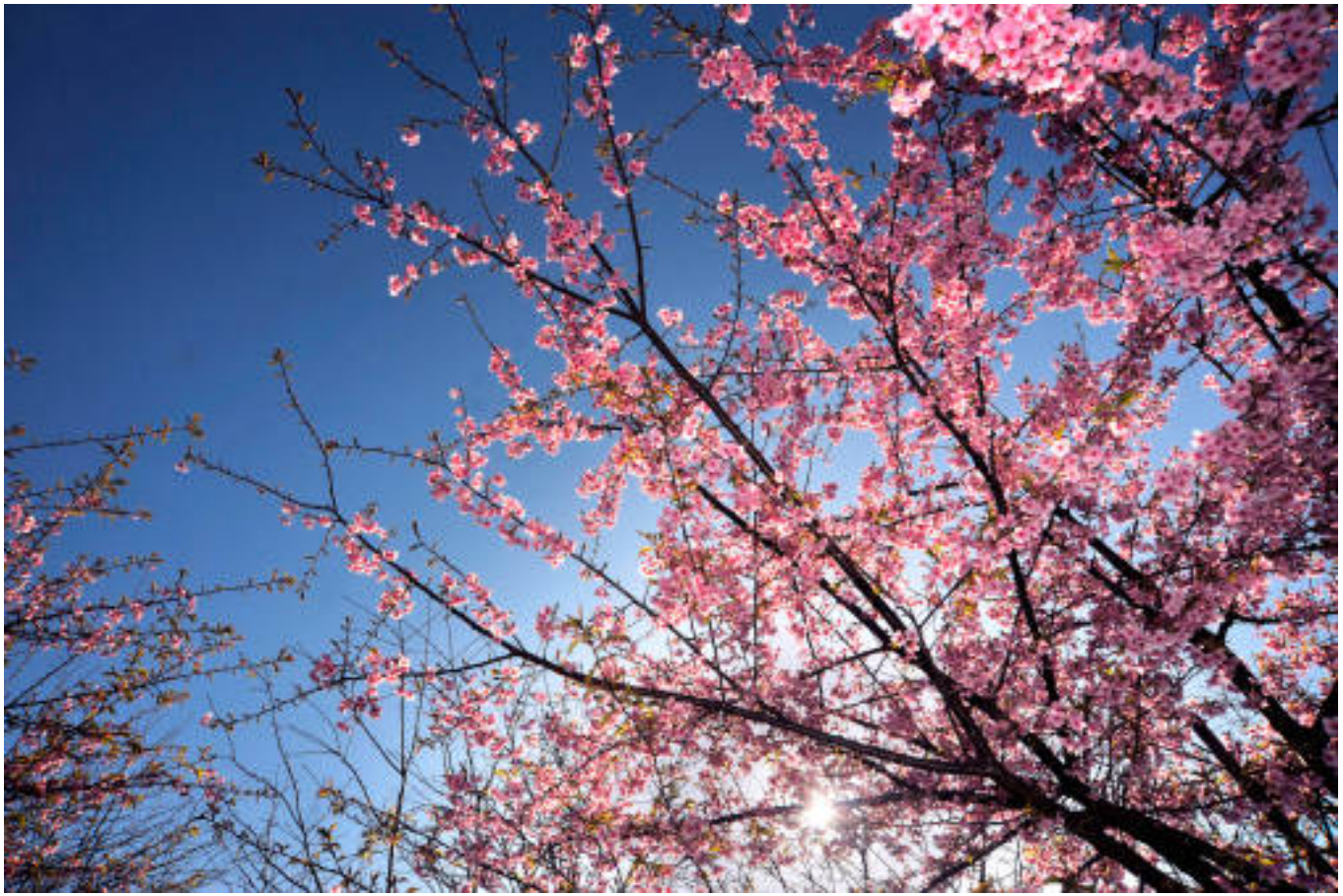
Alberi: Introduzione & BST



Alberi

- In molte situazioni trattiamo collezioni di oggetti su cui sono definite in modo naturale delle relazioni gerarchiche.
- Ad esempio, ogni persona ha una relazione di discendenza diretta con i suoi due genitori, i quali a loro volta sono in relazione con i propri genitori, e così via.
- Un albero genealogico è un tipico esempio di collezione di oggetti su cui sono definite delle relazioni gerarchiche.

