7.3. Hash con open addressing ... 14
Funzioni di hash ... 14
7.4. Hashing doppio ... 14
8. Alberi Binari di Ricerca (BST) ... 15
8.1. Proprietà ... 15
8.2. Visite ... 15
8.3. Ricerca ... 15
8.4. Massimo e minimo ... 16
8.5. Successore e predecessore ... 16
8.6. Inserimento e cancellazione ... 16
Inserimento ... 16
Cancellazione ... 17
8.7. Riepilogo delle complessità ... 18
9. Alberi rosso-neri (Red-Black tree) ... 18
9.1. Proprietà ... 18
9.2. Rotazioni ... 19
9.3. Inserimento ... 20
RB-INSERT-FIXUP ... 20
Analisi ... 20
9.4. Cancellazione ... 21
RB-DELETE-FIXUP ... 22
9.5. Un’applicazione: Join ... 24
10. Balanced-Tree (B-Tree) ... 24
10.1. Ricerca ... 24
10.2. Creazione ... 25
10.3. Split di nodi pieni ... 25
10.4. Inserimento ... 25
10.5. Cancellazione ... 26
11. Insiemi disgiunti – Union Find ... 26

11.1. Make 26
11.2. Union 26
11.3. Find 26
12. Algoritmi sui grafi 26
12.1. Breadth-first search 26
12.2. Breadth-first tree 27
12.3. Depth-first search 27
12.4. Classificazione degli archi 27
12.5. Topological sort 27
12.6. Componenti fortemente connesse 27
13. Minimum spanning tree (MST) 28
13.1. Algortimo generico 28
13.2. Algoritmo di Kruskal 28
13.3. Algoritmo di Prim 29
14. Single source shortest path problem 29
14.1. Algoritmo di Dijkstra 29

Nota: sono stati sottolineati (in orizzontale o in verticale) i procedimenti e/o gli algoritmi non banali .

Il libro di testo a cui questi appunti fanno riferimeto è “Introduzione agli algoritmi e strutture dati” – McGraw-Hill (Cormen, Leiserson, Rivest, Stein) – 2° edizione.

Gli algoritmi sono stati clippati dalle dispense ad uno zoom di 150% da Acrobat 8 e poi rimpicciolite al 50% in MS Word 2007.

1. Concetti chiave

1.1. In-place

Un algoritmo si dice *in-place* quando in ogni istante c'è al più un numero costante di elementi memorizzati al di fuori dello spazio assegnato di input.

1.2. Tight

Un codice si dice *tight* quando contiene tutte e sole operazioni necessarie.

1.3. Stabilità

Un algoritmo di ordinamento si dice *stabile* quando gli elementi con chiave uguale non vengono spostati di posizione. Ad esempio, dato

$$A = \begin{Bmatrix} a = 3 \\ b = 2 \\ c = 2 \end{Bmatrix}$$

se eseguo un algoritmo di ordinamento non-stabile sul vettore A, potrei ottenere

$$A' = \begin{Bmatrix} c = 2 \\ b = 2 \\ a = 3 \end{Bmatrix}$$

al contrario, se eseguissi un algoritmo stabile sul vettore A, sarei sicuro di ottenere

$$A'' = \begin{Bmatrix} b = 2 \\ c = 2 \\ a = 3 \end{Bmatrix}$$

1.4. Algoritmo asintoticamente ottimale

...spiegare...

1.5. Arrotondamenti

Con $\lfloor x \rfloor$ si intende il valore arrotondato per difetto di x.

Con $\lceil x \rceil$ si intende il valore arrotondato per eccesso di x.

1.6. Complessità e limiti superiori

Con O si intende il limite asintotico superiore di una funzione.

Con Θ si intende il limite asintotico stretto di una funzione.

Con Ω si intende il limite asintotico inferiore di una funzione.

Con $T(n)$ si intende la funzione che rappresenta la complessità di un algoritmo per un input di dimensione n.

Per approfondimenti vedi §3 del libro di testo.

Ordinamento

2. Equazioni ricorsive

2.1. Sostituzione

2.2. Iterazione

2.3. Albero delle chiamate ricorsive

3. Algoritmi di ordinamento per confronto

Gli algoritmi di ordinamento per confronto hanno complessità

$$\Omega(n \log n)$$

3.1. Ordinamento tramite InsertionSort di un vettore

Ordina come in una mano di carte ed è buono per piccoli input.

L'algoritmo è *in-place*.

L'algoritmo è *stabile*.

```
Insertion Sort(A)
1: for j ← 2 to length(A) do
2:    key ← A[j]
3:    i ← j − 1
4:    while (i > 0) and (key < A[i]) do
5:       A[i + 1] ← A[i]
6:       i ← i − 1
7:    end while
8:    A[i + 1] ← key
9: end for
```

Algoritmo	Complessità peggiore	Complessità media	Complessità migliore
InsertionSort	$\Theta(n^2)$	$O(n^2)$	$O(n)$

3.2. Ordinamento tramite SelectionSort di un vettore

L'algoritmo è *in-place*.

L'algoritmo è *stabile* se si usa $\leq$ invece di $<$.

```
SelectionSort(A)
 1: curlength ← length(A)
 2: for j ← 1 to lenght(A) do
 3:    curmin ← A[1]
 4:    for i ← 2 to curlength do
 5:       if curmin < A[i] then
 6:          A[i − 1] ← A[i]
 7:       else
 8:          A[i − 1] ← curmin
 9:          curmin ← A[i]
10:       end if
11:    end for
12:    B[j] ← curmin
13:    curlength ← curlength - 1
14: end for
```

Algoritmo	Complessità peggiore	Complessità media	Complessità migliore
SELECTIONSORT	$\Theta(n^2)$	=	=

3.3.Ordinamento tramite BUBBLESORT di un vettore

```
Bubble Sort(A)
 1: for j ← length(A) downto 2 do
 2:    for i ← 2 to j do
 3:       if A[i − 1] < A[i] then
 4:          swap(A[i − 1], A[i])
 5:       end if
 6:    end for
 7: end for
```

3.4.Ordinamento tramite MERGESORT di un vettore

L’algoritmo è asintoticamente ottimale.

