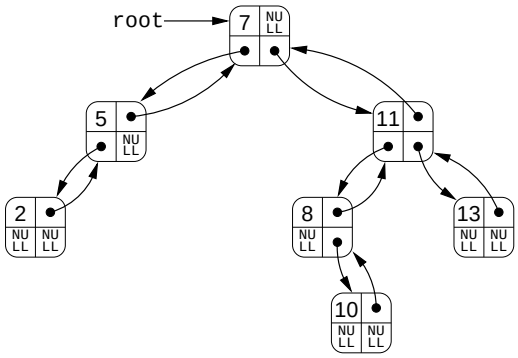

Figura 16.11 Esempio di albero binario: ogni nodo contiene il dato da immagazzinare e tre puntatori che definiscono le sue relazioni di parentela. I dati sono stati inseriti nei nodi in modo da soddisfare la proprietà fondamentale di un albero binario di ricerca.

Vediamo anche come rendere tale ricerca un'operazione efficiente, ossia tale da richiede al piú $\mathcal{O}(\log N)$ operazioni elementari per un insieme di N dati.

16.3.1 Alberi binari di ricerca

Nel Paragrafo 5.2.2 abbiamo visto come funziona l'algoritmo della ricerca binaria. Sebbene la ricerca binaria su insiemi di dati già ordinati sia efficiente, si tratta di un problema di scarso interesse perché nella pratica il problema di fondo è proprio quello di disporre di una struttura ordinata. Di solito il problema è quello di dover mantenere un insieme dinamico di dati in una struttura in cui si possano aggiungere e togliere elementi; questa struttura deve essere sufficientemente ordinata da permettere la ricerca efficiente di un elemento.

Esistono molte strutture di dati che permettono di immagazzinare dati in modo tale che la ricerca di un elemento sia un'operazione efficiente. La maggior parte di queste strutture ha una topologia ad albero. Come abbiamo in parte scoperto nel Capitolo 13 e come vediamo di nuovo qui, il costo per la gestione di una struttura di dati ad albero è al piú proporzionale alla profondità massima dell'albero. Per questo gli algoritmi che sono stati inventati nel corso degli ultimi 50 anni puntano principalmente a mantenere l'albero piú bilanciato possibile con il numero minore di operazioni. Per motivi di spazio e concisione, preferiamo concentrarci sulla struttura piú semplice, l'*albero binario*. Rimandiamo a testi piú completi [9] per la trattazione di algoritmi piú complicati, come gli alberi *red-black* o gli heap di Fibonacci.

In un albero binario, come quello nella Figura 16.11, ogni nodo possiede (oltre al dato che contiene) tre puntatori: il primo punta al nodo genitore e i seguenti due ai nodi figli, se esistono. Per convenzione, assegniamo il valore `NULL` ai puntatori che dovrebbero puntare a nodi che al momento non sono presenti nell'albero (ad esempio, il puntatore

al nodo genitore della radice e quelli ai nodi figli delle foglie). In considerazione del fatto che il nodo che sta alla radice dell'albero potrebbe cambiare durante le operazioni di aggiornamento dell'albero, conviene mantenere anche una variable `root` che punta al nodo che funge da radice. La struttura del nodo potrebbe essere ad esempio la seguente:

```
struct node {
  dataType datum;
  struct node *up, *left, *right;
};
```

dove `datum` è il dato immagazzinato nel nodo, `up` punta al nodo genitore, mentre `left` e `right` ai due nodi figli. Come al solito preferiamo definire un nuovo tipo `myNode` con il comando

```
typedef struct node myNode;
```

al fine di rendere piú conciso il programma.

In uno heap i nodi della struttura hanno una posizione ben precisa nel vettore degli elementi e quindi anche nella memoria fisica del computer; le relazioni di parentela sono determinate proprio dalla loro posizione nel vettore. In un generico albero binario, invece, l'ordine con cui i nodi sono scritti nella memoria è irrilevante: le loro relazioni di parentela sono esplicitate dai puntatori che ogni nodo possiede (per analogia considerate le liste concatenate illustrate nella Figura 16.4). Ad esempio, volendo riservare memoria per nuovi nodi dell'albero, non è necessario che tale memoria sia contigua a quella già in uso. Tuttavia è importante chiarire che una volta che un nodo è stato salvato in una certa locazione di memoria non può piú essere mosso, perché i puntatori che fanno riferimento al quel nodo usano l'indirizzo della locazione di memoria. Quindi sconsigliamo vivamente di usare il comando di riassegnazione dinamica della memoria (`realloc`) quando la struttura di dati è basata sull'uso dei puntatori.

