

Programmazione Algoritmi e Computabilità

2024-2025

Introduzione

Il corso si prefigge l'obiettivo di fornire conoscenze, sia teoriche che pratiche, per la progettazione e implementazione di un'applicazione *software*, mediante un processo di sviluppo *agile* e *API-led*. Si punterà, su piccola scala, al design di algoritmi efficienti e computabili, con le relative strutture dati. Questo primo obiettivo sarà raggiunto studiando la teoria della complessità e della computabilità algoritmica. Su una scala maggiore, si studierà l'architettura *software*. Questo secondo obiettivo sarà raggiunto impiegando il linguaggio Java, abbinato a *middleware*, *framework* e librerie, insieme a *tool* legati all'IDE Eclipse.

Complessità degli algoritmi

Riprendiamo i concetti di analisi matematica, come la notazione asintotica, per stabilire l'appartenenza del tempo di esecuzione alle classi di complessità. La funzione $f(n)$ rappresenta il tempo di esecuzione o l'occupazione di risorse di un algoritmo su un input di dimensione n . Utilizzeremo questa funzione come astrazione in informatica per nascondere complessità irrilevanti.

La notazione O ("o grande") rappresenta un limite superiore, mentre Ω rappresenta un limite inferiore. La notazione Θ indica limiti sia superiori che inferiori esatti. Le prime due notazioni sono generalmente più facili da trovare.

La notazione O è la più comunemente definita. Una funzione $f(n)$ appartiene alla classe $O(g(n))$ se esistono due costanti $c > 0$ e $n_0 > 0$ tali che $f(n) \leq cg(n)$ per ogni $n \geq n_0$. In altre parole, a partire da n_0 , la curva analizzata sta sempre al di sotto di $cg(n)$.

La notazione Ω è utilizzata per indicare un limite inferiore. Una funzione $f(n)$ appartiene alla classe $\Omega(g(n))$ se esistono due costanti $c > 0$ e $n_0 \geq 0$ tali che $f(n) \geq cg(n)$ per ogni $n \geq n_0$.

La notazione Θ è utilizzata per indicare limiti sia superiori che inferiori esatti. Una funzione $f(n)$ appartiene alla classe $\Theta(g(n))$ se esistono tre costanti $c_1, c_2 > 0$ e $n_0 \geq 0$ tali che $c_1g(n) \leq f(n) \leq c_2g(n)$ per ogni $n \geq n_0$.

È importante notare che stiamo facendo un abuso di notazione. In realtà, dovremmo usare il simbolo \in invece di $=$ per indicare l'appartenenza alle classi di complessità. Ad esempio, dovremmo scrivere $f(n) \in O(g(n))$ per indicare che O è un limite superiore asintotico per la funzione appartenente alla sua classe.

Teorema: date due funzioni $f(n)$ e $g(n)$, $f(n) \in \Theta(g(n))$ se e solo se $f(n) \in O(g(n))$ e $f(n) \in \Omega(g(n))$. (ricontrollare dalle slide)

Ad esempio, $O(n^2)$ e $\Omega(n^2)$ sono limiti inferiori e superiori per qualsiasi polinomio quadratico, aggiustando adeguatamente i parametri del polinomio. Vale anche $\Theta(n^2)$. Un polinomio quadratico è anche $O(n^3)$ ma non $\Omega(n^3)$. Anche quando calcoliamo O , ci conviene stare vicini al sup di Θ .

Proprietà: - *transitiva*: vale per tutte e tre le notazioni. Si dimostra immediatamente applicando la definizione - *riflessiva*: vale per tutte e tre. Ogni funzione è limite inferiore e superiore per sé stessa. - *simmetria*: vale solo per Θ . Se una funzione è nello stesso ordine di un'altra, allora l'altra è nello stesso ordine della prima. - per le altre due vale la simmetria transposta. Il limite superiore per una diventa limite inferiore per l'altra.