La famiglia dei paperi

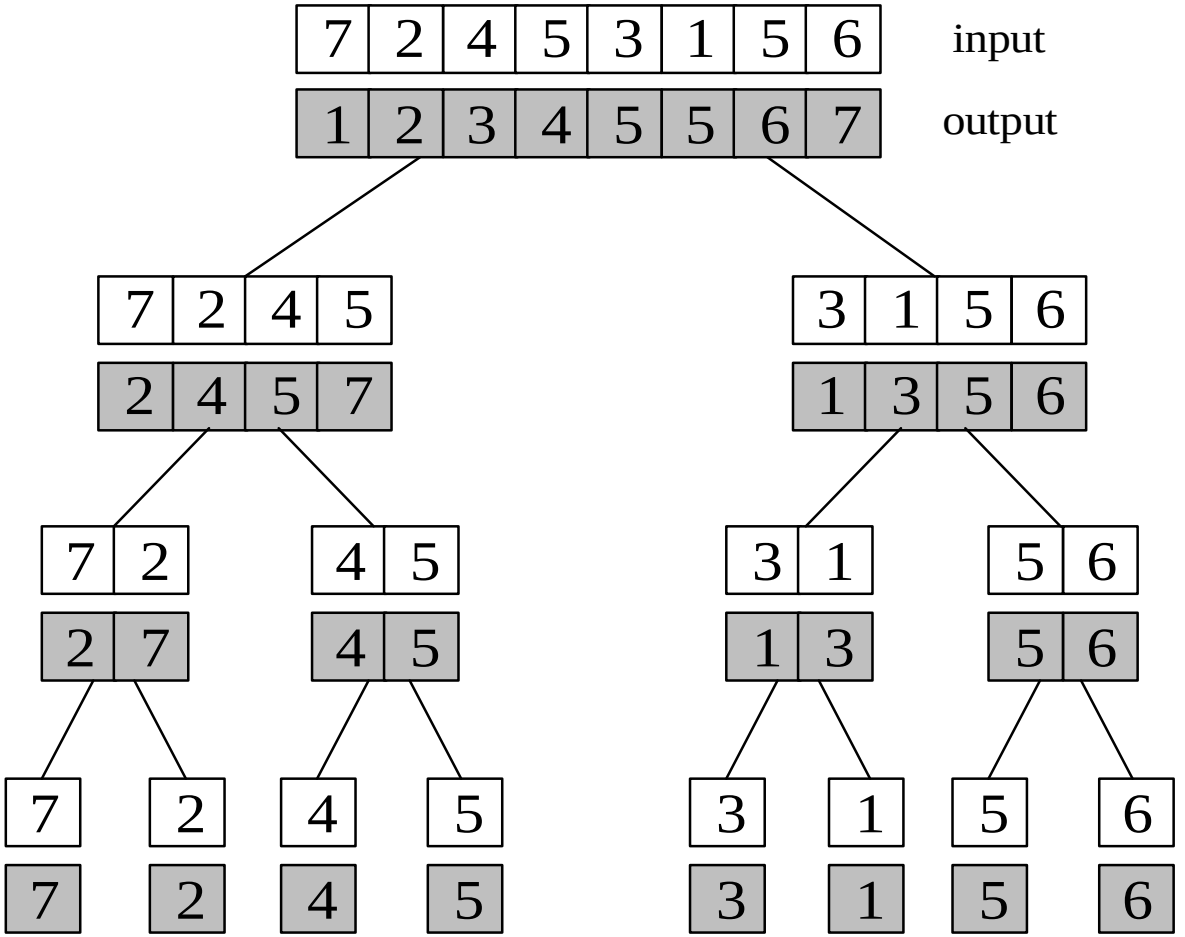
L'albero genealogico secondo Don Rosa

...sceneggiatura e disegni: Don Rosa • A cura di Lidia Cannatella • Lettering: Diego Ceresa • Coloritura: Litomilano-Carugate • Stampa: Rotolito Lombarda Pioltello • Direttore Responsabile: Paolo Cavaglione • Allegato a ZIO PAPERONE n. 87 - © Disney - I.P.

