Algoritmi e strutture dati

Alberi

Menú di questa lezione

In questa lezione introduciamo la struttura dati ad albero, e vediamo una prima specializzazione: gli alberi binari di ricerca.

Alberi

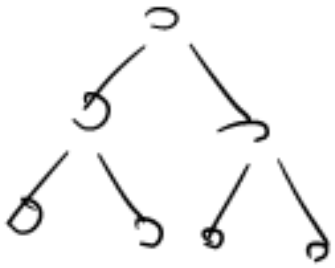
Gli alberi sono strutture dati fondamentali dinamiche e sparse. A seconda degli usi che se ne fanno, possono essere basate sull'ordinamento oppure no. Da un lato, possiamo dire che gli alberi generalizzano le liste (se vediamo il puntatore **next** come un successore, allora nelle liste il successore è unico e negli alberi no). In questo senso, possiamo anche dire che i grafi, che vedremo piú avanti, a loro volta generalizzano gli alberi.

Alberi

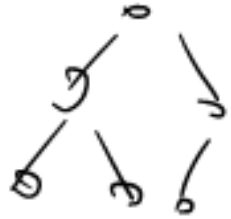
D'altra parte, peró, dal punto di vista delle strutture dati, liste, alberi, e grafi sono oggetti molto diversi e, in generale, con usi molto diversi; il fatto che uno sia la generalizzazione dell'altro non significa che gli utilizzi si ereditino. Inoltre, il concetto di albero lo abbiamo giá incontrato varie volte. Bisogna fare attenzione a non confondere i vari concetti: qui studiamo una struttura dati concreta; le heaps sono array che possono essere convenientemente visti come alberi; un albero di decisione (come nella dimostrazione della complessitá minima per l'ordinamento basato su confronti), o un albero di ricorsione, sono semplicemente strutture concettuali, non strutture dati. La struttura dati **albero** è troppo generica e ubiqua per essere associata a uno o più nomi specifici.

Alberi

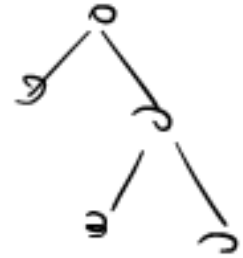
In informatica, un **albero radicato** (semplicemente **albero**) è un grafo aciclico connesso tale che ogni coppia di vertici è connessa da al piú un cammino. I vertici vengono chiamati **nodi**. Un albero è k-ario se ogni nodo ha al più k figli distinti (esempio: albero binario). Le seguenti definizioni valgono per tutti gli alberi (normalmente usiamo solo alberi binari; queste definizioni sono date direttamente nel caso binario ma sono estendibili al caso k-ario): **completo**: ogni livello è completo, cioè tutti i nodi di uno stesso livello hanno esattamente zero o due figli; **quasi completo**: ogni livello, tranne eventualmente l'ultimo, è completo; **bilanciato**: tra il percorso semplice piú breve e quello semplice piú lungo la radice ed una foglia esiste una differenza, al massimo costante; e **pieno**: ogni nodo ha zero o due figli.



