1. Alberi

L'albero ordinato (spesso chiamato più semplicemente "albero") è una struttura informativa fondamentale, che si presta a rappresentare svariate situazioni, quali:

- partizioni successive di un insieme in sottoinsiemi disgiunti;
- organizzazioni gerarchiche di dati;
- procedimenti enumerativi o decisionali.

Esempio (Classificazioni botaniche e zoologiche) Le classificazioni delle specie vegetali e animali usate nelle scienze naturali rappresentano un ben noto esempio di strutturazione gerarchica. La Figura 1.1 mostra una delle tante classificazioni del Regno Vegetale. ■

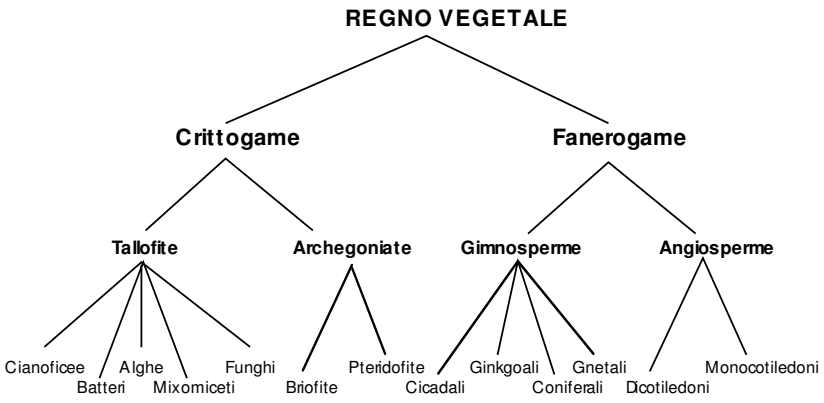

Figura 1.1: Classificazione del regno vegetale.

Esempio (Chiamate ricorsive) Le chiamate di un algoritmo ricorsivo possono essere rappresentate con una struttura ad albero. Ad esempio, le chiamate dell'algoritmo hanoi-ricorsiva() per $n = 3$ discusse nell'Esempio (2) possono essere raffigurate come in Figura 1.2. ■

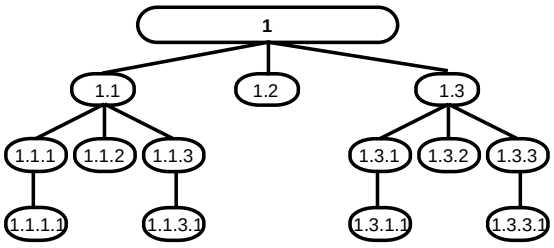

Figura 1.2: Struttura delle chiamate ricorsive di hanoi-ricorsiva() per $n = 3$.

La terminologia usata nel trattare gli alberi è uno strano misto di termini matematici, botanici e di parentela. Sia T un albero ordinato di n nodi di radice r. Siano $T_1, \ldots, T_k$ gli insiemi disgiunti non vuoti in cui sono partizionati i rimanenti $n-1$ nodi di T, aventi come radici, rispettivamente, i nodi $r_1, \ldots, r_k$. Ciascun T_i è detto *sottoalbero* (*subtree*, in inglese) di T radicato in r_i, mentre ciascun r_i è detto *figlio* (*child*) di r. I nodi $r_1, \ldots, r_k$ sono tra loro *fratelli* (*sibling*), ed r è il loro *padre* (*parent*). Un *discendente* di un nodo u è un figlio di u oppure un figlio di un discendente di u; analogamente, un *antenato* di u è suo padre oppure il padre di un suo antenato. Un nodo senza figli è detto *foglia* (*leaf*), mentre la *radice* dell'albero (*root*) è l'unico nodo senza padre. In pratica, in un albero ordinato è stabilito un doppio ordinamento tra i nodi. In primo luogo c'è un ordinamento "verticale" per *livelli*, cioè in base alla distanza dei nodi dalla radice. La radice r è a livello 0, tutti i figli di r stanno al livello 1, i figli dei figli di r stanno al livello 2, e così via fino all'esaurimento dei nodi. L'*altezza* di un albero è il massimo livello delle sue foglie. In secondo luogo, è stabilito un ordinamento "orizzontale" tra i nodi che stanno allo stesso livello, distinguendo così tra il primo figlio di un nodo (il figlio maggiore), il suo secondo figlio, e così via fino all'ultimo (il figlio minore).

Come visto negli esempi precedenti, i nodi si disegnano spesso con cerchietti, mentre coppie padre-figlio si visualizzano con linee che collegano coppie di cerchietti. La terminologia italiana pare di origine patriarcale (si usano i termini padre, figlio e fratello, anziché madre, figlia e sorella), mentre quella inglese è più "politicamente corretta". Inoltre gli alberi sono usualmente rappresentati rovesciati, con la radice in alto e le foglie in basso (si veda la Figura 1.3).