Affinché la ricerca di un elemento in un albero binario sia un'operazione efficiente è necessario che gli elementi siano ordinati al suo interno rispettando la *proprietà fondamentale di un albero binario di ricerca*: se il dato x è conservato nel nodo `n`, allora tutti i dati conservati nel sottoalbero puntato[1] da `n.left` sono minori o uguali a x, mentre tutti quelli nel sottoalbero puntato da `n.right` sono maggiori o uguali a x.

Come nel caso della struttra di tipo heap, anche questa proprietà di ordinamento permette di conservare lo stesso insieme di dati in alberi diversi; per un esempio guardate la Figura 16.12. I differenti alberi si ottengono perché gli elementi possono essere stati inseriti o cancellati dalla struttura in ordine diverso.

Osserviamo, però, che in un albero binario, al contrario di ciò che avviene in uno heap, c'è una correlazione molto forte tra il valore (o la priorità) dei dati e la loro posizione nell'albero, infatti in ambedue gli alberi nella Figura 16.12 si vede che il valore dei dati è crescente se i nodi si leggono da sinistra verso destra (senza tenere conto della

[1] Con un leggero abuso di notazione intendiamo che il sottoalbero puntato da `p` è quello il cui nodo radice è puntato da `p`.

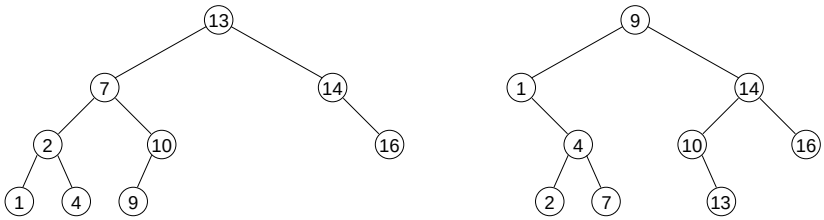

Figura 16.12 Esempi di alberi binari di ricerca: lo stesso insieme di dati può essere immagazzinato in forme diverse. Al contrario di ciò che avviene in uno heap (cfr. Figura 16.6), qui la topologia dell'albero identifica l'ordinamento dei dati, i cui valori crescono se letti da sinistra verso destra.

loro profondità nell'albero). Questa correlazione tra i valori dei dati e la loro posizione nell'albero è ciò che rende efficiente la ricerca di un elemento in un albero binario.

Qui di seguito mostriamo in una forma schematica le principali operazioni eseguibili su un albero binario di ricerca. Tenete presente che tutte le operazioni descritte richiedono al piú un tempo proporzionale alla profondità massima dell'albero $h_{\max}$ (lasciamo al lettore la verifica di questa affermazione), la quale tipicamente è dell'ordine di $\mathcal{O}(\log N)$. Quindi nel caso medio si tratta di operazioni efficienti.

Noterete, inoltre, che nessuna delle funzioni che descriviamo a breve riserva la memoria per i nodi dell'albero; assumiamo infatti che la "creazione" del nodo, ossia l'assegnazione dello spazio nella memoria, il riempimento dei suoi campi di dati e l'assegnazione del valore `NULL` ai puntatori, sia eseguita prima di chiamare una qualsiasi delle funzioni che stiamo per descrivere. Per questo motivo (e per motivi di completezza) preferiamo descrivere nuovamente anche quelle funzioni che abbiamo già introdotto nel Capitolo 13.

16.3.2 Inserimento e ricerca in un albero binario di ricerca

L'*inserimento* di un nuovo dato nell'albero è un'operazione relativamente semplice. Confrontate il nuovo elemento con la radice dell'albero e capite in quale sottoalbero deve essere posizionato. Quindi ripetete la stessa operazione nel sottoalbero appena scelto, fino a che non trovate un sottoalbero vuoto, ossia lo spazio per posizionare il nuovo elemento. La funzione ricorsiva nel Listato 16.5 esegue l'inserimento di un nuovo nodo `*new`, i cui puntatori sono già stati inizializzati a `NULL`, in un albero di cui `*old` è la radice.