Le notazioni di O , Θ e Ω sembrano simili agli ordinamenti totali dei numeri. Non vale però la tricotomia (cioè che debba valere per forza minore, uguale o maggiore). Le funzioni oscillanti rendono difficile il confronto asintotico.

Regole di semplificazione: 1. posso ignorare le costanti moltiplicative nell'ordine 2. la somma di due funzioni avrà O somma tra le O dei due membri (casi di somma: sequenze, *branching*. Per il *branching* si conta il costo del ramo più lento. Per le *subroutine*, si somma il loro costo) 3. il prodotto di funzioni avrà O pari al prodotto degli O dei membri (si applica ai cicli, in base all' O del numero di iterazioni e all' O del corpo)

In caso di dubbi sull'ordine, si applica il teorema dei limiti. Una funzione è dello stesso O di un'altra se il limite all'infinito del loro rapporto è zero. Notare che non vale l'inverso. L'inverso parziale è che il limite del loro rapporto

sia finito se esiste. Se il rapporto è finito, allora il numeratore è Ω del denominatore. Non vale l'inverso, posso solo dire che se esiste allora è una quantità positiva. Se il limite del rapporto è finito positivo, allora vale Θ . Se il limite non esiste, non sono asintoticamente confrontabili.

Limiti temporali

Un problema computazionale ha complessità $O(f(n))$ (*upper bound*) se esiste un algoritmo per la sua risoluzione con complessità $O(f(n))$. Un problema computazionale ha complessità $\Omega(f(n))$ (*lower bound*) se ogni algoritmo per la sua risoluzione ha complessità $\Omega(f(n))$. Riuscendo a dimostrare che un problema ha delimitazione inferiore $\Omega(f(n))$, e trovando un algoritmo che ha delimitazione superiore $O(f(n))$, allora si ottiene, a meno di costanti, un algoritmo risolutivo ottimale.

Per la ricerca del minimo in insiemi non ordinati di n elementi la complessità inferiore è $\Omega(n)$ perché è necessario scandire almeno una volta l'intero insieme. Un algoritmo $O(n)$ è dunque ottimo. Il *mergesort* ha complessità $n \log n$ per un problema $n \log n$ ed è dunque ottimale. Il *bubblesort* è subottimale perché ha complessità n^2 .

La dimensione dell'*input* non è il solo criterio in base al quale valutare l'esecuzione degli algoritmi. Distinguiamo caso migliore, peggiore, medio. Dato tempo(I) il tempo di esecuzione dell'algoritmo sull'istanza I , abbiamo

$$\begin{aligned} T_{worst}(n) &= \max_{\text{istanze } I \text{ di dimensione } n} \{\text{tempo}(I)\} \\ T_{best}(n) &= \min_{\text{istanze } I \text{ di dimensione } n} \{\text{tempo}(I)\} \\ T_{avg}(n) &= \sum_{\text{istanze } I \text{ di dimensione } n} \{\mathcal{P}(I) \text{ tempo}(I)\} \end{aligned}$$

dove $\mathcal{P}(I)$ è la distribuzione di probabilità dei casi.

Per la ricerca sequenziale (cerca lungo l'intera lista, fermati alla prima istanza trovata), il tempo migliore è 1 (elemento cercato in testa), il peggiore è n (elemento in ultima posizione), media $(n+1)/2$. Questo vale se assumiamo che le istanze siano equidistribuite e che la probabilità di ogni elemento sia $1/n$. Applicando la formula di prima esce la probabilità del caso medio (serie aritmetica).

Spesso il calcolo del caso medio è molto complesso e, in molti casi, si scopre che il caso medio è molto vicino al caso peggiore. Pertanto, è spesso conveniente determinare il limite superiore nel caso peggiore.