Alberi

- Un altro esempio è l'albero delle chiamate ricorsive visto quando abbiamo studiato mergesort e quicksort. In questo caso gli oggetti di interesse sono le funzioni eseguite in un programma e le relazioni sono le chiamate che attiviamo

Albero delle chiamate ricorsive di mergesort



Albero radicato, terminologia

- Un albero radicato è una coppia **T = (N, A)** costituita da un insieme **N** di **nodi** (o “vertici”) e un insieme **A** (sottoinsieme di N x N) di coppie di nodi dette **archi**.
- Il fatto che un albero sia **radicato** significa che un vertice viene indicato come “radice” ed è diverso da tutti gli altri in quanto è l'unico privo di padre.
- In un albero radicato, ogni nodo *v* tranne la radice ha uno ed un solo genitore (o padre) *u* tale che *(u, v)* ∈ A.

Albero radicato, terminologia

- Un nodo *u* può avere zero o più figli *v* tali che *(u, v)* ∈ A; il loro numero viene chiamato **grado** del nodo.
- Un nodo senza figli è chiamato **foglia**.
- I nodi che non sono né foglie né la radice sono chiamati **nodi interni**.
- Gli **antenati** di un nodo *u* sono i nodi raggiungibili da *u* salendo di padre in padre nell'albero.
- I **discendenti** di un nodo *u* sono i nodi raggiungibili da *u* scendendo di figlio in figlio nell'albero.