Quindi per inserire un nuovo nodo `*new` in un albero è sufficiente chiamare la funzione nel Listato 16.5 con il puntatore alla radice dell'albero come secondo argomento: `insertNode(new, root)`. Nel caso particolare in cui state inserendo il primo nodo dell'albero, il puntatore `root` sarebbe `NULL` e dovreste quindi solo assegnargli il valore `new`, senza chiamare la funzione `insertNode`.

```
1 void insertNode(myNode *new, myNode *old) {
2   if (new->datum <= old->datum) {
3     if (old->left == NULL) {
4       old->left = new;
5       new->up = old;
6     } else {
7       insertNode(new, old->left);
8     }
9   } else {
10    if (old->right == NULL) {
11      old->right = new;
12      new->up = old;
13    } else {
14      insertNode(new, old->right);
15    }
16  }
17 }
```

Listato 16.5 Una funzione risorsiva per l'inserimento di un nodo in un albero binario.

Osserviamo una differenza fondamentale tra uno heap e un vero albero binario. Quando si rappresenta uno heap come albero binario, la topologia di quest'ultimo è fissa e determinata solo dalla dimensione N dello heap, quindi sono immediatamente noti, ad esempio, gli elementi corrispondenti alle foglie. Al contrario, un vero albero binario ha una topologia dinamica che non è univocamente determinata dal numero N dei suoi nodi. In questo caso, ad esempio, i nodi senza figli non sono immediatamente individuabili; vanno cercati scorrendo la lista dei nodi oppure percorrendo l'albero dalla radice verso il basso. In generale, mentre gli algoritmi che lavorano su uno heap possono indifferentemente partire dalla radice o dalle foglie, quelli che operano su un albero devono considerare che l'albero ha un unico punto di accesso: la sua radice. Alternativamente potremmo accedere a un albero da un suo generico nodo, ma questo non ci fornisce nessuna informazione sulla posizione di quel nodo nella topologia dell'albero.

Quesito 16.3 *Sequenze di inserimento di dati in un albero*

Nella Figura 16.12 sono mostrati due alberi binari diversi generati con lo stesso insieme di dati, ma inserito secondo due ordini differenti. Scrivete, per ognuno dei due alberi, almeno tre possibili sequenze d'inserimento dei dati che portano al riempimento mostrato nella figura. Osservate che le sequenze di inserimento compatibili con un dato albero binario soddisfano delle relazioni di ordine solo parziale (quali?) e per questo non sono univocamente definite.

Quesito 16.4 *Sbilanciamento nel riempimento di un albero*

Nel caso siano inseriti in un albero binario di ricerca elementi di pari valore la funzione `insertNode` manifesta un errore sistematico, perché posiziona questi elementi sempre alla sinistra di quelli già presenti nella struttura. Questo errore sistematico può dar luogo ad alberi molto sbilanciati. Come potreste risolvere questo problema modificando leggermente la funzione `insertNode`? La soluzione che sfrutta l'uso dei numeri pseudocasuali è forse la piú evidente, ma ne esiste una ancora piú semplice che non fa uso di variabili (pseudo)casuali. Quale?

Laboratorio 16.6 *Profondità degli alberi binari*

Scrivete una funzione che calcoli la profondità di un nodo. Dopo aver riempito, con la funzione `insertNode`, un albero con N elementi di valore aleatorio, misuratene la profondità massima e quella media dei nodi dell'albero. Determinate la media di queste misure sperimentali su molte realizzazioni diverse del rumore, ossia dell'ordine con cui i nodi vengono aggiunti all'albero. Stimate numericamente come crescono queste due profondità all'aumentare di N, eseguendo, ad esempio con `gnuplot`, un'interpolazione (possibilmente lineare, se scegliete le giuste variabili!).