La ricerca binaria per liste ordinate si basa su tre indici: inferiore, medio, superiore. Ad ogni passo si confronta l'elemento all'indice medio con il *target* della ricerca. In caso di differenza, restringiamo la ricerca alla metà corrispondente di lista. In un numero finito di passi l'algoritmo collassa su di un solo elemento. Se esso non è quello cercato, la ricerca fallisce. In alcuni casi può capitare che gli indici si incrocino, e anche in questo caso si può dedurre che l'elemento cercato non sia presente in lista. Il tempo migliore è costante (l'elemento centrale è quello cercato, e viene trovato subito): $\Theta(1)$. Il tempo peggiore si ha quando l'elemento cercato è l'ultimo considerato, o non c'è. Dato che si procede per dimezzamenti della lista, il numero di passi è pari $\log_2 n$ del numero di elementi. Il caso medio è $\log n - 1 + \frac{1}{n}$, simile al caso peggiore.

Strutture dati

Si analizzeranno ora le strutture dati nell'ottica dell'analisi degli algoritmi. Si introduca innanzitutto il concetto di *Abstract Data Type*. Esso risponde alla domanda “*che cosa?*”, nel senso che definisce la semantica, le operazioni correlate, gli ingressi, le uscite e i vincoli legati al tipo di dati in questione. La struttura dati vera e propria, invece, è una concretizzazione dell'ADT, e risponde alla domanda “*come?*”, fornendo i dettagli implementativi. Nella programmazione a oggetti, gli ADT corrispondono alle interfacce, mentre i tipi concreti corrispondono alle classi. Per fornire un esempio, definiamo astrattamente un dizionario. I dati sono un insieme di coppie chiave / valore. Le operazioni consentite sono l'inserimento (aggiunta di una nuova coppia all'insieme), la cancellazione (rimozione di una coppia dall'insieme data la sua chiave) e ricerca (recuperare una coppia dall'insieme data la chiave). Volendo essere più precisi, potremmo anche definire il tipo di ritorno di ogni operazione, compreso il valore restituito in caso di errori. In Java, la definizione prende la forma seguente:

```
public interface Dizionario {
    public void insert(Object e, Comparable k);
    public void delete(Comparable k);
    public Object search(Comparable k);
}
```

Questa implementazione è non generica (usiamo `Object` e la versione non generica di `Comparable`). Volendo sfruttare i tipi generici, invece:

```
public interface Dizionario <E, K extends Comparable <? Super k >> {
    public void insert(E e, K k);
    public void delete(K k);
    public E search(K k);
}
```

Rappresentazioni indicizzate e collegate

Le tecniche di rappresentazione dei dati possono essere divise in due categorie: *rappresentazioni indicizzate* e *rappresentazioni collegate*. Il primo raggruppamento memorizza i dati in aree contigue di memoria, come in un *array*. Questo permette di accedere direttamente ai dati tramite un indice. Di contro, però, la dimensione è fissa, causando spreco di spazio dovuto alla frammentazione interna, da mitigare con riallocazioni frequenti che richiedono un tempo lineare. Il secondo tipo utilizza invece *record* spazialmente separati, ma collegati logicamente da puntatori. L'effettiva compattezza e lo spazio occupato dipendono dalla specifica implementazione. Esempi di struttura collegata sono la lista semplice, la lista doppiamente collegata e la lista circolare doppiamente collegata. Le rappresentazioni collegate hanno dimensioni variabili, veramente dinamiche. Le aggiunte e le rimozioni sono effettuate a tempo costante. L'accesso dei dati invece richiede un tempo medio lineare, perché è necessariamente sequenziale.