ESEMPIO
DI ALBERO
BINARIO COMPLETO

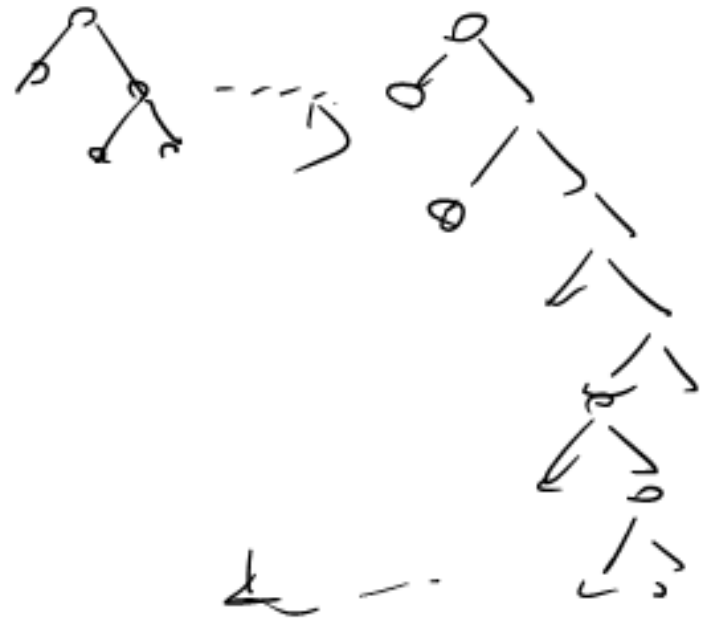


ESEMPIO DI
ALB. BINAR.
PIENO COMPLETO



ESEMPIO DI
ALB. BINAR.
PIENO

ESEMPI DI
ALBERI BILANCIATI



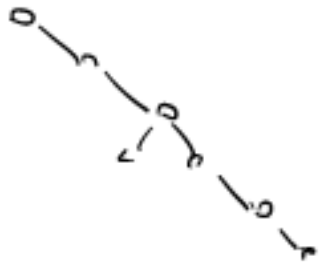
Alberi

Anche queste definizioni sono soggette a variazione in certi testi, ed in certe situazioni. Per esempio, alcuni definiscono un albero bilanciato solo quando la differenza tra due percorsi semplici è al massimo uno, che è molto diverso da come lo abbiamo definito noi. Queste definizioni hanno peró lo stesso spirito, e a seconda del caso, useremo definizioni diverse se necessario (per esempio, negli esercizi).

Alberi

Un albero puó anche essere **diretto** o **indiretto**. Gli alberi diretti sono tali che ogni sottoalbero ha un nodo predeterminato chiamato **radice**, che, in maniera astratta, dota l'albero di un ordinamento topologico privilegiato. Negli alberi indiretti la radice viene invece individuata in maniera arbitraria, cosi come la direzionalità dei cammini. Gli alberi che noi vediamo in questa sezione sono tutti diretti. Quando esiste una direzione nei cammini (fissata negli alberi diretti, e arbitraria in quelli indiretti), si individuano in maniera naturale le **foglie**, cioè i nodi senza figli. L'**altezza** di un albero è il massimo numero di archi su un percorso semplice dalla radice ad una foglia.