Esempio (Alberi ordinati) La Figura 1.4 mostra tre alberi ordinati T_1, T_2, e T_3. Pur essendo formati dagli stessi 5 nodi, i tre alberi sono diversi tra loro. Infatti, la radice di T_1 (il nodo 1) è diversa da quella di T_2 e T_3 (il nodo 2). Inoltre, considerando i nodi allo stesso livello ordinati da sinistra verso destra, T_2 e T_3 sono diversi tra loro. Infatti, pur avendo la stessa radice (il nodo 2), gli stessi figli della radice (i nodi 1, 3, 4), e lo stesso figlio del nodo 3 (il nodo 5), l'ordine dei figli di 2 non è lo stesso in T_2 e in T_3. ■

1.1 Specifica

L'albero ordinato può essere interpretato come una generalizzazione di una sequenza lineare, in cui ciascun elemento ha ancora un solo predecessore (il padre), ma in generale può avere più di un

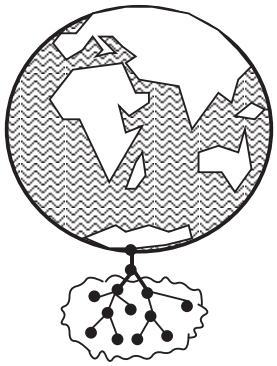

Figura 1.3: L'albero nel mondo informatico.

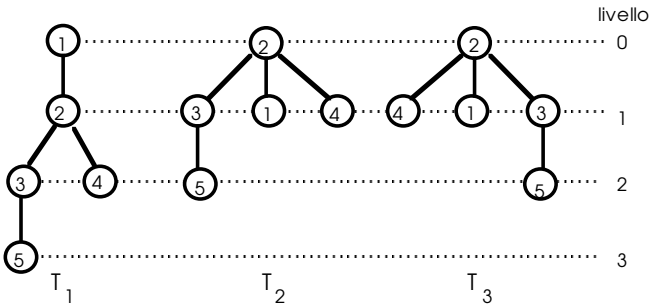

Figura 1.4: Alberi ordinati.

successore (i figli). Infatti, una sequenza è un particolare albero in cui ogni nodo ha al più un figlio e c'è una sola foglia.

Analogamente alle operazioni sulle sequenze, quelle definite sugli alberi ordinati prevedono la possibilità di interrogare un nodo per ottenere informazioni sui suoi parenti, quali il padre, i figli o i fratelli. A seconda della particolare utilizzazione della struttura di dati albero, alcune di queste "parentele" possono rivelarsi non necessarie; per lo stesso motivo, anche la modalità con cui l'insieme dei figli viene ispezionato può dipendere dall'applicazione. La specifica presentata in questa sezione è generica e permette di accedere a tutte le informazioni di parentela. Nel seguito, vedremo come versioni ridotte di questa specifica possono essere realizzate in maniera efficiente, occupando una quantità limitata di memoria.

Nella nostra specifica, i concetti di nodo e di radice di sottoalbero sono considerati sinonimi: le operazione che dovrebbero restituire dei nodi (come per esempio quelle che permettono di ispezionare i figli) restituiscono elementi di tipo TREE, che rappresenta la radice di un sottoalbero. La creazione di un nuovo albero corrisponde alla creazione di un singolo nodo isolato, senza padre né figli. Così come nelle liste, esistono operazioni quali read() e write() che permettono di leggere e scrivere il contenuto dei nodi.

TREE

```
% Costruisce un nuovo albero, costituito da un solo nodo e contenente v
Tree(item v)

% Legge il valore
item read()

% Scrive v nel nodo
write(item v)

% Restituisce il padre; nil se questo nodo è radice
TREE parent()

% Restituisce il primo figlio; nil se questo nodo è foglia
TREE leftmostChild()

% Restituisce il prossimo fratello del nodo a cui è applicato; nil se assente
TREE rightSibling()

% Inserisce il sottoalbero t come primo figlio di questo nodo
insertChild(TREE t)
   precondition: t.parent() = nil

% Inserisce il sottoalbero t come successivo fratello di questo nodo
insertSibling(TREE t)
   precondition: t.parent() = nil

% Distrugge il sottoalbero radicato nel primo figlio di questo nodo
deleteChild()

% Distrugge il sottoalbero radicato nel prossimo fratello di questo nodo
deleteSibling()
```

Sono presenti le operazioni parent(), leftmostChild() e rightSibling(), che restituiscono rispettivamente il padre, il primo figlio e il fratello successivo di un nodo. Le ultime due possono essere utilizzate per scorrere la lista dei figli. Se parent() restituisce **nil**, il nodo è radice; se leftmostChild() restituisce **nil**, il nodo è foglia (ovviamente, un nodo può essere radice e foglia contemporaneamente). Se rightSibling() restituisce **nil**, è stata ispezionata l'intera lista di figli.

La creazione di nuovi alberi sfrutta la ricorsività della definizione di questa struttura di dati; oltre al caso base, rappresentato dalla costruzione di alberi costituiti da singoli nodi isolati (descritta