Riprendiamo l'esempio relativo all'implementazione del dizionario. Volendolo costruire come struttura dati collegata, possiamo basarlo sull'array. L'inserimento di una nuova coppia richiede la riallocazione dell'array. Si alloca un nuovo spazio di memoria con lo spazio per un elemento in più. Successivamente lo si riempie copiando il contenuto della versione precedente dell'array finché non si raggiunge l'elemento precedente, in ordine di chiave, rispetto all'elemento da inserire. Si inserisce infine l'elemento nuovo, e dopo di esso tutti gli elementi preesistenti che lo seguono in ordine di chiave. La cancellazione richiede di trovare prima l'indice dell'elemento da cancellare, e in seguito di riallocare l'array con un posto in meno, saltando nella copia l'elemento da scartare. Entrambe le operazioni di inserimento e cancellazione hanno tempo lineare. Dato che l'array è ordinato, la ricerca può essere effettuata in tempo logaritmico sfruttando l'algoritmo di ricerca binaria. Un'implementazione collegata del dizionario può essere generata utilizzando una lista collegata. Le nuove coppie si inseriscono in testa, a tempo costante. L'operazione, infatti, richiede l'aggiornamento di un puntatore (quello della testa, se la lista è vuota) o due puntatori (testa e riferimento al successivo, se la lista non è vuota). L'eliminazione è a tempo lineare perché richiede di scandire sequenzialmente la lista per trovare l'elemento desiderato, e di riscrivere i puntatori per collegare tra loro gli elementi precedente e successivo ad esso. Anche la ricerca è sequenziale e dunque a tempo lineare.

Definiamo la struttura dati astratta per una pila. Si tratta di una semplice sequenza di elementi, e la particolarità sta nella politica d'accesso LIFO (*Last In, First Out*). Possiamo definire operazioni per verificare se sia vuota o meno, per inserire ed eliminare elementi dalla cima, o per osservare tale cima in sola lettura. Anche la pila, come il dizionario, può essere implementata come struttura indicizzata o collegata. Una possibile implementazione indicizzata in Java prende la forma seguente:

```
public interface Pila {
    private int maxSize;
    private int top;
    private Object[] listArray;
}

class LPila implements Pila {
    private Record top;
    private int size;
    ...
}
```

La struttura dati astratta di una coda deve permettere due operazioni (*enqueue*, “accoda”, e *dequeue*, “rimuovi”), e di accedere in sola lettura all'estremo di interesse. La coda può anche essere bilaterale (nota come *deque*, ovvero *Double-Ended Queue*), e permettere accodamento e scodamento di elementi da entrambi gli estremi. L'implementazione indicizzata di una coda può essere effettuata tenendo due indici, *front* e *rear*, per il primo e ultimo elemento della coda all'interno del vettore. Dato che inserimenti e cancellazioni possono lasciare spazi vuoti sia in testa che in coda, conviene interpretare l'array come una struttura circolare e leggerlo “a orologio” lungo un verso stabilito. Questo riduce la necessità di riallocazioni dovute al ricompattamento dei dati. A livello implementativo l'effetto può essere ottenuto usando l'aritmetica modulare nel calcolo degli indici. In questa versione circolare è possibile distinguere tra coda piena e vuota soltanto se è presente almeno uno spazio vuoto tra il primo e l'ultimo elemento. Non è possibile infatti dedurre dagli indici *front* e *rear* se l'array sia completamente pieno o completamente vuoto, perché in entrambi i casi la loro posizione si sovrappone. L'implementazione collegata per le code e le pile è molto semplice da gestire. Tutte le operazioni sono a tempo costante perché si opera sempre e solo

sugli elementi in testa (o eventualmente in coda). Un albero radicato è una coppia $T = (N, A)$ costituita da un insieme N di *nodi* e da un insieme A di coppie di nodi, dette *archi*. I dati sono contenuti nei nodi. Le relazioni gerarchiche tra dati sono rappresentate dagli archi che li collegano. Chiamiamo *grado* di un nodo il suo numero di figli. Chiamiamo *foglie* i nodi senza figli. Chiamiamo *cammino* la sequenza di nodi che collega un nodo antenato ad un nodo discendente tramite sole relazioni padre-figlio. La *lunghezza* di un cammino corrisponde al numero di archi attraversati. Il cammino tra la radice ed un qualsiasi nodo è univoco. La lunghezza di tale cammino è detta *profondità* del nodo. L'*altezza* dell'albero è la lunghezza del cammino più lungo possibile al suo interno. Un albero k -ario completo è un albero in cui tutte le foglie hanno la stessa profondità e tutti i nodi interni hanno grado k . Esso ha k^h foglie a profondità h , e $\frac{k^h-1}{k-1}$ nodi interni. Un albero binario completo ha dunque $2^h - 1$ nodi interni e 2^h foglie.