L’algoritmo fa uso della procedura ausiliare MERGE.

L’algoritmo **non** è *in-place*.

L’algoritmo è *stabile*.

```
MergeSort(A, p, r)
 1: if p < r then
 2:    q ← ⌊(p+r)/2⌋
 3:    MergeSort(A, p, q)
 4:    MergeSort(A, q + 1, r)
 5:    Merge(A, p, q, r)
 6: end if
```

```
Merge(U, x, y, z)
1: k ← i ← x
2: j ← y + 1
3: while (i ≤ y) ∨ (j ≤ z) do
4:   if i > y then
5:     V[k] ← U[j]
6:     j ← j + 1
7:   else
8:     if j > z then
9:       V[k] ← U[i]
10:      i ← i + 1
11:    else
12:      if U[i] < U[j] then
13:        V[k] ← U[i]
14:        i ← i + 1
15:      else
16:        V[k] ← U[j]
17:        j ← j + 1
18:      end if
19:    end if
20:  end if
21:  k ← k + 1
22: end while
23: U[x, ..., z] ← V[x, ..., z]
```

Algoritmo	Complessità peggiore	Complessità media	Complessità migliore
MERGESORT	$\Theta(n \log n)$	=	=

3.5. Ordinamento tramite HEAPSORT di un vettore tramite un heap

L'algoritmo è asintoticamente ottimale.

L'algoritmo combina le qualità di MergeSort con InsertionSort.

L'algoritmo sfrutta la struttura dati heap (§4) convertendo prima l'array in un heap ($O(n)$) e poi eseguendo $n-1$ volte HEAPIFY ($(n-1) \cdot O(\log n)$), ottenendo così una complessità $T(n) = O(n \log n)$.

Questo algoritmo è eccellente, ma una buona implementazione di QUICKSORT riesce a batterlo.

L'algoritmo è *in-place*.

L'algoritmo **non** è *stabile*.

```
HeapSort(A)
1: BuildHeap(A)
2: for i ← length(A) downto 2 do
3:   swap(A[1], A[i])
4:   heapsize(A) ← heapsize(A) − 1
5:   Heapify(A, 1)
6: end for
```

Algoritmo	Complessità peggiore	Complessità media	Complessità migliore
HEAPSORT	$O(n \log n)$	=	=

3.6. Ordinamento tramite QuickSort di un vettore

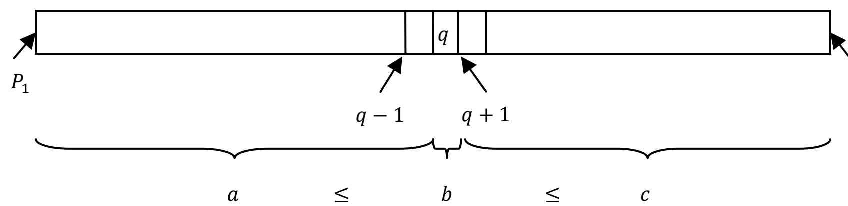
L'algoritmo è buono per input di grandi dimensioni.

Si nota che l'agoritmo è buono anche se viene usata memoria virtuale.

Spesso è l'algoritmo migliore a livello pratico.

L'algoritmo procede in questo modo:

1) Partiziona l'array in due parti utilizzando lasciando un elemento nel mezzo utilizzando Partition (§4.3) e trovando quindi q:



2) Procede ricorsivamente su a e c.

Nel caso peggiore lo sbilanciamento fra la dimensione di a e c è massimo: il costo di Partition è $\Theta(n)$ e QuickSort ha un costo $T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n) =$ [i] $\Theta(n^2)$ ovvero come InsertionSort quando il vettore è già ordinato.

Nel caso migliore lo sbilanciamento fra la dimensione di a e c è minimo: Partition produce segmenti con rapporto di grandezza sempre uguale e QuickSort ha costo $T(n) \le 2T\left(\frac{n}{2}\right) + \Theta(n)$, quindi $T(n) = O(n \log n)$.

L'algoritmo è *in-place*.

L'algoritmo **non** è *stabile*.

```
QuickSort((A, p, r)
1: if p < r then
2:     q ← Partition(A, p, r)
3:     QuickSort(A, p, q)
4:     QuickSort(A, q + 1, r)
5: end if
```

Algoritmo	Complessità peggiore	Complessità media	Complessità migliore
QuickSort	$\Theta(n^2)$	$O(n \log n)$	$\Theta(n \log n)$

3.7. Riepilogo delle complessità

Algoritmo	Complessità peggiore	Complessità media	Complessità migliore

[i] Serie aritmetica, si risolve intuitivamente.

InsertionSort	$\Theta(n^2)$	$O(n^2)$	$O(n)$
SelectionSort	$\Theta(n^2)$	=	=
MergeSort	$\Theta(n \log n)$	=	=
HeapSort	$\Theta(n \log n)$	=	=
QuickSort	$O(n^2)$	?	$O(n \log n)$

4. Heap

Gli heap si suddividono in max-heap (con il massimo nella radice) e min-heap (con il minimo nella radice). Di seguito verranno trattati i max-heap.

Un heap è un albero binario[i] quasi completo[ii] in cui valgono le seguenti proprietà:

$$key(x) \geq key(left(x))$$
$$key(x) \geq key(right(x))$$

Si deduce che il massimo si trova nella radice dell'heap.

Si deduce anche la seguente proprietà:

$$A[parent(i)] \geq A[i]$$

Notiamo che un albero binario completo, come anche un heap con n nodi, ha $\left\lceil \frac{n}{2} \right\rceil$ foglie.