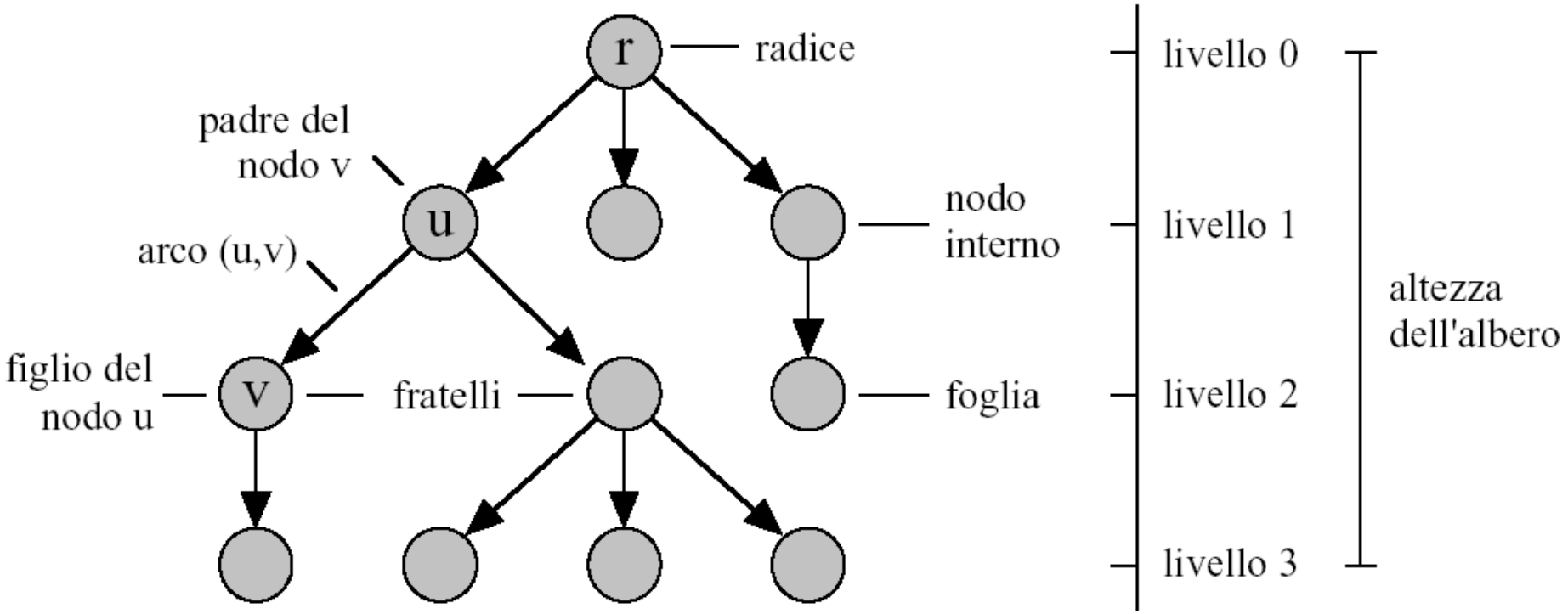
Albero radicato, terminologia

- La **profondità (o livello)** di un nodo è il numero di archi che bisogna attraversare per raggiungerlo a partire dalla radice. Può essere definita ricorsivamente come segue:
  - la radice ha profondità zero;
  - se un nodo ha profondità k, tutti i suoi figli hanno profondità k+1.

Albero radicato, terminologia

- Nodi con lo stesso genitore vengono detti **fratelli** e hanno la stessa profondità.
- L'**altezza** di un albero è la massima profondità a cui si trova una foglia.
- Un albero è un particolare tipo di grafo: è un **grafo connesso minimale** (se si toglie un arco, non è più connesso) o, se vogliamo dare una caratterizzazione in termini della ciclicità, è un **grafo aciclico massimale** (se si aggiunge un arco, non è più aciclico).

Alberi: riassunto



Alberi: definizione ricorsiva

- Un nodo singolo è un albero (è la radice dell'albero)
- Supponiamo che n sia un nodo e T_1, T_2, T_3, ..., T_k siano alberi con radici n_1, n_2, n_3, ..., n_k rispettivamente. Possiamo costruire un nuovo albero rendendo n il padre di n_1, n_2, n_3, ..., n_k. In questo nuovo albero n è la radice e T_1, T_2, T_3, ..., T_k i sottoalberi della radice. I nodi n_1, n_2, n_3, ..., n_k sono i figli di n.

Confusione nella terminologia... come sempre!

Quando parliamo di alberi, ci scontriamo con la stessa confusione nella terminologia che abbiamo già incontrato parlando di liste. Infatti...

un albero è un tipo di dato

...ma anche una struttura dati

Albero come TDD
(alcune possibili operazioni)

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 nell'albero, o `null` se v è la radice.

`figli`(*nodo* v) → ⟨*nodo*, *nodo*, . . . , *nodo*⟩
restituisce, uno dopo l'altro, i figli del

Errore nel testo: dovrebbe essere
aggiungiNodo(*nodo u, nodo v*) -> *nodo*

`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.

Albero come struttura dati
(ad esempio per implementare il TDD Dizionario)

classe `AlberoBinarioDiRicerca` **implementa** `Dizionario`:
$S(n) = O(n)$

dati:

un albero binario di ricerca T di altezza h e con n nodi, ciascuno contenente coppie $(elem, chiave)$ in cui le chiavi sono prese da un universo totalmente ordinato.

operazioni:

`search`$(chiave\ k) \rightarrow elem$ $\quad T(h) = O(h)$

partendo dalla radice, traccia un cammino nell'albero per cercare un elemento con chiave k. Su ogni nodo, usa la proprietà di ricerca per decidere se proseguire nel sottoalbero sinistro o destro.

`insert`$(elem\ e, chiave\ k)$ $\quad T(h) = O(h)$

crea un nuovo nodo v contenente la coppia (e, k), e lo aggiunge all'albero come foglia nella posizione opportuna, in modo da mantenere la proprietà di ricerca.