Alberi: altezza

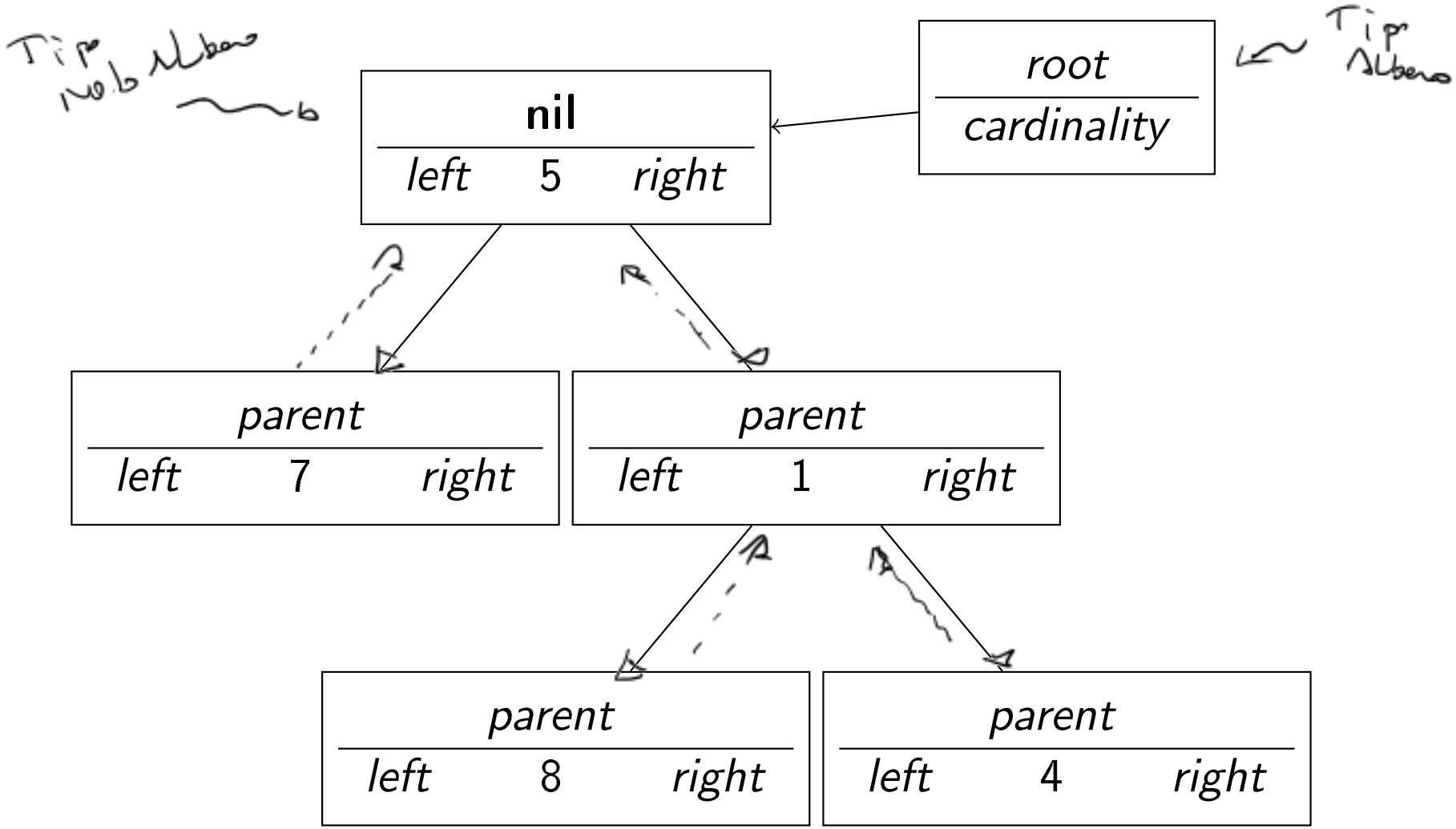


Quando abbiamo studiato le heaps, che, abbiamo detto, si possono vedere come alberi binari, abbiamo scoperto che gli alberi binari quasi completi con n elementi hanno altezza $\Theta(log(n))$; pertanto questo vale anche per gli alberi completi. Piú avanti, dimostreremo che questo fatto vale anche per gli alberi bilanciati. Ma quando un albero non ha nessuna proprietà strutturale come quelle menzionate, allora la sua altezza è $\Theta(n)$ nel caso peggiore, e $\Theta(log(n))$ nel caso medio. Quando necessario, faremo riferimento a questo fatto.

Alberi: struttura

I nodi di un albero binario possono essere pensati come oggetti che possiedono, almeno, una **chiave** ($x.key$), e tre puntatori: il **padre** ($x.p$), il **figlio destro** ($x.right$), ed il **figlio sinistro** ($x.left$). Tutti i puntatori sono **nil** quando non sono definiti. L'operazione di creazione di un albero vuoto è immediata. Le operazioni di inserimento e cancellazione di un elemento, invece, dipendono fortemente dall'applicazione che abbiamo in mente. Non possiamo dare delle operazioni generiche, perchè queste cambiano fortemente da caso a caso.