Notiamo che l'altezza di un albero binario completo, come quella di un heap, è $h = \Theta(\log n)$.
Notiamo che le operazioni su un heap hanno al massimo complessità $O(\alpha \cdot h)$, quindi sono $O(\log n)$.

Notiamo anche che le operazioni sui livelli "più bassi" (ovvero sui livelli più vicini alle foglie) sono le più costose.
Notiamo quindi che gli alberi "più pieni" risultano più performanti e che gli heap sono gli alberi "più pieni" (o "più bassi") possibili.

4.1. Heapify

Questo algoritmo prende in input un albero i cui figli $left$ e $right$ della radice sono heap e rende l'albero stesso un heap.

L'algoritmo procede in questo modo:

1) Viene verificato che la proprietà dell'heap non sia soddisfatta nella radice, se lo è si termina.
2) Viene scambiata la radice con il figlio più grande e si torna al passo 1 prendendo in considerazione il nodo che era radice.

[i] Ovvero ogni nodo dell'albero ha 2 figli.
[ii] Ovvero non è necessario che l'ultimo livello sia completo.

Notiamo che l'algoritmo attraversa tutta l'altezza dell'albero. Ha complessità $O(h)$ in quanto l'albero può essere quasi-completo. Se fosse completo avrebbe complessità $\Theta(h)$.

```
Heapify(A, i)
 1: l ← left(i)
 2: r ← right(i)
 3: if (l ≤ heapsize(A)) ∧ (A[l] > A[i]) then
 4:    largest ← l
 5: else
 6:    largest ← i
 7: end if
 8: if (r ≤ heapsize(A)) ∧ (A[r] ≥ A[largest]) then
 9:    largest ← r
10: end if
11: if largest ≠ i then
12:    swap(A[i], A[largest])
13:    Heapify(A, largest)
14: end if
```

Algoritmo	Complessità peggiore	Complessità media	Complessità migliore
HEAPIFY	$O(h) = O(\log n)$	?	?

4.2. BUILDHEAP

Viene semplicemente eseguito HEAPIFY su ogni elemento dell'array passato da $\left\lfloor \frac{length(A)}{2} \right\rfloor$ a 1, ovvero su tutti i padri delle foglie e poi, salendo, fino alla radice.

La complessità di questa routine è $T(n) = \text{costo di una chiamata} \cdot \text{numero chiamate} = O(\log n) \cdot O(n) = O(n \log n)$.

```
BuildHeap(A)
 1: heapsize(A) ← length(A)
 2: for i ← ⌊length(A)/2⌋ downto 1 do
 3:    Heapify(A, i)
 4: end for
```

Algoritmo	Complessità peggiore	Complessità media	Complessità migliore
BUILDHEAP	$O(n \log_2 n)$ *oppure* $O(n)$[i]	$\Theta(n)$	=

4.3. PARTITION

Seleziona un pivot= $A[r]$, ..., e restituisce l'array $[\leq x; x; > x]$.

Si nota che $n = r - p + 1$.

[i] Ottenere questa complessità richiede un algoritmo più complicato.

```
Partition(A, p, r)
 1: x ← A[p]
 2: i ← p − 1
 3: j ← r + 1
 4: while TRUE do
 5:   repeat
 6:     j ← j − 1
 7:   until A[j] ≤ x
 8:   repeat
 9:     i ← i + 1
10:   until A[i] ≥ x
11:   if i < j then
12:     swap(A[i], A[j])
13:   else
14:     return j
15:   end if
16: end while
```

Algoritmo	Complessità peggiore	Complessità media	Complessità migliore
PARTITION	$\Theta(n)$	=	=

4.4.HEAPMAXIMUM

Restituisce la radice di un heap (il suo valore massimo).

Algoritmo	Complessità peggiore	Complessità media	Complessità migliore
HEAPMAXIMUM	$O(1)$	=	=

4.5.HEAPEXTRACTMAX

Restituisce la radice di un heap (il suo valore massimo) e sposta l'ultimo elemento nella radice.

Esempio: heapify

Algoritmo	Complessità peggiore	Complessità media	Complessità migliore
HEAPEXTRACTMAX	$O(\log n)$	=	=

4.6.HEAPINCREASEKEY

Aumenta il valore di un nodo, lo scambia con il padre fino a quando questo è più piccolo o arriva alla radice.

Algoritmo	Complessità peggiore	Complessità media	Complessità migliore
HEAPINCREASEKEY	$O(\log n)$	=	=

4.7.HEAPINSERT

Aggiunge un elemento in coda, lo imposta con HEAPINCREASEKEY.

Algoritmo	Complessità peggiore	Complessità media	Complessità migliore
HEAPINSERT	$O(\log n)$	=	=

4.8.Riepilogo delle complessità

Scrivere qualcosa qui.

5. Algoritmi di ordinamento in tempo lineare

5.1.Ordinamento tramite CountingSort

Lavora su elementi compresi fra 0 e k.

Dati n numeri di b bit e un intero positivo $r \leq b$, abbiamo complessità $\Theta\left(\frac{b}{r}(n + 2^r)\right)$.

Scegliendo $r = b$, otteniame complessità $\Theta(n)$, che è asintoticamente ottimale.

L'algoritmo **non** è *in-place*.

L'algoritmo è *stabile*.

```
Countign Sort(A, B, k)
 1: for i ← 0 to k do
 2:    C[i] ← 0
 3: end for
 4: for j ← 1 to length(A) do
 5:    C[ A[j] ] ← C[ A[j] ] + 1
 6: end for
 7: for i ← 2 to k do
 8:    C[i] ← C[i − 1] + C[i]
 9: end for
10: for j ← length(A) downto 1 do
11:    B[ C[ A[j] ] ] ← A[j]
12:    C[ A[j] ] ← C[ A[j] ] − 1
13: end for
```

Algoritmo	Complessità peggiore	Complessità media	Complessità migliore
CountingSort	$O(n \log n)$ $\Theta(n)$ $con\ k = O(n)$	?	?