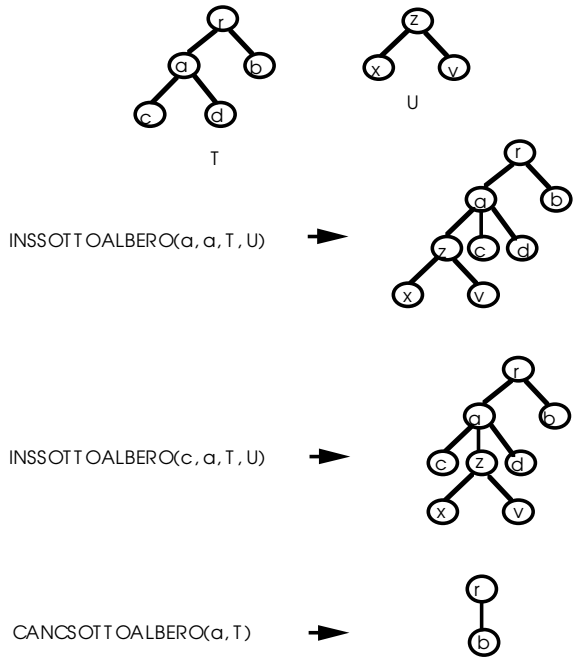

Figura 1.5: Inserzioni e cancellazioni di sottoalberi. **TODO**: "INSSOTTOALBERO(a,a,T,U)" $\rightarrow$ "a.insertChild(z)", "INSSOTTOALBERO(c,a,T,U)" $\rightarrow$ "c.insertSibling(z)", "CANCSOTTOALBERO(a,T)" $\rightarrow$ "r.deleteChild()"

in precedenza), è possibile "aggiungere" un sottoalbero come figlio (operazione insertChild()) o come fratello successivo (operazione insertSibling()) di un particolare nodo. La precondizione per applicare questi metodi con successo è che l'albero da aggiungere non sia già sottoalbero di un altro albero (parent() = **nil**). Infine, sono presenti le operazioni deleteChild() e deleteSibling() che distruggono il sottoalbero radicato nel primo figlio e nel fratello successivo del nodo su cui si applica l'operazione. Gli effetti delle operazioni insertChild(), insertSibling(), deleteChild() sono illustrati nella Figura 1.5.

1.2 Visite

La visita di un albero ordinato consiste nel seguire una "rotta" di viaggio che consenta di esaminare ogni nodo dell'albero esattamente una volta. La visita può essere eseguita in vari modi, detti "ordini", tra cui i quattro più importanti sono: ordine anticipato (o *previsita*), ordine posticipato (o *postvisita*), ordine simmetrico (o *invisita*), ordine per livelli. I primi tre metodi sono *visite in profondità*, perché si segue un percorso radice-foglia, mentre l'ultima è detta anche *visita in ampiezza*, perché vengono visitati i nodi livello per livello, "orizzontalmente", fino a raggiungere il livello massimo dell'albero.

Sia T un albero non vuoto di radice r. Se r non è una foglia ma ha $k > 0$ figli, indichiamo con $T_1, \ldots, T_k$ i k sottoalberi di T radicati nei k figli di r. Gli ordini di visita in profondità sono definiti ricorsivamente come segue:

- la *previsita* di T consiste nell'esaminare r e poi nell'effettuare, nell'ordine, la previsita di $T_1, \ldots, T_k$;
- la *postvisita* di T consiste nell'effettuare, nell'ordine, la postvisita di $T_1, \ldots, T_k$ e poi nell'esaminare r;

– Fissato $i \geq 1$, la *invisita* di T consiste nell'effettuare la invisita di $T_1, \ldots, T_i$, nell'esaminare r, e poi nell'effettuare la invisita di $T_{i+1}, \ldots, T_k$.

Esempio (Visita di alberi ordinati) Gli ordini di visita dell'albero della Figura 1.1 sono:

Previsita: Regno vegetale, Crittogame, Tallofite, Cianoficee, Batteri, Alghe, Mixomiceti, Funghi, Archegoniate, Briofite, Pteridofite, Fanerogame, Gimnosperme, Cicadali, Ginkgoali, Coniferali, Gnetali, Angiosperme, Dicotiledoni, Monocotiledoni;

Postvisita: Cianoficee, Batteri, Alghe, Mixomiceti, Funghi, Tallofite, Briofite, Pteridofite, Archegoniate, Crittogame, Cicadali, Ginkgoali, Coniferali, Gnetali, Gimnosperme, Dicotiledoni, Monocotiledoni, Angiosperme, Fanerogame, Regno vegetale;

Invisita ($i = 1$): Cianoficee, Tallofite, Batteri, Alghe, Mixomiceti, Funghi, Crittogame, Briofite, Archegoniate, Pteridofite, Regno vegetale, Cicadali, Gimnosperme, Ginkgoali, Coniferali, Gnetali, Fanerogame, Dicotiledoni, Angiosperme, Monocotiledoni;

Per livelli: Regno vegetale, Crittogame, Fanerogame, Taliofite, Archegoniate, Gimnosperme, Angiosperme, Cianoficee, Batteri, Alghe, Mixomiceti, Funghi, Briofite, Pteridofite, Cicadali, Ginkgoali, Coniferali, Gnetali, Dicotiledoni, Monocotiledoni. ■

In pratica, l'ordine di visita anticipato coincide con quello del diritto di successione al trono nell'albero genealogico di una dinastia regnante. Per illustrare la differenza tra gli ordini di visita anticipato e posticipato, c'è un modo pittoresco che non tira in ballo la ricorsione. Si immagini che l'albero sia una sorta di isola fatta a fiordo: i cerchietti rappresentano fari e le linee che collegano coppie di fari rappresentano alte scogliere. Una nave fa la circumnavigazione dell'isola, partendo dalla radice e costeggiando sempre le scogliere. Il capitano segna sul libro di bordo il nome di ciascun faro che viene incontrato nella navigazione, ma solo quando lo incontra per la prima volta. È facile verificare che se la circumnavigazione procede in senso antiorario, allora si ottiene proprio l'ordine di visita anticipato, mentre se procede in senso orario si ottiene l'ordine di visita posticipato inverso (cioè letto da destra verso sinistra, ovvero dall'ultimo nodo visitato al primo). Lo schema di procedura dfs() serve per effettuare sia la visita anticipata che quella posticipata.

```
dfs(TREE t)
    precondition: t ≠ nil
(1) esame "anticipato" del nodo radice di t
    TREE u = t.leftmostChild()
    while u ≠ nil do
        dfs(u)
        u = u.rightSibling()
(2) esame "posticipato" del nodo radice di t
```