Rappresentiamo il tipo di dato astratto per un albero:

tipo Albero:

dati:

un insieme di nodi (di tipo nodo) e un insieme di archi

operazioni:

numNodi() -> intero

restituisce il numero di nodi presenti nell'albero

grado(nodo v) -> intero

restituisce il numero di figli del nodo v

padre(nodo v) -> nodo

restituisce il padre del nodo v dell'albero, o null se v è la radice

figli(nodo v) -> (nodo, nodo, ..., nodo)

restituisce, uno dopo l'altro, i figli del nodo v

aggiungiNodo(nodo u) -> nodo

inserisce un nuovo nodo v come figlio di u nell'albero e lo restituisce. Se v è il primo nodo ad essere inserito nell'albero, esso diventa la radice (e u viene ignorato).

aggiungiSottoalbero(Albero a, nodo u)

inserisce nell'albero il sottoalbero a in modo che la radice di a diventi figlia di u.

rimuoviSottoalbero(nodo v) -> Albero

stacca e restituisce l'intero sottoalbero radicato in v. L'operazione cancella dall'albero il nodo v e tutti i suoi discendenti.

Le rappresentazioni concrete degli alberi sono sempre collegate. Per alberi nei quali ogni nodo abbia un numero limitato di figli, possiamo definire un numero limitato di puntatori ai figli. Quando il numero di figli necessita di essere arbitrario, si possono invece tenere un puntatore al primo figlio e un puntatore al fratello successivo.

Gli alberi più usati sono gli alberi binari. Di essi possiamo dare una definizione ricorsiva. L'insieme vuoto \emptyset è un albero binario. Se T_s e T_d sono alberi binari ed r è un *nodo* allora la terna ordinata (r, T_s, T_d) è un albero binario. In memoria ogni nodo ha un puntatore al padre e due puntatori ai figli. Gli alberi binari si possono utilizzare, tra le altre cose, per rappresentare espressioni matematiche e per costruire classificatori. Definiamo *algoritmi di visita* gli algoritmi che consentono l'accesso sistematico ai nodi e agli archi di un albero. Gli algoritmi di visita si distinguono in base all'ordine di accesso ai nodi. Analizziamo innanzitutto un algoritmo di visita generico:

algoritmo visitaGenerica(nodo r)

S <- {r}

while (S != vuoto) do

estrai un nodo u da S

visita il nodo u

S <- S unito {figli di u}

Esso richiede tempo $O(n)$ per visitare un albero con n nodi a partire dalla radice. Un possibile algoritmo non generico è la visita in profondità (DF), che parte dalla radice e visita i nodi di figlio in figlio fino al raggiungimento di una foglia. Risale poi al primo antenato con figli non visitati, e ripete la visita verso le foglie a partire da quei figli. L'algoritmo ha tre varianti. La prima, detta *preordine* o *ordine anticipato*, visita prima il nodo, poi i sottoalberi sinistro e destro. La seconda, detta *inordine* o *ordine simmetrico*, visita prima il sottoalbero sinistro, poi il nodo base, poi il sottoalbero destro. L'ultima, detta *postordine* o *ordine posticipato*, visita prima il sottoalbero sinistro, poi il destro, e infine il nodo.

```
// preordine iterativo
algoritmo visitaDFS(nodo r)
  Pila S
  S.push(r)
  while(not S.isEmpty()) do
    u <- S.pop()
    if(u != null) then
      visita il nodo u
      S.push(figlio destro di u)
      S.push(figlio sinistro di u)
```

```
// preordine ricorsivo
algoritmo visitaDFSRicorsiva(nodo r)
  if(r = null) then return
  visita il nodo r
  visitaDFSRicorsiva(figlio sinistro di r)
  visitaDFSRicorsiva(figlio destro di r)
```