`delete`$(elem\ e)$ $\quad T(h) = O(h)$

se il nodo v contenente l'elemento e ha al più un figlio, elimina v collegando il figlio all'eventuale padre. Altrimenti scambia il nodo v con il suo predecessore ed elimina il predecessore.

Alberi “speciali”

Alberi “speciali”

- Esistono alcune tipologie di alberi con vincoli strutturali che ne facilitano la rappresentazione o consentono di realizzare operazioni su collezioni di elementi in modo particolarmente efficiente.
- I principali vincoli strutturali riguardano il grado dei nodi e la profondità.
- Un albero **d-ario** è un albero in cui i nodi hanno **al più** grado *d.*

Nota: nel libro di testo, a pagina 73, si legge: “”un albero d-ario è un albero in cui tutti i nodi tranne le foglie hanno grado d”. Si tratta di un refuso: la definizione corretta prevede che i nodi abbiano “**al più** grado d”, non “grado d”.

Alberi “speciali”

- Un albero d-ario in cui d=2 si dice **albero binario**
- Un **albero binario completo** è un albero binario in cui ogni livello è completamente pieno
- Un **albero binario quasi completo** è un albero binario in cui ogni livello, tranne eventualmente l'ultimo, è completamente pieno, e tutti i nodi sono il più a sinistra possibile.

Proprietà degli alberi binari completi e quasi completi

- **Proprietà 1**: Un albero binario completo di altezza h ha $2^{h+1}-1$ nodi
- **Proprietà 2**: Sia T un albero binario quasi completo di altezza h. Allora il numero n di nodi di è tale che $2^h \leq n \leq 2^{h+1}-1$
- **Proprietà 3**: L'altezza di un albero binario quasi completo T con n nodi è $h = \lfloor \log_2 n \rfloor$ (si ricorda che $\lfloor\ \rfloor$ indica la parte intera)

Proprietà 1 (indicata con Proposition 5.1)

Proposition 5.1. Un *albero binario completo* di altezza h ha $2^{h+1} - 1$ nodi.

Soluzione. La prova è per induzione su h.

- *Caso base* ($h = 0$). Un albero binario completo di altezza $h = 0$ ha un solo nodo (la radice). Inoltre, $2^{0+1} - 1 = 1$. Quindi il caso base è verificato.
- *Passo induttivo:* ($h > 1$). Assumiamo (per ipotesi induttiva) che un albero binario completo di altezza $h - 1$ abbia $2^h - 1$ nodi, e dimostriamo che un albero binario T completo di altezza h ha $2^{h+1} - 1$ nodi.
  Come è fatto T? Ha la radice, più un sottoalbero sinistro T_s e un sottoalbero destro T_d entrambi completi e di altezza $h - 1$. Per ipotesi induttiva $n_s = n_d = 2^h - 1$ (quin con n_s e n_d abbiamo indicato il numero di nodi di T_s e T_d, rispettivamente). Quindi:

$$n = 1 + n_s + n_d = 1 + (2^h - 1) + (2^h - 1) = (2 \cdot 2^h) - 1 = 2^{h+1} - 1$$

Proprietà 2 (indicata con Proposition 5.3)

Proposition 5.3. Sia T un albero binario quasi completo di altezza h. Allora il numero n di nodi di T è tale che

$$2^h \leq n \leq 2^{h+1} - 1$$

Soluzione. Per dimostrare questo risultato abbiamo bisogno di capire quale è il numero minimo ed il numero massimo che un albero binario quasi completo di altezza h può avere.