Se l'esame del nodo t viene effettuato solo nella riga (1), ignorando la riga (2), si ottiene la procedura pre-visit(), mentre se viene effettuato nella riga (2), ignorando la riga (1), si ottiene la procedura post-visit().

La invisita richiede codice leggermente più complesso; quando in-visit() viene eseguito sul nodo t, la stessa procedura viene richiamata ricorsivamente sui primi i figli di t, se presenti; viene effettuata la visita simmetrica di t, e quindi in-visit() viene richiamata sui restanti figli.

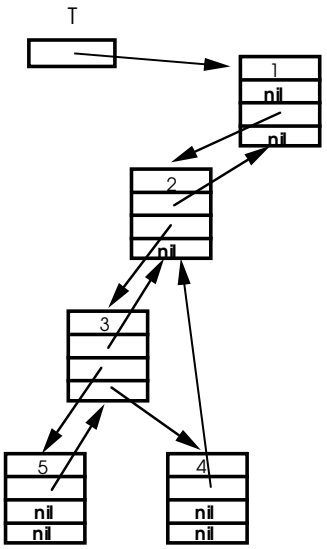

Figura 1.6: Realizzazione con puntatori padre/primo-figlio/fratello dell'albero T_1 di Figura 1.4

```
in-visit(TREE t, int i)
    precondition: t ≠ nil
    TREE u = t.leftmostChild()
    int k = 0
    while u ≠ nil and k < i do
        k = k + 1
        in-visit(u, i)
        u = u.rightSibling()
    esame "simmetrico" del nodo t
    while u ≠ nil do
        in-visit(u)
        u = u.rightSibling()
```

```
bfs(TREE t)
    precondition: t ≠ nil
    QUEUE Q = Queue()
    Q.enqueue(t)
    while not Q.isEmpty() do
        TREE u = Q.dequeue()
        esame "per livelli" del nodo
          u
        u = u.leftmostChild()
        while u ≠ nil do
            Q.enqueue(u)
            u = u.rightSibling()
```

La visita per livelli illustrata nell'algoritmo bfs() utilizza un approccio completamente diverso. Invece di essere basata su una visita ricorsiva (che implicitamente utilizza una pila), è una procedura iterativa basata su una coda. All'inizio, si inserisce la radice nella coda. Quando un nodo viene estratto, vengono inseriti tutti i suoi figli nella coda, in ordine. È facile dimostrare per induzione che tutti i nodi del livello i-esimo vengono esaminati prima dei nodi del livello $(i+1)$-esimo.

Nelle tre procedure di visita in profondità descritte viene effettuata esattamente una chiamata ricorsiva per ciascun nodo, per un totale di n chiamate, dove n è il numero di nodi dell'albero. Nella visita in ampiezza, ogni nodo è inserito in coda esattamente una volta. Pertanto, la complessità delle quattro procedure di visita è $\Theta(n)$, purché si fornisca una realizzazione in cui le operazioni utilizzate nelle procedure di visita richiedano tempo $O(1)$.

1.3 Realizzazione con puntatori padre/primo-figlio/fratello

La realizzazione più naturale, detta a puntatori *padre/primo-figlio/fratello*, segue fedelmente la specifica, memorizzando per ogni nodo il valore, il puntatore al padre, il puntatore al primo figlio e il puntatore al fratello successivo. La Figura 1.6 illustra un albero realizzato in questo modo. I

nodi sono memorizzati direttamente in strutture di tipo TREE, che vengono create come nodi isolati contenenti un singolo valore.

Le operazioni per spostarsi lungo l'albero (parent(), leftmostChild() e rightSibling()), così come le operazioni di lettura e scrittura del valore del nodo (read() e write()) vengono realizzate in tempo $O(1)$, perché semplicemente leggono o scrivono il valore corrispondente. Non vengono mostrate per via della loro semplicità.

Più interessanti sono le operazioni che realizzano le funzionalità di modifica dell'albero. Le operazioni di inserimento di alberi esistenti sono simili alle corrispondenti operazioni nelle liste; insertChild() inserisce un elemento in testa alla lista dei figli, mentre insertSibling() lo inserisce dopo un particolare figlio. In entrambi i casi, il costo è $O(1)$, in quanto si tratta semplicemente di "agganciare" l'albero t nella posizione selezionata. Le operazioni di cancellazione, invece, hanno un costo $O(n)$ perché operano ricorsivamente sull'albero, usando la procedura ricorsiva delete().

1.4 Realizzazione con vettore dei padri

In certi casi, non sono richieste tutte le operazioni precedentemente introdotte. Ad esempio, in alcuni problemi sui grafi l'output è rappresentato da un albero, che viene costruito e navigato "al contrario", seguendo i puntatori ai padri. In questo caso, i riferimenti a figli e fratelli sono inutili, e l'albero può essere rappresentato tramite un *vettore dei padri*, con un consumo minimo di memoria.