5.2.Ordinamento tramite RadixSort

L'algoritmo è utilizzato per ordinare record con più campi chiave.

L'algoritmo era utilizzato per ordinare le schede perforate.

Nota: l'algoritmo sfrutta un algoritmo di ordinamento a scelta. Questo algoritmo deve essere stabile per garantire il corretto funzionamento di RadixSort.

L'algoritmo è *in-place* se l'algoritmo che sfrutta per l'ordinamento lo è.

L'algoritmo è *stabile*.

```
RadixSort(A, k)
 1: for i ← 1 to k do
 2:    SortStable(A, i) {Ordina A rispetto all'i-esima cifra}
 3: end for
```

Algoritmo	Complessità peggiore	Complessità media	Complessità migliore
RadixSort	$\Theta(n)$	?	?

5.3.Ordinamento tramite BUCKETSORT

L'algoritmo sfrutta InsertionSort per eseguire l'ordinamento. Cos'è un bucket?

Il numero di bucket è $n-1$. Il bucket i contiene i valori $[\frac{i}{10}; \frac{i+1}{10}]$.

```
Bucket Sort(A)
 1: n ← length(A)
 2: for i ← 0 to n − 1 do
 3:    InsertList(A[i], B[⌊nA[i]⌋])
 4: end for
 5: for i ← 0 to n − 1 do
 6:    sort(B[i])
 7: end for
 8: concatena le liste B[0], B[1], . . . , B[n − 1]
```

Algoritmo	Complessità peggiore	Complessità media	Complessità migliore
BUCKETSORT	$\Theta(n)$[i]	?	?

6. Algoritmi e strutture dati di supporto agli algoritmi di ordinamento in tempo lineare

6.1.RANDOMIZEDSELECT

Trova l'i-esimo elemento ordinato.

```
RandomizedSelect(A, p, r, i)
 1: if p = r then
 2:    return A[p]
 3: end if
 4: q ← RandomizedPartition(A, p, r)
 5: k ← q − p + 1
 6: if i ≤ k then
 7:    return RandomizedSelect(A, p, q, i)
 8: else
 9:    return RandomizedSelect(A, q + 1, r, i − k)
10: end if
```

Algoritmo	Complessità peggiore	Complessità media	Complessità migliore
RANDOMIZEDSELECT	$\Theta(n^2)$	$O(n)$	?

[i] Se la somma dei quadrati delle dimensioni dei bucket è lineare nel numero totale degli elementi.

6.2. SELECT

6.3. Operazioni su stack LIFO

6.4. Operazioni su code FIFO

6.5. Operazioni sulle liste concatenate

Semplici operazioni di ricerca, inserimento e cancellazione su liste.

Algoritmo	**Complessità peggiore**	**Complessità media**	**Complessità migliore**
LISTSEARCH	$\Theta(n)$	=	=
LISTINSERT	$O(1)$	=	=
LISTDELETE	$O(1)$	=	=

Hashing

7. Hashing

7.1. Tabella di indirizzamento diretto

7.2. Hash con chaining

Funzioni di hash

Metodo della divisione

Metodo della moltiplicazione

7.3. Hash con open addressing

Funzioni di hash

Scansione lineare

Scansione quadratica

7.4. Hashing doppio

Grafi

I grafi si suddividono nelle seguenti categorie:

1) Grafo indiretto aciclico
2) Grafo indiretto connesso
3) Albero (libero)
4) Albero radicato
5) Albero binario (vettore con $key; info; parent; left; right$)

8. Alberi Binari di Ricerca (BST)

8.1.Proprietà

- Tutti i nodi del sotto-albero sinistro di x sono più piccoli.
- Tutti i nodi del sotto-albero destro di x sono più grandi.
- Ogni nodo ha le seguenti puntatori ad altri nodi: $left, right, p$ (padre).

Notiamo che

$$h_{max} = n - 1 = \Theta(n)$$
$$h_{min} = \Theta(\log n)$$

8.2.Visite

Le visite hanno complessità $\Theta(n)$: attraversano un l'intero albero.
Il libro mostra questi algoritmi con ricorsione di coda e iterativi.

- Pre-order (stampa prima x, poi prosegue a sinistra e poi a destra).
- In-order (va a sinistra, stampa x, prosegue a destra).
- Post-order (va a sinistra, prosegue a destra, stampa x).

8.3.Ricerca

La ricerca prosegue confrontando il nodo ricercato con x e scegliendo se fermarsi, proseguire a destra o a sinistra. Facilmente implementabile sia con una ricorsione di cosa che con un algoritmo iterativo.
La ricerca ha complessità $O(h)$ in quanto deve percorrere potenzialmente un intero percorso dalla radice ad una foglia che potenzialmente è il più lungo: h.

```
TreeSearch(x, k)
1: if (x = NIL ) ∨ (k = KEY(x)) then
2:    return x
3: end if
4: if k ≤ KEY(x) then
5:    return TreeSearch(left(x), k)
6: else
7:    return TreeSearch(right(x), k)
8: end if
```

8.4.Massimo e minimo

Il minimo è l'elemento più a sinistra, il massimo quello più a destra.

Hanno complessità $O(h)$ in quanto devono percorrere un percorso intero radice-foglia, che è potenzialmente il più lungo: h.

```
TreeMin(x)
1: while left(x) ≠ NIL do
2:    x ← left(x)
3: end while
4: return x
```

8.5.Successore e predecessore

Il successore restituisce il minimo del sotto-albero destro di x. Qualora questo fosse NIL, viene restituito l'antenato y più prossimo di x il cui figlio sinistro è anche antenato di x, ovvero l'antenato più prossimo di x il cui primo nodo nel percorso che lo collega a x è a sinistra (suo figlio sinistro).

Il predecessore è simmetrico.