Un algoritmo alternativo è la *visita in ampiezza* (BFS). Partendo dalla radice, l'algoritmo procede visitando i nodi per livelli successivi. Un nodo ad un dato livello è visitabile solo se tutti i nodi del livello superiore sono stati visitati.

```
// versione iterativa
algoritmo visitaBFS(nodo r)
  Coda C
  C.enqueue(r)
  while(not C.isEmpty()) do
    u <- C.dequeue()
    if(u != null) then
      visita il nodo u
      C.enqueue(figlio sinistro di u)
      C.enqueue(figlio destro di u)
```

Un *albero binario di ricerca* (*binary search tree*, BST) è un albero binario che soddisfa le seguenti proprietà:

1. ogni nodo v contiene un elemento $elem(v)$ cui è associata una *chiave*(v) presa da un dominio totalmente ordinato
2. le chiavi nel sottoalbero sinistro di v sono minori o uguali alla *chiave*(v)
3. le chiavi nel sottoalbero destro di v sono maggiori o uguali alla *chiave*(v)

Gli alberi binari di ricerca sono utilizzati per implementare con efficienza i *dizionari*, definiti come collezioni di coppie chiave/valore. In questo caso, si chiede che le chiavi da ordinare tramite l'albero siano univoche. In casi d'uso che non richiedano chiavi univoche, possono esserci varie altre regole per ordinare gli elementi.

Su un albero binario generico il costo della ricerca è $O(n)$ indipendentemente dalla strategia applicata, che sia in profondità o in ampiezza. Se l'albero è di ricerca, possiamo partire dalla radice e decidere in che verso muoverci sfruttando l'ordinamento delle chiavi. L'algoritmo di ricerca, formalmente, è:

```
algoritmo search(chiave k) -> elem
  v <- radice di T
  while (v != null) do
    if (k = chiave(v)) then return elem(v)
    else if (k < chiave(v)) then v <- figlio sinistro di v
    else v <- figlio destro di v
  return null
```

Il caso peggiore si verifica quando la chiave desiderata corrisponde ad una foglia a massima profondità, o quando essa è introvabile. In tal caso, il numero di passi è pari all'altezza dell'albero, e la complessità è $O(h)$, dove h è l'altezza dell'albero. Possiamo mettere in relazione l'altezza h dell'albero con il suo numero n di elementi, per ottenere una stima di complessità in funzione di quest'ultimo. Il caso peggiore corrisponde alla ricerca sull'albero degenerare linearizzato, nel quale ogni elemento ha un solo figlio. In questo caso l'albero degenera in una lista collegata con $h = n - 1$ elementi, e la ricerca ha dunque complessità $O(n)$ perché richiede la scansione lineare. L'altezza è $h = \Theta(n)$. Il caso migliore è un albero completo bilanciato, in cui ogni elemento ha entrambi i figli, tranne le foglie, e in cui i sottoalberi destro e sinistro di ogni elemento hanno lo stesso numero di membri. In tal caso, l'altezza dell'albero è $h = \Theta(\log(n))$.

Valendo l'ordinamento gerarchico delle chiavi, gli elementi di chiave minima e massima saranno rispettivamente il più a sinistra e il più a destra dell'albero. L'algoritmo per trovare il minimo a partire dalla radice deve procedere verso sinistra fino a una foglia. L'algoritmo per trovare il massimo a partire dalla radice deve procedere verso destra fino a una foglia.

```

algoritmo max(nodo u) -> nodo
  v <- u
  while(figlio destro di v != null) do
    v <- figlio destro di v
  return v

algoritmo min(nodo u) -> nodo
  v <- u
  while(figlio sinistro di v != null) do
    v <- figlio sinistro di v
  return v