Si supponga che i nodi dell'albero T siano numerati da 1 ad n. Il vettore dei padri p contiene, per ogni nodo i, $1 \leq i \leq n$, l'indice del padre: $p[i] = j$, se j è il padre di i; $p[i] = 0$, se i è la radice.

Esempio (Vettore dei padri) Il contenuto del vettore dei padri per l'albero T_1 della Figura 1.4 è: $p[1] = 0$, $p[2] = 1$, $p[3] = 2$, $p[4] = 2$, $p[5] = 3$. ■

In questa realizzazione, l'albero non è ordinato: non è infatti chiara la relazione tra fratelli. Normalmente, questo non è un problema; altrimenti, è possibile assumere una relazione implicita, per esempio imponendo che un figlio abbia sempre un numero maggiore del padre e che nodi fratelli siano numerati in modo crescente da sinistra verso destra (ciò implica che la radice è nella prima posizione del vettore).

Ovviamente la specifica deve essere adattata, tenendo conto del fatto che la struttura albero e il nodo non coincidono più, eliminando le operazioni inutili e/o ridondanti e aggiungendo l'operazione setParent(), che "aggiunge" un sottoalbero ad un nodo esistente. Mostriamo qui di seguito una semplice realizzazione, tralasciando per semplicità di memorizzare i valori dei nodi.

TREE

```
int[] p                                          % Vettore dei padri

% Costruisce una "foresta" con n nodi isolati
Tree(int n)
    int[]p = new int[1...n]
    for i = 0 to n − 1 do  p[i] = 0

% Restituisce il padre del nodo i; restituisce 0 se i è radice
int parent(int i)
    return p[i]

% Rende il nodo i un figlio del nodo j
setParent(int i, int j)
    p[i] = j
```

```
TREE
    NODE parent                                   % Puntatore al padre
    NODE child                                    % Puntatore al primo figlio
    NODE sibling                                  % Puntatore al successivo fratello
    item value                                    % Valore del nodo

    Tree(item v)                                  % Crea un nuovo nodo
        TREE t = new TREE
        t.value = v
        t.parent = t.child = t.sibling = nil
        return t

    insertChild(TREE t)
        t.parent = this
        t.sibling = child                         % Inserisce t prima dell'attuale primo figlio
        child = t

    insertSibling(TREE t)
        t.parent = parent
        t.sibling = sibling                       % Inserisce t prima dell'attuale fratello
        sibling = t

    deleteChild()
        NODE newChild = child.rightSibling()
        delete(child)
        child = newChild

    deleteSibling()
        NODE newBrother = sibling.rightSibling()
        delete(sibling)
        sibling = newBrother

    delete(TREE t)
        NODE u = t.leftmostChild()
        while u ≠ nil do
            TREE next = u.rightSibling()
            delete(u)
            u = next
        delete t
```

1.5 Alberi binari

Un albero binario è un particolare albero ordinato in cui ogni nodo ha al più due figli e si fa distinzione tra il figlio *sinistro* ed il figlio *destro* di un nodo. Questa proprietà è assai sottile, poiché impone che due alberi T e U aventi gli stessi nodi, gli stessi figli per ogni nodo e la stessa radice, siano distinti qualora un nodo u sia designato come figlio sinistro di un nodo v in T e come figlio destro del medesimo nodo in U.

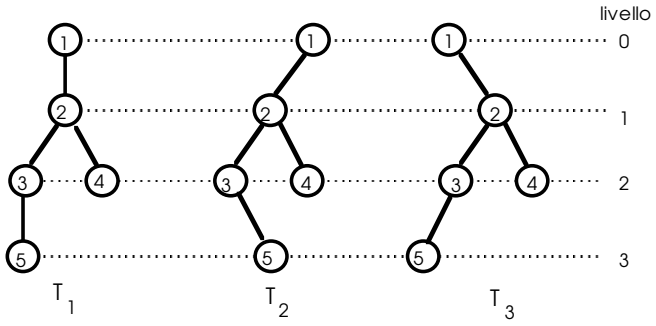

Figura 1.7: Alberi ordinati e alberi binari.

Esempio (Alberi binari) La Figura 1.7 mostra tre alberi $T_1, \ldots, T_3$. Visti come alberi ordinati, essi sono tutti identici tra loro. T_1 non è binario, perché non è chiara la convenzione per distinguere i figli sinistri dai destri. Considerando i figli disegnati a sinistra (destra) come figli sinistri (destri), allora T_2 e T_3 sono binari; inoltre essi sono tra loro distinti, poiché non hanno gli stessi figli destri e sinistri. ■

L'importanza degli alberi binari risiede sia sulla semplicità della loro struttura che sulle innumerevoli applicazioni in cui possono essere impiegati.

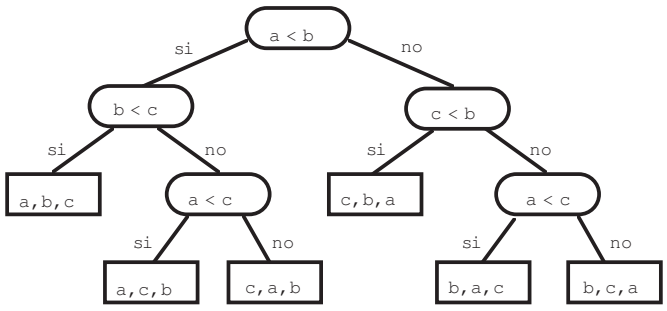

Figura 1.8: Albero delle scelte per l'ordinamento di tre numeri.