Alberi: struttura



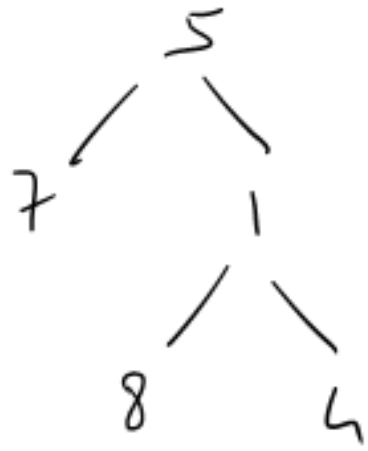
Alberi: struttura

Un albero sono la struttura ideale per memorizzare oggetti che hanno una natura ricorsiva. Ad esempio, formule matematiche o formule logiche. In intelligenza artificiale, come altro esempio, un albero decisionale è un algoritmo fondamentale di ragionamento automatico, che ha proprio una natura ricorsiva. Anche se non possiamo vedere operazioni di modifica (negli alberi generici), è interessante studiare come una struttura ad albero puó essere visitata. Ogni visita diversa corrisponde a uno scopo diverso, e puó dare informazioni diverse sull'albero. I nomi delle visite che vediamo sono originati, in parte, dal loro uso negli alberi semi-ordinati (che vedremo più avanti), normalmente vengono chiamate visite **in order**, **pre order**, e **post order**. Queste sono anche casi particolari delle visite che studieremo nei grafi.

Alberi: *TreeInOrderTreeWalk*

```
proc TreeInOrderTreeWalk (x)
  if (x ≠ nil )
    then
      TreeInOrderTreeWalk(x.left)
      Print(x.key)
      TreeInOrderTreeWalk(x.right)
```

$$T(n) = T(\quad) + T(\quad) + \Theta(1)$$



INOV: 7 5 8 1 4

PREOV: 5 7 1 8 4

POSOV: 7 8 4 1 5

Correttezza e complessitá di *TreeInOrderTreeWalk*

La **terminazione** di questa procedura si vede immediatamente dalla struttura ricorsiva degli alberi e della procedura. Lo stesso vale per la sua **correttezza**, basta vedere che tutti gli elementi sono visitati esattamente una volta. La complessità di una visita, come vedremo, non è direttamente collegata all'altezza dell'albero, e pertanto non presenta la separazione in caso medio e peggiore. Consideriamo la generica visita di un nodo x tale che ci sono k nodi nel sotto-albero radicato a sinistra, il che implica che ci sono $n - k - 1$ nodi nel sotto-albero radicato a destra. Possiamo considerare che il costo della visita a x stesso sia costante.

Correttezza e complessità di *TreeInOrderTreeWalk*

Il costo totale è dato dalla ricorrenza:

$$T(n) = T(k) + T(n-k-1) + \Theta(1).$$

Sospettiamo che $T(n) = \Theta(n)$, e dimostriamo, attraverso la sostituzione, che $T(n) = O(n)$. Scegliamo di ipotizzare $T(n) \leq c \cdot n$ per qualche $c > 0$:

$$\begin{aligned} T(n) &= T(k) + T(n-k-1) + \Theta(1) & \\ &\leq c \cdot k + c \cdot (n-k-1) + \Theta(1) & \text{ipotesi} \\ &= c \cdot k + c \cdot n - c \cdot k - c + \Theta(1) & \text{calcolo} \\ &= c \cdot n & \text{calcolo} \end{aligned}$$

Quindi $T(n) = O(n)$. Poichè non è possibile visitare un albero senza aver visitato tutti i nodi, la complessità di una visita è certamente $\Omega(n)$. Concludiamo quindi che la complessità di *TreeInOrderTreeWalk* è $\Theta(n)$, in tutti casi.

(è tutti gli altri visite)

Alberi: altre visite

Altre visite possibili si ottengono cambiando l'ordine delle chiamate ricorsive, e ottengono risultati diversi. Per fare un esempio, quando in un albero è memorizzata una formula matematica, questa puó essere valutata attraverso una visita post-order, perchè vengono prima valutati i figli di un nodo e poi il nodo stesso. La complessitá, la correttezza, e la terminazione di entrambe le altre visite si possono mostrare come nel caso della visita in order.