Questi algoritmi hanno complessità $O(h)$ in quanto potenzialmente saliamo fino alla radice o scendiamo fino ad una foglia.

```
TreeSuccessor(x)
1: if right(x) ≠ NIL then
2:    return TreeMin(right(x))
3: end if
4: while (y ≠ NIL ) ∧ (x = right(y)) do
5:    x ← y
6:    y ← p(y)
7: end while
8: return y
```

8.6.Inserimento e cancellazione

Inserimento

L'inserimento confronta i nodi a partire dalla radice con quello che vogliamo aggiungere (x) fino a raggiungere un riferimento a un sotto-albero NIL, quindi aggiunge il nodo x lì. Il nodo così aggiunto sarà quindi sempre una foglia.

L'operazione avrà complessità $O(h)$ in quanto il percorso attraversato è potenzialmente il più lungo: h.

```
TreeInsert(T, z)
1:  y ← NIL
2:  x ← root[T]
3:  while x ≠ NIL do
4:    y ← x
5:    if KEY(z) < KEY(x) then
6:      x ← left(x)
7:    else
8:      x ← right(x)
9:    end if
10: end while
11: p(z) ← y
12: if y = NIL then
13:   root(T) ← z
14: else
15:   if KEY(z) < KEY(y) then
16:     left(y) ← z
17:   else
18:     right(y) ← z
19:   end if
20: end if
```

Cancellazione

La cancellazione prevede tre casi:

1) Il nodo da cancellare non ha figli.

   Banale.

2) Il nodo da cancellare ha un solo figlio.

   Banale: il figlio prende il posto del padre.

3) Il nodo da cancellare ha due figli.

   Memorizziamo il valore del suo successore (o minimo del sotto-albero destro) (cfr. 8.5) e lo cancelliamo, quindi sostituiamo il nodo da cancellare con il valore memorizzato. Alla fine riaggiungiamo a z i figli del nodo cancellato, se ci sono O semplicemente gli agganciamo al padre del nodo cancellato?!? Ovvero: cosa si intende per "si copiano i dati satelliti di y in z"?

In particolare, l'algoritmo descritto dal libro di testo si divide in queste fasi: (z è il nodo da cancellare)

1) Assegna a y l'elemento da cancellare (z stesso se non ha due figli, altrimenti $successor(z)$).
2) Assegna a x il puntatore al figlio non-NIL di y, se esiste.
3) Cancella y ed eventualmente rimpiazza il suo vuoto con x.

   Caso particolare se chi viene cancellato è la radice dell'albero.
4) Se chi è stato cancellato (y) non era radice dell'albero, si aggiorna il puntatore $left$ o $right$ del padre.
5) Se è stato cancellato il successore di z e non z stesso, si copiano i dati satelliti di y in z.

L'operazione ha complessità $O(h)$ sempre per il fatto che potenzialmente attraversa il percorso più lungo.

```
TreeDelete(T, z)
1:  if (left(z) = NIL ) ∨ (right(z) = NIL ) then
2:     y ← z
3:  else
4:     y ← TreeSuccessor(z)
5:  end if
6:  if left(y) ≠ NIL  then
7:     x ← left(y)
8:  else
9:     x ← right(y)
10: end if
11: if x ≠ NIL  then
12:    p(x) ← p(y)
13: end if
14: if p[y] = NIL  then
15:    root(T) ← x
16: else
17:    if y = left(p(y)) then
18:       left(p(y)) ← x
19:    else
20:       right(p(y)) ← x
21:    end if
22: end if
23: if y ≠ z then
24:    KEY(z) ← KEY(y)
25: end if
26: return y
```

8.7.Riepilogo delle complessità

Algoritmo	Complessità
Visite	$O(n)$
Ricerca	$O(h)$
Massimo e minimo	$O(h)$
Successore e predecessore	$O(h)$
Inserimento e cancellazione	$O(h)$

9. Alberi rosso-neri (Red-Black tree)

Gli alberi rosso-neri sono Alberi Binari di Ricerca (BST) che garantiscono che le operazioni elementari richiedano un tempo $O(\log n)$.

9.1.Proprietà

1) I nodi sono rossi o neri.
2) La radice è nera.
3) Le foglie sono nere.
4) I figli dei nodi rossi sono neri.
5) Ogni cammino radice-foglia ha lo stesso numero di nodi neri.

I cammini radice-foglia risultano lunghi al massimo il doppio di ogni altro:

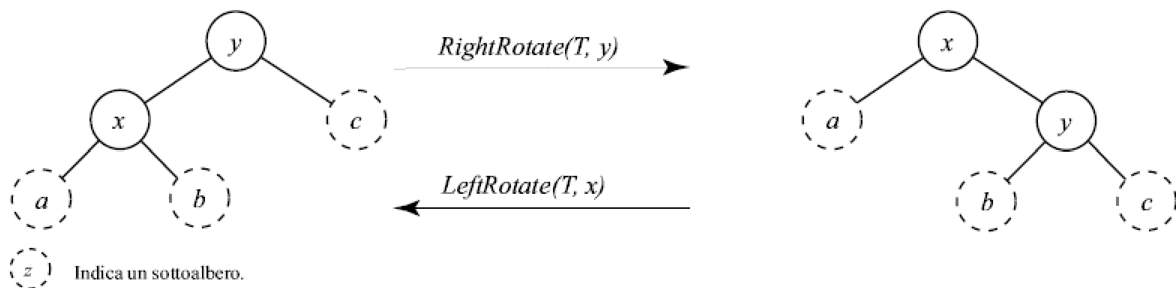
$$h \leq 2 \log{(\mathrm{n}+1)}$$

Nel libro di testo invece di usare NIL nei puntatori, viene usato un nodo $nil[T]$ colorato di nero. Questo perché ci è comodo usare $nil[T]$ al posto delle foglie, e come padre della radice. I suoi valori sono ininfluenti escluso il colore (nero, in quanto foglia).