Esempio (Albero delle scelte) Le possibili sequenze di confronti (a due alternative) effettuate da un algoritmo per risolvere un problema possono essere visualizzate con un albero binario, detto albero delle scelte. In tale albero, i nodi con figli rappresentano confronti tra dati del problema, le coppie padri-figli i risultati dei confronti, e le foglie soluzioni possibili del problema. Un percorso radice-foglia rappresenta la sequenza di confronti effettuati per individuare una soluzione e il livello massimo di una foglia dà il numero di confronti effettuati nel caso pessimo.

Un albero delle scelte che minimizzi il livello massimo delle foglie fornisce con tale valore una limitazione inferiore al numero di decisioni che ogni algoritmo che risolva il problema deve effettuare nel caso peggiore.

La Figura 1.8 illustra un albero delle scelte per il problema di ordinare tre numeri a, b, c. Poiché un qualsiasi algoritmo di ordinamento di n numeri deve fare una sequenza di scelte che permetta di individuare la soluzione corretta tra le $n!$ permutazioni possibili dei dati di ingresso, l'albero delle scelte per un qualsiasi algoritmo di ordinamento ha almeno $n!$ foglie. Poiché ad ogni livello dell'albero il numero di nodi al più raddoppia, la lunghezza del percorso radice-foglia più lungo non può essere inferiore a $\log_2 n!$ Essendo $n! = n(n-1)\cdots 3\cdot 2\cdot 1 \geq n(n-1)\cdots(n/2) \geq (n/2)^{n/2}$, si ha $\log_2 n! \geq (n/2)\log_2(n/2)$. Pertanto una *limitazione inferiore* alla complessità del problema dell'ordinamento per algoritmi basati su confronti è $\Omega(n\log n)$. ■

Una possibile specifica per gli alberi binari include le operazioni per leggere e scrivere i figli destro e sinistro, che sostituiscono le operazioni per leggere e scrivere il primo figlio e i fratelli successivi.

TREE

```
% Restituisce il figlio sinistro (destro) di questo nodo; restituisce nil se assente
TREE left()
TREE right()

% Inserisce il sottoalbero t come figlio sinistro (destro) di questo nodo
insertLeft(TREE t)
    precondition: t.parent = nil
insertRight(TREE t)
    precondition: t.parent = nil

% Distrugge il sottoalbero sinistro (destro) di questo nodo
deleteLeft()
deleteRight()
```

Una realizzazione ragionevole fa uso di puntatori, come nel caso degli alberi ordinati, i cui campi *child* e *sibling* sono sostituiti da *left* e *right*. La complessità di tutte le operazioni è $O(1)$, con esclusione di quella di cancellazione, che agendo ricorsivamente può avere complessità $O(n)$. Per motivi di spazio, le operazioni parent(), left(), right(), read() e write() non sono mostrate; semplicemente, restituiscono il valore della variabile corrispondente.

TREE

```
Tree(item v)
    TREE t = new TREE
    t.parent = nil
    t.left = t.right = nil
    t.value = v
    return t

insertLeft(TREE T)
    T.parent = this
    left = T

insertRight(TREE T)
    T.parent = this
    right = T

deleteLeft()
    if left ≠ nil then
        left.deleteLeft()
        left.deleteRight()
        delete left
        left = nil

deleteRight()
    if right ≠ nil then
        right.deleteLeft()
        right.deleteRight()
        delete right
        right = nil
```

La sostanziale equivalenza di queste due realizzazioni non deve sorprendere, poiché nel proporre

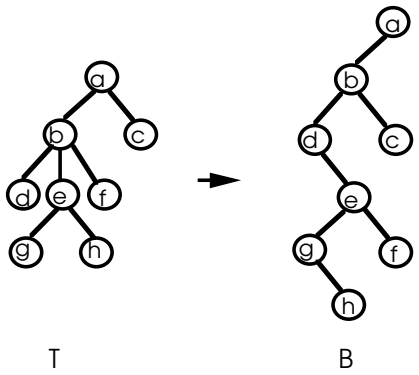

Figura 1.9: Rappresentazione di un albero ordinato T con un albero binario B.

quella per gli alberi ordinati si è implicitamente trasformato l'albero ordinato in uno binario! Come illustrato nella Figura 1.9, è sempre possibile rappresentare un albero ordinato T con un albero binario B avente gli stessi nodi e la stessa radice, in cui ogni nodo di B ha come figlio sinistro il primo figlio che ha in T e come figlio destro il fratello successivo che ha in T. È facile verificare che le sequenze dei nodi esaminati su T e su B coincidono se T e B sono entrambi visitati in ordine anticipato, oppure se T è visitato in ordine posticipato e B è visitato in ordine simmetrico.