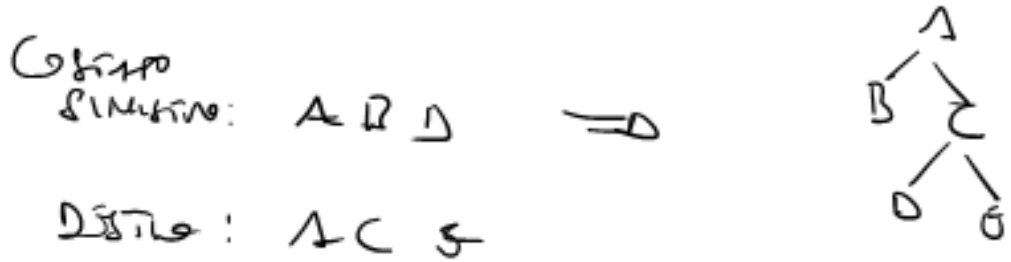
Alberi: altre visite

```
proc TreePreOrderTreeWalk (x)
  if (x ≠ nil )
    then
      Print(x.key)
      TreeInOrderTreeWalk(x.left)
      TreeInOrderTreeWalk(x.right)
```

Alberi: altre visite

```
proc TreePostOrderTreeWalk (x)
  if (x ≠ nil )
    then
      { TreeInOrderTreeWalk(x.left)
        TreeInOrderTreeWalk(x.right)
        Print(x.key)
```

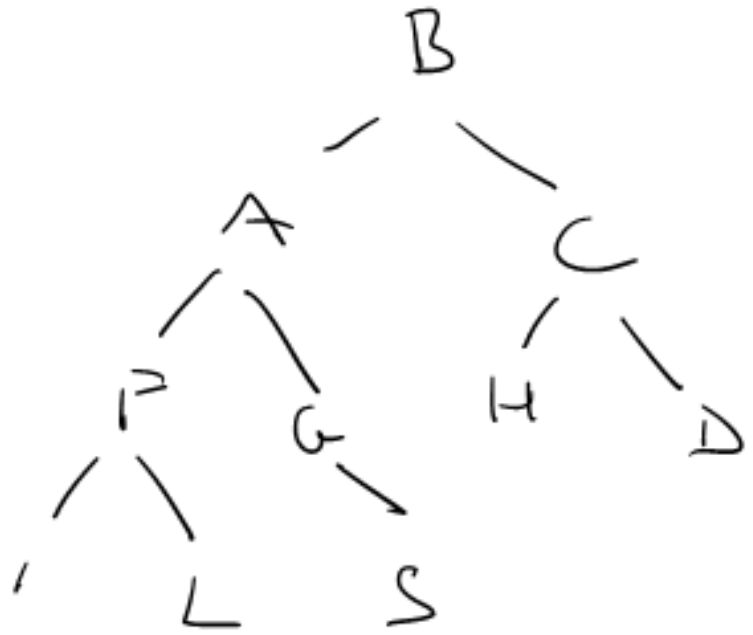
Alberi: intuizione algoritmica



Considerata l'alta versatilitá degli alberi, questi sono utilizzati in molti contesti diversi per sviluppare e testare l'intuizione algoritmica. Esempi includono problemi tipici come: trovare un algoritmo per stampare tutte e sole le chiavi della **frontiera** di T; trovare un algoritmo per stampare tutte e sole le chiavi del **costato sinistro** (o destro) di T; dire quale/quali visite sono necessarie per ricostruire la struttura di T; arricchire la struttura di T in maniera che ogni nodo punti, anche, allo **zio** (se esiste), oltre che al padre, e molti altri.

Esercizio 85:

	Sx							Dx		
In:	I	F	L	A	G	S	B	H	C	D
Pre:	B	A	F	I	L	G	S	C	H	D



h = 3

A

Alberi binari di ricerca: introduzione

Tra tutti i possibili alberi, un caso particolare sono gli **alberi binari di ricerca** (**BST**), che quindi sono una struttura dinamica, basata sull'ordinamento, e implementata in maniera sparsa. Associamo agli alberi binari di ricerca le operazioni di inserimento, cancellazione, ricerca, minimo, massimo, successore, e predecessore. Possiamo farlo perchè la struttura è basata sull'ordinamento delle chiavi. Se decidessimo di usare una lista per memorizzare chiavi intere, avremmo comunque potuto implementare tutte queste operazioni, ma non sarebbero state interessanti dal punto di vista della complessitá, nè media nè pessima. I nodi di un BST sono nodi un albero definiti come nel caso generale.