Viene definita l'altezza nera $bh(x)$ di un nodo x il numero di nodi neri nel percorso fra x e una foglia, x escluso.

9.2. Rotazioni

Le rotazioni preservano le proprietà dei RBT e hanno costo $O(1)$. Agiscono solo su un numero fisso di nodi, modificandone esclusivamente i puntatori.



L'algoritmo LEFT-ROTATE suppone che il il figlio destro di x non sia $nil[T]$ e procede nel modo seguente:

1) Viene memorizzato il figlio destro di x in y.
2) Il figlio sinistro di y prende il posto di quello destro di x.
3) y prende il posto di x. (caso particolare se x era radice)
4) x diventa figlio sinistro di y.

```
LeftRotate(T, x)
 1: y ← right(x)
 2: right(x) ← left(y)
 3: if left(y) ≠ NIL then
 4:    p(left(y)) ← x
 5:    p(y) ← p(x)
 6:    if p(x) = NIL then
 7:       root(x) ← y
 8:    else
 9:       if x = left(p(x)) then
10:          left(p(x)) ← y
11:       else
12:          right(p(x)) ← y
13:       end if
14:    end if
15: end if
16: left(y) ← x
17: p(x) ← y
```

9.3.Inserimento

L'inserimento viene effettuato come in un normale Albero Binario di Ricerca (BST), ma alla fine il nodo inserito viene colorato di rosso, i suoi figli vengono impostati a $nil[T]$ e viene eseguita una procedura "RB-INSERT-FIXUP"per ristabilire le proprietà dell'RBT descritta qui di seguito.

RB-INSERT-FIXUP

Questo algoritmo ristabilisce le proprietà di un RBT dopo un inserimento.

Notiamo che z è rosso.

Le proprietà che possono esser state violate in seguito all'inserimento del nodo z sono le seguenti:

1) Se z è la radice, la proprietà 2 è violata in quanto la radice deve essere nera.
2) Se il padre di z è rosso, la proprietà 4 è violata in quanto i nodi rossi deveno avere solo figli neri.

Notiamo che non possono essere presenti entrambe le violazioni contemporaneamente[i].

La proprietà 2 viene banalmente ripristinata colorando di nero la radice dell'albero.

D'altra parte, esistono tre casi in cui la proprietà 4 viene violata, se escludiamo i loro simmetrici:

Consideriamo y lo zio di z ($y = right[p[p[z]]]$ o simmetrico).

Notiamo che, essendo la proprietà 4 violata, $p[z]$ è rosso e, di conseguenza, $p[p[z]]$ non è $nil[T]$ ed è nero[ii].

Caso 1 – y è rosso

Notiamo che sia $p[z]$ che y sono rossi.

Poiché $p[p[z]]$ è nero (vedi sopra), coloriamo entrambi di nero e $p[p[z]]$ di rosso.

Caso 2 – y è nero e z è figlio destro

Notiamo che z è figlio destro.

Effettuando una rotazione sinistra, z diventa figlio sinistro e ricadiamo nel caso 3.

Notiamo che questa rotazione non influisce sull'altezza nera (i nodi sono rossi)

Caso 3 – y è nero e z è figlio sinistro

Coloriamo $p[z]$ (suo padre) di nero e $p[p[z]]$ (suo nonno) di rosso.

Eseguiamo quindi una rotazione destra su $p[p[z]]$ (suo nonno)

Analisi

L'inserimento escludendo il "fix", ha complessità $O(\log n)$.

---

[i] Vedasi libro di testo § 13.3 – Inizializzazione – c (pag. 239).
[ii] In quanto la radice deve essere nera.

RB-INSERT-FIXUP ha complessità $O(\log n)$ in quanto il ciclo **while** viene ripetuto esclusivamente nel caso 1 e il puntatore z sale di 2 livelli per volta.

E' interessante notare che non vengono mai effettuate più di 2 rotazioni da parte di **while** in quanto nei casi 2 e 3 esso termina.

```
RBTreeInsert(T, x)
 1: TreeInsert(T, x)
 2: color[x] ← RED
 3: while (x ≠ root(T)) ∧ (color[p(x)] ≠ BLACK) do
 4:   if p(x) = left(p(p(x))) then
 5:     y ← right(p(p(x)))
 6:     if color[y] = RED then
 7:       color[p(x)] ← BLACK
 8:       color[y] ← BLACK
 9:       color[p(p(x))] ← RED
10:       x ← p(p(x))
11:     else
12:       if x = right(p(x)) then
13:         x ← p(x)
14:         LeftRotate(T, x)
15:       end if
16:       color[p(x)] ← BLACK
17:       color[p(p(x))] ← RED
18:       RightRotate(T, p(p(x)))
19:     end if
20:   else
21:     trattiamo i casi simmetrici...
22:   end if
23: end while
24: color[root(T)] ← BLACK
```

9.4.Cancellazione

Ricordiamo che y è l'elemento da cancellare nell'albero: z stesso se z non ha due figli, altrimenti $successor(z)$ (Vedi §8.6).

Il procedimento è similare all'inserimento: dopo aver cancellato il nodo come in un normale Binary Search Tree (BST), se il nodo y è nero, viene eseguita la procedura RB-DELETE-FIXUP che ripristina le proprietà dell'RBT. (Se y è rosso le proprietà rimangono valide[i]).

Questa operazione ha complessità $O(\log n)$.

Notiamo che la procedura di cancellazione presenta una differenza importante: l'assegnazione

$$7: p[x] \leftarrow p[y]$$

è ora incondizionata al contrario della procedura per i Binary Search Tree (BST):

$$7: \textbf{if } x \neq NIL \; 8: \textbf{then } p[x] \leftarrow p[y]$$

[i] Infatti le altezze nere nell'albero non sono cambiate, non sono stati creati nodi rossi consecutivi e la radice resta nera (se il nodo y fosse stato rosso non sarebbe potuto essere la radice)

Questa modifica garantisce che il padre di x adesso sia il nodo che in precedenza era il padre di y, anche se è $nil[T]$.