Utilizzando le operazioni introdotte, la visita di un albero binario viene effettuata molto semplicemente da dfs(). La *previsita* si ottiene esaminando il nodo t soltanto nella riga (1) e ignorando le righe (2) e (3). Analogamente, la *invisita* o la *postvisita* si hanno esaminando il nodo t solo nella riga (2) o in (3), rispettivamente.

```
dfs(TREE t)
    if t ≠ nil then
(1)     esame "anticipato" del nodo radice di t
        dfs(t.left())
(2)     esame "simmetrico" del nodo radice di t
        dfs(t.right())
(3)     esame "posticipato" del nodo radice di t
```

1.6 Altre realizzazioni

Se gli alberi sono dotati di caratteristiche particolari, è possibile progettare realizzazioni alternative per essi. Per esempio, un *albero d-ario completo* è un albero in cui ogni nodo interno ha esattamente d figli. Tale albero può essere memorizzato in un vettore, inserendo la radice nell'indice 1, mentre i figli di un nodo i sono memorizzati negli indici $d \cdot (i-1)+2$, $d \cdot (i-1)+3$, …, $d \cdot i+1$. Vedremo un esempio di questo metodo di memorizzazione quando introdurremo gli alberi binari heap [cfr. Capitolo 10.1]. Alternativamente, se il numero massimo di figli di un nodo è limitato superiormente da un intero d, è possibile sostituire la lista primo figlio / fratello con un vettore di puntatori di dimensione d; gli eventuali figli mancanti sono sostituiti con puntatori **nil**.

1.7 Reality check

L'elenco delle applicazioni degli alberi è così vasto che è difficile sceglierne una. La struttura base di un file system, ad esempio, è un albero; le directory sono nodi interni, mentre i file sono foglie. Tuttavia, poiché alcuni file system permettono di elencare lo stesso file in più directory (tramite

hard link o symbolic link), sarebbe più corretto descrivere tali file system come grafi diretti aciclici [cfr. Capitolo 9.6].

1.8 Esercizi

Esercizio 1.1 (Altezza)**.** L'altezza di un albero ordinato è il massimo livello delle sue foglie. Si fornisca una funzione che calcoli in tempo ottimo l'altezza di un albero ordinato T di n nodi.

Esercizio 1.2 (Cancella le foglie con somma k)**.** Dato un albero ordinato i cui nodi contengono valori interi, se ne vogliono cancellare tutte le foglie per le quali il percorso radice-foglia ha somma complessiva dei valori uguale a k. Fornire una procedura di complessità ottima.

Esercizio 1.3 (Larghezza)**.** La larghezza di un albero ordinato è il numero massimo di nodi che stanno tutti al medesimo livello. Si fornisca una funzione che calcoli in tempo ottimo la larghezza di un albero ordinato T di n nodi.

Esercizio 1.4 (Visite)**.** Gli ordini di visita di un albero binario di 9 nodi sono i seguenti: A, E, B, F, G, C, D, I, H (anticipato), B, G, C, F, E, H, I, D, A (posticipato) e B, E, G, F, C, A, D, H, I (simmetrico). Si ricostruisca l'albero binario.

Esercizio 1.5 (Numero nodi per sottoalbero)**.** Dato un albero binario t non vuoto, si vuole memorizzare, in ciascun nodo u, il numero di nodi che si trovano nel sottoalbero radicato in u. Si fornisca un algoritmo lineare nel numero di nodi.

Esercizio 1.6 (Cancella foglie)**.** Dato un albero binario i cui nodi contengono interi, si vuole cancellare ogni foglia che sia un figlio sinistro e contenga lo stesso intero del padre. Si scriva una procedura ricorsiva di complessità ottima.

Esercizio 1.7 (Inserisci foglie)**.** Dato un albero binario, si vuole aggiungere ad ogni foglia un figlio sinistro contenente il valore 0. Si scriva una procedura ricorsiva di complessità ottima.

Esercizio 1.8 (Raggruppa le foglie con 0 e 1)**.** Dato un albero binario le cui foglie contengono 0 od 1 e i cui nodi interni contengono solo 0, si vuole cambiare il contenuto delle foglie in modo che, visitandole da sinistra verso destra, si incontrino prima tutti gli 0 e poi tutti gli 1. Si scriva una procedura ricorsiva di complessità ottima.

Esercizio 1.9. Dato un albero binario, i cui nodi contengono elementi interi, si scriva una procedura di complessità ottima per ottenere l'albero inverso, ovvero un albero in cui il figlio destro (con relativo sottoalbero) è scambiato con il figlio sinistro (con relativo sottoalbero).

Esercizio 1.10. Dato un albero binario i cui nodi contengono interi, si vuole aggiungere ad ogni foglia un figlio contenente la somma dei valori che appaiono nel cammino dalla radice a tale foglia. Si scriva una procedura ricorsiva di complessità ottima.

Esercizio 1.11 (Numero di foglie)**.** Si dimostri per induzione che in un albero binario il numero delle foglie è uguale al numero dei nodi con due figli più uno.

Esercizio 1.12 (Albero binario completo)**.** Un albero binario completo di altezza k è un albero binario in cui tutti i nodi, tranne le foglie, hanno esattamente due figli, e tutte le foglie si trovano al livello k. Si dimostri per induzione che, in un albero binario completo di altezza k, il numero dei nodi è $2^{k+1} - 1$ ed il numero delle foglie è 2^k.

Esercizio 1.13 (Visite iterative)**.** Utilizzando le pile, si scrivano tre procedure iterative di complessità ottima per effettuare, rispettivamente, le visite anticipata, differita e simmetrica di un albero binario.