Il numero massimo di nodi che un albero binario quasi completo di altezza h può avere è pari al numero di nodi di albero binario completo di altezza h, ossia $n_{max} = 2^{h+1} - 1$ (vedi Proposizione 5.1). Ora osserviamo che un albero binario quasi completo di altezza h con numero minimo di nodi ha la seguente forma (dove T_s e T_d sono degli alberi binari completi di altezza $h - 2$):

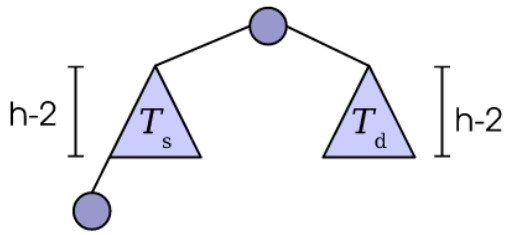


Fig. 5.1. Alberi binari quasi completo con minimo numero di nodi

Se indichiamo con n_s il numero di nodi dell'albero T_s e con n_d il numero di nodi dell'albero T_d (poichè sia T_s che T_d sono alberi binari completi di altezza $h - 2$, $n_s = n_d = 2^{h-1} - 1$). Allora:

$$n_{min} = 1 + (n_s + 1) + n_d = 2 + n_s + n_d = 2 + 2 \cdot (2^{h-1} - 1) = 2^h$$

Proprietà 3 (indicata con Proposition 5.4)

Proposition 5.4. L'altezza di un albero binario quasi completo T con n nodi è $h = \lfloor \log n \rfloor$.

Soluzione. Per la Proposizione 5.3 abbiamo che $2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1}$, cioè $h \leq \log n < h + 1$ e, quindi, $h \leq \lfloor \log n \rfloor < h + 1$. Possiamo concludere che $h = \lfloor \log n \rfloor$

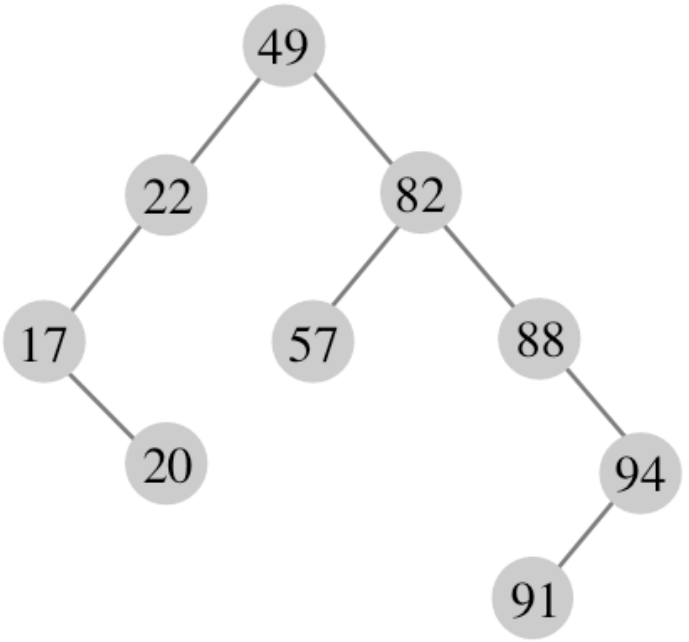
La seguente proposizione fornisce un limite superiore al numero di nodi di ciascun sottoalbero di un albero binario completo

Alberi binari di ricerca
BST = Binary Search Tree

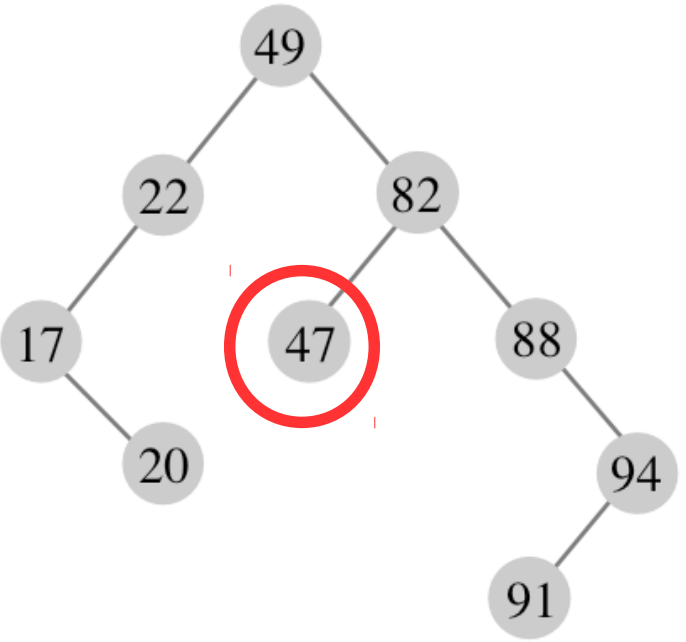
Albero binario che soddisfa le seguenti proprietà