Ricordiamo che x è il puntatore al figlio non-$nil[T]$ di y, se esiste, altrimenti $nil[T]$ (Vedi §8.6).

RB-DELETE-FIXUP

Questa procedura ripristina le violazioni alle proprietà dei RBT introdotte dalla cancellazione di un nodo se il nodo y è nero.

Si possono verificare i seguenti problemi:

1) Se y era radice e un figlio rosso di y diventa radice, la proprietà 2 è violata, infatti la radice dovrebbe essere nera.
2) Se i nodi x e $p[y]$, ora anche $p[x]$, erano rossi, la proprietà 4 è violata, infatti i nodi rossi non dovrebbero avere figli rossi.
3) In seguito alla rimozione di y (nero), i percorsi che contenevano y violano la proprietà 5, infatti ogni percorso radice-foglia dovrebbe avere lo stesso numero di nodi neri.

   Questo problema viene risolto assegnando un peso extra ai percorsi che contengono x. Questo lo facciamo "aggiungendo" il colore del nodo eliminato y a suo figlio. Ora però x non è nè rosso nè nero: è "doppiamente nero" o "rosso e nero". Il "rosso e nero" diventa semplicemente nero, mentre il "nero extra" viene spostato verso l'alto fino a quando:

   a. Non viene spostato su un nodo rosso. In tal caso il rosso viene perduto e il nodo diventa nero (come prima).
   b. Si arriva alla radice: il "nero extra" viene perso.
   c. Si riescono ad effettuare opportune rotazioni e ricolorazioni.

L'algoritmo procede nel seguente modo. Definiamo ω come fratello di x.

Caso 1 – ω è rosso.

Descrizione necessaria.

Caso 2 – ω è nero e i suoi figli sono neri.

Descrizione necessaria.

Caso 3 – ω è nero, suo figlio sinistro è rosso e suo figlio destro è nero.

Descrizione necessaria.

Caso 4 – ω è nero e suo figlio destro è rosso.

Descrizione necessaria.

RBTreeDelete(T, z)

1: **if** $(left(z) = \text{NIL}) \vee (right(z) = \text{NIL})$ **then**
2: $\quad y \leftarrow z$
3: **else**
4: $\quad y \leftarrow \text{TreeSuccessor}(z)$
5: **end if**
6: **if** $left(y) \neq \text{NIL}$ **then**
7: $\quad x \leftarrow left(y)$
8: **else**
9: $\quad x \leftarrow right(y)$
10: **end if**
11: **if** $x \neq \text{NIL}$ **then**
12: $\quad p[x] \leftarrow p[y]$
13: **end if**
14: **if** $p(y) = \text{NIL}$ **then**
15: $\quad root(T) \leftarrow x$
16: **else**
17: $\quad$ **if** $y = left(p(y))$ **then**
18: $\quad\quad left(p(y)) \leftarrow x$
19: $\quad$ **else**
20: $\quad\quad right(p(y)) \leftarrow x$
21: $\quad$ **end if**
22: **end if**
23: **if** $y \neq z$ **then**
24: $\quad \text{KEY}(z) \leftarrow \text{KEY}(y)$
25: **end if**
26: **if** $\text{color}[x] = \text{BLACK}$ **then**
27: $\quad \text{RBDeleteFixUp}(T, x)$
28: **end if**
29: return y

RBDeleteFixUp(T, x)

```
1:  while (x ≠ root(T)) ∧ (color[x] = BLACK) do
2:    if (x = left(p(x))) then
3:      w ← right(p(x))
4:      if color[w] = RED then
5:        color[w] ← BLACK
6:        color[p(x)] ← RED
7:        LeftRotate(T, p(x))
8:        w ← right(p(x))
9:      end if
10:     if (color[left(w)] = BLACK) ∧ (color[right(w)] = BLACK) then
11:       color[w] ← RED
12:       x ← p(x)
13:     else
14:       if color[right(w)] = BLACK then
15:         color[left(w)] ← BLACK
16:         color[w] ← RED
17:         RightRotate(T, w)
18:         w ← right(p(x))
19:       end if
20:       color[w] ← color[p(x)]
21:       color[p(x)] ← BLACK
22:       color[right(w)] ← BLACK
23:       LeftRotate(T, p(x))
24:       x ← p(x)
25:     end if
26:   else
27:     trattiamo i casi simmetrici...
28:   end if
29: end while
30: color[x] ← BLACK
```

9.5.Un'applicazione: Join

10. Balanced-Tree (B-Tree)

10.1. Ricerca

BTreeSearch(x, k)

```
1:  i ← 1
2:  while (i ≤ n[x]) ∧ (KEY_i(x) < k) do
3:    i ← i + 1
4:  end while
5:  if (i ≤ n[x]) ∧ (k = KEY_i(x)) then
6:    return(x, i)
7:  end if
8:  if leaf(x) then
9:    return NIL
10: else
11:   DiskRead(c_i[x])
12:   return BTreeSearch(c_i[k], k)
13: end if
```

10.2. Creazione

BTreeCreate(T)

```
1: x ← AllocateNode()
2: leaf(x) ← TRUE
3: n[x] ← 0
4: DiskWrite(x)
5: root[T] ← x
```

10.3. Split di nodi pieni

BTreeSplit(x, i, y)

```
1: z ← AllocateNode()
2: leaf(z) ← leaf(y)
3: n[z] ← t − 1
4: for j ← 1 to t − 1 do
5:     KEY_j(z) ← KEY_{j+t}(y)
6: end for
7: if ¬(leaf(y)) then
8:     for j ← 1 to t do
9:         c_j[z] ← c_{j+t}[y]
10:    end for
11: end if
12: n[y] ← t − 1
13: for j ← n[x] + 1 downto i + 1 do
14:    c_{j+1}[x] ← c_j[x]
15: end for
16: c_{i+1}[x] ← z
17: for j ← n[x] downto i do
18:    KEY_{j+1}(x) ← KEY_j(x)
19: end for
20: KEY_i(x) ← KEY_t(y)
21: n[x] ← n[x] + 1
22: DiskWrite(y); DiskWrite(x); DiskWrite(z)
```