1.9 Soluzioni

Soluzione Esercizio 1.1

Poiché la limitazione inferiore è $\Omega(n)$ per la dimensione dei dati, la funzione deve richiedere tempo $O(n)$. Si utilizza lo schema della visita differita nella quale si calcola l'altezza di ogni sottoalbero di T. L'altezza di un sottoalbero radicato in un generico nodo u è zero, se u è una foglia, mentre si ottiene sommando uno all'altezza massima dei sottoalberi radicati nei figli di u, altrimenti. Il tempo richiesto è $\Theta(n)$ e quindi ottimo. La chiamata della funzione è depth(T).

```
int depth(TREE t)
    int max = 0
    TREE u = t.leftmostChild()
    while u ≠ nil do
        int t = depth(u) + 1
        if t > max then
            max = t
        u = u.rightSibling()
    return max
```

Soluzione Esercizio 1.2

Una limitazione inferiore alla complessità del problema è $\Omega(n)$ poiché è necessario esaminare tutte le foglie dell'albero. L'algoritmo risolutivo si basa su una visita anticipata. Durante la discesa calcola per ogni nodo u la somma dei valori contenuti nei nodi incontrati a partire dalla radice fino ad t. Quando giunge ad una foglia, la cancella se tale somma è uguale a k. Si assume una realizzazione con puntatori padre/primo figlio/fratello. Per cancellare la foglia usa le operazioni deleteChild() e deleteSibling(), che richiedeno tempo $O(1)$ per cancellare un sottoalbero di un solo nodo. La complessità della procedura è $\Theta(n)$ e la chiamata iniziale è leafK($T,k,0$). Se la chiamata iniziale restituisce **true**, l'intero albero (costituito da una sola foglia di valore k) deve essere cancellato dal chiamante.

```
boolean leafK(TREE t, int k, int somma)
    somma = somma + t.read()
    if somma = k and t.leftmostChild() = nil then
        return true                          % Foglia con valore k; deve essere cancellata
    else
        while t.leftmostChild() ≠ nil and leafK(t.leftmostChild(), k, somma) do
            t.deleteChild()
        TREE u = t.leftmostChild()           % Nodo da cui cancellare
        TREE v = iif(u ≠ nil, u.rightSibling(), nil)   % Potenziale nodo da cancellare
        while v ≠ nil do
            if leafK(v, k, somma) then
                u.deleteSibling()            % u rimane fisso e un fratello viene eliminato
            else
                u = v                        % u avanza
            v = u.rightSibling()             % Prossimo fratello
        return false
```

Soluzione Esercizio 1.5

Si procede con una postvisita, memorizzando in ciascun nodo u il valore 1, se u è una foglia, o 1 più la somma dei valori contenuti nel figlio sinistro e nel figlio destro, altrimenti.

```
int count(TREE t)
    if t = nil then
        return 0
    else
        int somma = 1 + count(t.left()) + count(t.right())
        t.write(somma)
        return somma
```

Soluzione Esercizio 1.6

Tutte le foglie e quindi tutti i nodi dell'albero devono essere visitati nel caso pessimo. L'algoritmo proposto è basato su una previsita e ha complessità lineare nel numero di nodi e quindi ottima. Quando un nodo t è visitato, si verifica se ha un figlio sinistro ($u \neq$ **nil**), se è foglia (u.left() = u.right() = **nil**) e se ha lo stesso valore del padre (t.read() = u.read()), nel qual caso lo si cancella.

```
deleteLeaf(TREE t)
    if t ≠ nil then
        TREE u = t.left()
        if u ≠ nil and u.left() = u.right() = nil and t.read() = u.read() then
            t.deleteLeft()
        deleteLeaf(t.left())
        deleteLeaf(t.right())
```

Soluzione Esercizio 1.7

Valgono le stesse considerazioni dell'esercizio precedente riguardo alla limitazione inferiore e all'uso dei puntatori, ma stavolta l'algoritmo proposto è basato su una postvisita.

```
insertLeaf(TREE t)
    if t ≠ nil then
        insertLeaf(t.left())
        insertLeaf(t.right())
        if t.left() = t.right() = nil then
            TREE new = Tree(0)
            t.insertLeft(new)
```

Soluzione Esercizio 1.11

Sia $T(n)$ un generico albero binario con $n \geq 1$ nodi e si indichi con $n_2(T(n))$ il numero di nodi con due figli di $T(n)$ e con $F(T(n))$ il numero di foglie di $T(n)$. Si vuole dimostrare per induzione che $F(T(n)) = n_2(T(n)) + 1$.

La base dell'induzione è vera per $n = 1$, perché in tal caso $F(T(1)) = 1$ e $n_2(T(1)) = 0$. Si assuma la proprietà vera per $T(n)$. $T(n+1)$ può essere ottenuto da $T(n)$ aggiungendo un figlio o ad una foglia, o ad un nodo con un figlio. Nel primo caso, sia il numero delle foglie sia quello dei nodi con due figli non cambiano e quindi: $F(T(n+1)) = F(T(n)) = n_2(T(n)) + 1 = n_2(T(n+1)) + 1$. Nel secondo caso, sia il numero dei nodi con due figli sia quello delle foglie aumenta di uno e quindi: $F(T(n+1)) = F(T(n)) + 1 = n_2(T(n)) + 1 + 1 = n_2(T(n+1)) + 1$.