- ogni nodo *v* contiene un elemento elem(v) cui è associata una chiave chiave(v) presa da un dominio totalmente ordinato
- le chiavi degli elementi nel sottoalbero sinistro di v sono ≤ chiave(*v)*
- le chiavi degli elementi nel sottoalbero destro di v sono ≥ chiave(*v)*

Esempi



Albero binario
di ricerca



Albero binario
non di ricerca

Nota!!!

- Il libro di testo ammette che nell'implementazione degli alberi in generale, e di quelli binari in particolare, si possa mantenere il puntatore al padre nella struttura che modella un nodo, per semplificare (secondo il testo) alcune operazioni. Inoltre il testo implementa le funzioni search, insert e delete in maniera iterativa.
- Noi di norma **non manterremo il puntatore al padre e implementeremo le funzioni in maniera ricorsiva.**
- Seguiremo quindi l'approccio del testo Aho, Hopcroft, Ullman, indicato con AHU nel seguito.

Struct per un nodo dell'albero

Nel testo AHU

```
type
    nodetype = record
        element: elementtype;
        leftchild, rightchild: ↑ nodetype
    end;
```

Nella nostra implementazione in C++

```
struct dict::bstNode {
    dictionaryElem keyVal; /* elemento del
dictionazy, ovvero coppia chiave-valore */
    bstNode* leftChild;
    bstNode* rightChild;
};
```

Ripasso: il TDD Dizionario

tipo `Dizionario`:
dati:
un insieme S di coppie $(elem, chiave)$.

operazioni:

`insert`$(elem\ e, chiave\ k)$
aggiunge a S una nuova coppia (e, k).

`delete`$(chiave\ k)$
cancella da S la coppia con chiave k.

`search`$(chiave\ k) \rightarrow elem$
se la chiave k è presente in S restituisce l'elemento e ad essa associato, e `null` altrimenti.

Tipo Dizionario

Nel testo AHU

```
type
    SET = ↑ nodetype;
```

Nella nostra implementazione in C++

```
typedef bstNode* Dictionary;
```

search

Θ(h) nel caso peggiore, con h altezza dell'albero

Nel testo AHU

```
function MEMBER ( x: elementtype; A: SET ) : boolean;
    { returns true if x is in A, false otherwise }
    begin
        if A = nil then
            return (false)  { x is never in ∅ }
        else if x = A↑.element then
            return (true)
        else if x < A↑.element then
            return (MEMBER(x, A↑.leftchild))
        else  { x > A↑.element }
            return (MEMBER(x, A↑.rightchild))
    end; { MEMBER }
```

Fig. 5.2. Testing membership in a binary search tree.

insertElem

Θ(h) nel caso peggiore, con h altezza dell'albero

Nel testo AHU

```
procedure INSERT ( x: elementtype; var A: SET );
    { add x to set A }
    begin
        if A = nil then begin
            new(A);
            A↑ .element := x;
            A↑ .leftchild := nil;
            A↑ .rightchild := nil
        end
        else if x < A↑.element then
            INSERT(x, A↑.leftchild)
        else if x > A↑.element then
            INSERT (x, A↑.rightchild)
        { if x = A↑.element, we do nothing; x is already in the set }
    end; { INSERT }
```

Fig. 5.3. Inserting an element into a binary search tree.