Prendete quindi in considerazione i soli nodi senza figli, cioè le foglie dell'albero; questi nodi dovrebbero fornire una buona stima della profondità dell'albero visto che compaiono alla fine di ogni ramo. Calcolate numericamente la distribuzione di probabilità della loro profondità per diversi valori di N (vi suggeriamo di usare valori nell'intervallo $10^2 \leqslant N \leqslant 10^5$ e di calcolare le medie delle distribuzioni su almeno un migliaio di realizzazioni del rumore). Stimate numericamente il valore medio $\langle h \rangle$ e la varianza $\sigma_h^2 \equiv \langle h^2 \rangle - \langle h \rangle^2$ di queste distribuzioni. Come crescono con N? Cosa potete dedurre dal loro confronto con la crescita del valor medio e massimo della profondità di tutti i nodi? Provate infine a rappresentare graficamente, per diversi valori di N, la distribuzione di probabilità della variabile $z \equiv (h - \langle h \rangle)/\sigma_h$. Dovreste osservare che le distribuzioni per diversi N sono simili, sebbene abbiano un profilo non banale, ad esempio non sono simmetriche rispetto all'origine.

```
1 myNode *searchItem(dataType item, myNode *pNode) {
2   if (pNode == NULL || item == pNode->datum) {
3     return pNode;
4   } else if (item < pNode->datum) {
5     return searchItem(item, pNode->left);
6   } else {
7     return searchItem(item, pNode->right);
8   }
9 }
```

Listato 16.6 Una funzione ricorsiva per la ricerca in un albero binario.

Giacché abbiamo un albero binario di ricerca riempito dei suoi dati, possiamo illustrare alcune funzioni di ricerca nell'albero. La *ricerca* di un dato elemento `item` nell'albero si può fare, ad esempio, con la funzione ricorsiva del Listato 16.6. La funzione `searchItem` restituisce un puntatore al nodo contenente il dato che stiamo cercando oppure il valore `NULL` nel caso tale dato non sia presente nell'albero. Il numero di chiamate ricorsive può essere al piú pari alla profondità massima dell'albero nel caso si chiami la funzione `searchItem` con `root` come secondo argomento.

Laboratorio 16.7 *Funzioni non ricorsive su alberi binari*

Scrivete versioni non ricorsive delle funzioni per l'inserimento di un nodo e la ricerca di un elemento in un albero binario. Tenete presente che per funzioni molto semplici, come `insertNode` e `searchItem`, è talvolta preferibile usare le versioni non ricorsive, perché sono piú veloci e non particolarmente piú difficili da programmare. Ricordate, però, che le prestazioni di un algoritmo dipendono molto dal computer e del compilatore che state usando; vi consigliamo di verificarle sperimentalmente di volta in volta.

Un elemento facile da trovare in un albero binario di ricerca è il *minimo assoluto*, perché coincide sempre con l'elemento piú a sinistra. La funzione nel Listato 16.7 restituisce il puntatore al nodo contenente l'elemento minimo tra quelli nell'albero (o sottoalbero) la cui radice è puntata da `pNode`. Anche l'elemento massimo si può trovare con un'analoga funzione `maximumInTree`.

```
1 myNode *minimumInTree(myNode *pNode) {
2   while (pNode->left != NULL) {
3     pNode = pNode->left;
4   }
5   return pNode;
6 }
```

Listato 16.7 Una funzione che trova l'elemento minimo in un albero binario.

Un'ultima funzione di cui abbiamo bisogno è quella che trova il nodo contenente il *dato immediatamente successivo* a quello contenuto in un determinato nodo (fornito in input alla funzione). Una possibile versione di questa funzione è riportata nel Listato 16.8.

Nel caso semplice (righe 3-5) il nodo puntato da `pNode` ha un sottoalbero a destra; il suo successore non è altro che l'elemento minimo in quel sottoalbero. Leggermente piú complicato è il caso in cui il sottoalbero di destra non esiste e l'elemento successivo va ricercato in un nodo piú in alto. In questo caso (righe 6-10) occorre risalire l'albero con la seguente regola: fintanto che si sale verso sinistra, si incontrano elementi minori e quindi si deve continuare a salire; ci si ferma solo se si raggiunge la radice (in tal caso `pParent` vale `NULL`) o se si sale verso destra (in tal caso `pParent->right` è diverso da `pNode`). Nel