```

Chiamiamo *predecessore* di un nodo u in un BST il nodo v nell'albero di chiave massima minore o uguale a $chiave(u)$. Chiamiamo *successore* di un nodo u in un BST il nodo v nell'albero di chiave minima maggiore o uguale a $chiave(u)$. Algoritmicamente, il successore è il minimo del sottoalbero destro, se esiste. Se il sottoalbero destro non esiste, si risale l'albero fino ad un antenato che sia figlio sinistro del proprio nodo padre. Tale nodo padre sarà l'elemento sul cui sottoalbero destro sarà possibile trovare il successore dell'elemento desiderato. Il predecessore è il massimo del sottoalbero sinistro, se esiste. Se il sottoalbero sinistro non esiste, si risale lungo l'albero fino a trovare un antenato che sia figlio destro del proprio nodo padre, e si cerca il predecessore dell'elemento desiderato nel sottoalbero sinistro dell'antenato. Questo algoritmo richiede ovviamente di salvare un riferimento al padre in ogni nodo dell'albero.

```

algoritmo pred(nodo u) -> nodo
  if (u ha un figlio sinistro sin(u)) then
    return max(sin(u))
  while (parent(u) != null e u è figlio sinistro di suo padre) do
    u <- parent(u)
  return parent(u)

```

L'inserimento in un albero binario di ricerca va effettuato mantenendo la proprietà di ricerca. L'operazione di inserimento è inizialmente simile a quella di ricerca, perché risulta necessario trovare il punto dove inserire ordinatamente il nuovo elemento. Questi è sempre aggiunto come foglia, nel modo meno invasivo possibile. Anche l'inserimento, come la ricerca, ha complessità $O(h)$. La visita in profondità in ordine simmetrico riflette l'ordinamento delle chiavi.

Cancellare mantenendo la proprietà di ricerca è più complesso. Se l'elemento da eliminare è una foglia, può essere rimosso direttamente senza bisogno di altre considerazioni. Se l'elemento da eliminare ha un solo figlio, attacchiamo il figlio dell'elemento da eliminare al padre dell'elemento da eliminare, al posto dell'eliminato. Se l'elemento da eliminare ha due figli, deve essere sostituito con il suo predecessore, e quest'ultimo va poi eliminato secondo le regole dei casi precedenti. Questa procedura funziona perché, per definizione, il predecessore non ha figli destri. In questo modo, la sua eliminazione ricade necessariamente in uno dei primi due casi. La procedura può essere effettuata anche utilizzando l'elemento successore al posto del predecessore.

L'efficienza $O(h)$, come spiegato in precedenza si traduce, in termini di n , in $O(n)$ nel caso peggiore per alberi molto sbilanciati, e in $O(\log(n))$ nel caso peggiore per alberi bilanciati. A tal fine, definiamo il *fattore di bilanciamento* $\beta(v) = |h(\text{sinistro}(v)) - h(\text{destro}(v))|$. Definiamo un albero *bilanciato in altezza* se $\forall v$ vale $\beta(v) \leq 1$, *completo* se $\forall v$ $\beta(v) = 0$ e *degenerato in lista* se $\beta(v) = h$. Esiste una categoria di alberi, chiamata AVL, che comprende alberi binari bilanciati in altezza, che si auto-aggiustano per rotazione. La rotazione consiste nello spostare la posizione gerarchica di nodi senza però alterare gli archi. Ad esempio, un nodo figlio con un solo figlio può essere promosso a nodo con due figli (il figlio originale e il padre) alzandolo di un livello, e abbassando il padre. La rotazione ha costo $O(1)$. L'inserimento con rotazione richiede di calcolare i fattori di bilanciamento lungo il percorso tra la radice e il nuovo nodo inserito. Successivamente, in caso di criticità solitamente riconoscibili da uno sbilanciamento sopra la soglia di 2, si effettua una singola rotazione nel verso opportuno. Strategie di bilanciamento alternative alla rotazione comprendono lo spostamento e la fusione. Gli alberi rosso-neri utilizzano entrambe queste strategie.