10.4. Inserimento

BTreeInsert(T, k)

```
1: r ← root(T)
2: if n[r] = 2t − 1 then
3:     s ← AllocateNode()
4:     root(T) ← s
5:     n[s] ← 0
6:     c_1[s] ← r
7:     BTreeSplit(s, 1, r)
8:     BTreeInsertNonFull(s, k)
9: else
10:    BTreeInsertNonFull(r, k)
11: end if
```

10.5. Cancellazione

11. Insiemi disgiunti – Union Find

11.1. Make

11.2. Union

11.3. Find

12. Algoritmi sui grafi

12.1. Breadth-first search

BFS(G, s)

1: **for** each $v \in V \setminus \{s\}$ **do**
2: color$[v] \leftarrow$ WHITE
3: $d(v) \leftarrow \infty$
4: $\pi(v) \leftarrow$ NIL
5: **end for**
6: color$[s] \leftarrow$ GRAY
7: $d(s) \leftarrow 0$
8: $\pi(s) \leftarrow$ NIL
9: $Q \leftarrow \{s\}$
10: **while** $Q \neq \emptyset$ **do**
11: $u \leftarrow head(q)$
12: **for** each $v \in adj[u]$ **do**
13: **if** color$[v] =$ WHITE **then**
14: color$[v] \leftarrow$ GRAY
15: $d(v) \leftarrow d(u) + 1$
16: $\pi(v) \leftarrow u$
17: enqueue(Q, v)
18: **end if**
19: **end for**
20: dequeue(Q)
21: color$[u] \leftarrow$ BLACK
22: **end while**

12.2. Breadth-first tree

12.3. Depth-first search

DFS(G)

```
1: for each v ∈ V do
2:    color[v] ← WHITE
3:    π(v) ← NIL
4: end for
5: time ← 0
6: for each u ∈ V do
7:    if color[u] = WHITE then
8:       DFSVisit(u)
9:    end if
10: end for
```

DFSVisit(u)

```
1: color[u] ← GRAY
2: d(u) ← time ← time + 1
3: for each v ∈ adj[u] do
4:    if color[v] = WHITE then
5:       π[v] ← u
6:       DFSVisit(v)
7:    end if
8: end for
9: color[u] ← BLACK
10: f(u) ← time ← time + 1
```

12.4. Classificazione degli archi

12.5. Topological sort

12.6. Componenti fortemente connesse

SCC(G)

```
1: DFS(G)
2: G ← ComponenteTrasposta(G)
3: for each u ∈ V do
4:    color[u] ← WHITE
5:    π(u) ← NIL
6:    F[f(u)] ← u {vettore che accoglie i vertici nell'ordine in cui si è finito
      di analizzarli}
7: end for
8: time ← 0
9: for i = size(F) downto 1 do
10:    if (F[i] ≠ NIL ) ∧ (color[F[i]] = WHITE) then
11:       DFSVisit(F[i])
12:    end if
13: end for
```

ComponenteTrasposta(G)

```
1: for each v ∈ V do
2:    for each u ∈ adj[v] do
3:       adj_1[u] ← v
4:    end for
5: end for
6: for each v ∈ V do
7:    adj[v] ← adj_1[v]
8: end for
```

13. Minimum spanning tree (MST)

13.1. Algortimo generico

MST(G)

```
1: A ← ∅
2: while A non forma un MST do
3:    find(u, v) safe-edge per A
4:    A ← A ∪ {(u, v)}
5: end while
6: return A
```

13.2. Algoritmo di Kruskal

MST-Kruskal(G, w)

```
1: A ← ∅
2: for each v ∈ V do
3:    MAKE(v)
4: end for
5: sort(E, w) {ordiniamo in maniera crescente rispetto al peso w degli
   archi}
6: for each (u, v) ∈ E do
7:    if FIND(u) ≠ FIND(v) then
8:       A ← A ∪ {(u, v)}
9:       UNION(u, v)
10:   end if
11: end for
12: return A
```

13.3. Algoritmo di Prim

```
MST-Prim(G, w, r)
 1: Q ← V
 2: for each u ∈ V do
 3:    KEY(u) ← ∞
 4: end for
 5: π(r) ← NIL
 6: KEY(r) ← 0
 7: BuildHeap(Q)
 8: while Q ≠ ∅ do
 9:    u ← ExtractMin(Q)
10:    for each v ∈ adj[u] do
11:       if (v ∈ Q) ∧ (w(u, v) < KEY(v)) then
12:          π(v) ← u
13:          KEY(v) ← w(u, v)
14:          DecreaseKey(v, Q)
15:       end if
16:    end for
17: end while
```

14. Single source shortest path problem

14.1. Algoritmo di Dijkstra

```
Dijkstra(G, w, s)
 1: ISS(G, s)
 2: Q ← V
 3: S ← ∅
 4: while Q ≠ ∅ do
 5:    u ← EstractMin(Q)
 6:    S ← S ∪ {u}
 7:    for each v ∈ adj[u] do
 8:       Relax(u, v, w)
 9:    end for
10: end while
```

```
Relax(u, v, w)
 1: if d(v) > d(u) + w(u, v) then
 2:    d(v) ← d(u) + w(u, v)
 3:    π(v) ← u
 4: end if
```

```
InitializeSingleSource(G, s)
 1: for each v ∈ V do
 2:    d(v) ← ∞
 3:    π(v) ← NIL
 4: end for
 5: d(s) ← 0
```