Cancellazione (1)

Sfrutta una funzione ausiliaria deleteMin che, preso il nodo A **non vuoto** che individua univocamente l'albero binario di ricerca del quale è radice,

1. cancella l'elemento dell'albero individuato da A che ha chiave minore e
2. restituisce il contenuto informativo (ovvero, la coppia (chiave, valore)) associato al nodo cancellato

Cancellazione (2)

Nella nostra implementazione in C++, il prototipo di deleteMin sarà dunque

```
dictionaryElem deleteMin(Dictionary&)
```

deleteMin prende come argomento un albero non vuoto identificato univocamente dal puntatore al nodo che ne è radice, cancella da esso in nodo con chiave minima e restituisce un elemento di tipo dictionaryElem, ovvero una coppia (chiave, valore) implementata in modo opportuno in C++

Cancellazione (3)

deleteMin si basa su un'osservazione molto semplice: per come è costruito un BST, **il nodo con chiave minima di un BST è sempre quello più a sinistra**.

Non è detto che il nodo con chiave minima sia una foglia, ma sicuramente non ha figli a sinistra. Se ne avesse, per definizione di BST essi dovrebbero avere chiave minore!

Naturalmente possiamo cercare il nodo con chiave minima in un sottoalbero: sarà il nodo più a sinistra in quel sottoalbero.

Cancellazione (4)

Per cercare il nodo con chiave minima a partire da un nodo A basta “scendere” a sinistra. Quando trovo un nodo che non ha il figlio sinistro, quello è il minimo del sottoalbero radicato in A.

Per rimuovere tale nodo minimo, basta sostituirlo con il suo figlio destro (se esiste, altrimenti lo cancello e basta).

Cancellazione (5)

La funzione deleteElem è definita per casi e viene chiamata inizialmente sulla radice dell'albero che rappresenta il dizionario:

-- se il nodo *n* passato come argomento è vuoto, non c'é niente da cancellare

-- se la chiave dell'elemento da cancellare è > della chiave del nodo *n* passato come argomento, “scendo” a destra chiamando ricorsivamente deleteElem sul figlio destro di n, altrimenti “scendo” a sinistra

Cancellazione (6)

-- se la chiave dell'elemento da cancellare è uguale alla chiave del nodo *n* passato come argomento, tale nodo *n* è proprio quello che devo cancellare:

- se *n* non ha figli, lo cancello
- se *n* ha solo il figlio destro, lo cancello e sostituisco *n* con il figlio destro; analogamente se *n* ha solo il figlio sinistro
- se *n* ha entrambi i figli, chiamo deleteMin sul figlio destro di *n* e sostituisco il contenuto informativo che caratterizzava *n* con il valore restituito da deleteMin (cioè la coppia *(c, v)* t.c. *c* è la minima chiave del sottoalbero destro di *n*). In questo modo la proprietà dei BST è ancora mantenuta

deleteMin

(funzione ausiliaria di deleteElem, Θ(h) nel caso peggiore)

Nel testo AHU

```
function DELETEMIN ( var A: SET ) : elementtype;
    { returns and removes the smallest element from set A }
begin
    if A↑.leftchild = nil then begin
        { A points to the smallest element }
        DELETEMIN := A↑.element;
        A := A↑.rightchild
            { replace the node pointed to by A by its right child }
    end
    else { the node pointed to by A has a left child }
        DELETEMIN := DELETEMIN(A↑.leftchild)
end; { DELETEMIN }
```

Fig. 5.4. Deleting the smallest element.

deleteElem

Θ(h) nel caso peggiore, con h altezza dell'albero

Nel testo AHU

```
procedure DELETE ( x: elementtype; var A: SET );
    { remove x from set A }
begin
    if A <> nil then
        if x < A↑.element then
            DELETE(x, A↑.leftchild)
        else if x > A↑.element then
            DELETE(x, A↑.rightchild)
        { if we reach here, x is at the node pointed to by A }
        else if (A↑.leftchild = nil) and (A↑.rightchild = nil) then
            A := nil  { delete the leaf holding x }
        else if A↑.leftchild = nil then
            A := A↑.rightchild
        else if A↑.rightchild = nil then
            A := A↑.leftchild
        else { both children are present }
            A↑.element := DELETEMIN(A↑.rightchild)
end; { DELETE }
```

Fig. 5.5. Deletion from a binary search tree.