Alberi binari di ricerca: introduzione

Gli alberi binari di ricerca sono stati nominati molte volte nella letteratura informatica. I nomi più spesso associati alla loro introduzione e trattamento formale sono P.F. Windley, A.D. Booth, A.J.T. Colin, e T.N. Hibbard.

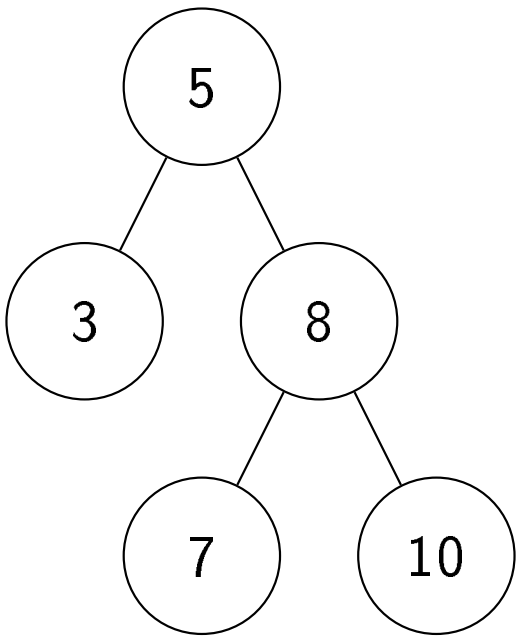
Alberi binari di ricerca: introduzione

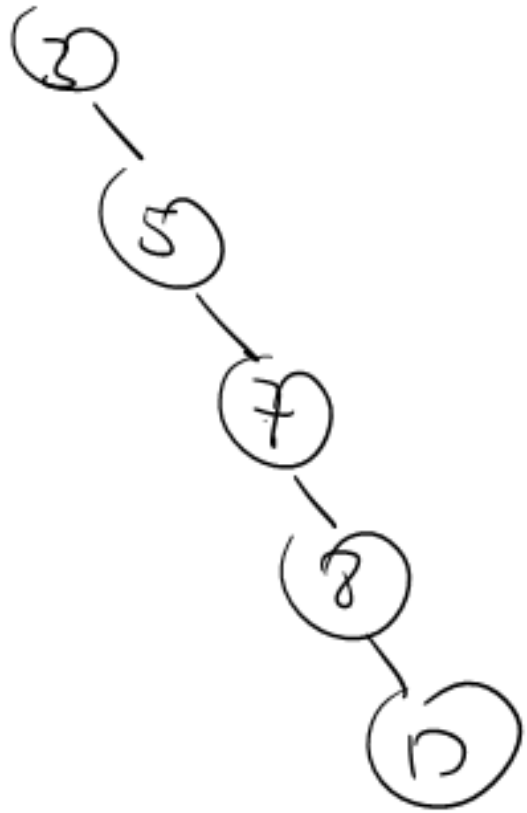
Le regole che un albero binario di ricerca deve rispettare (anche note come **proprietá BST**), sono:

1. Per ogni nodo x, se un nodo y si trova nel **sotto-albero sinistro**, allora $y.key \leq x.key$;
2. Per ogni nodo x, se un nodo y si trova nel **sotto-albero destro**, allora $y.key > x.key$.

Dunque si può dire che un BST è **parzialmente ordinato**.

Alberi binari di ricerca: introduzione





Alberi binari di ricerca: creazione e visita di un albero

I BST si creano vuoti come nel caso generale ed hanno esattamente la stessa struttura. Tutte le visite che abbiamo visto nella lezione sugli alberi generici hanno senso, naturalmente, su alberi particolari come i BST. Inoltre, in alcuni casi, come quello della visita in order, restituiscono un risultato ancora più naturale, per così dire. Infatti, se un albero è un BST regolare, il risultato della sua visita in order è l'insieme delle chiavi ordinato.

Alberi binari di ricerca: ricerca, minimo e massimo

Consideriamo adesso un BST giá costruito la cui radice è puntata da x e poniamoci il problema di cercare un elemento a chiave k. Vogliamo una procedura che, dati x e k, ritorna un puntatore al nodo (un nodo) che contiene k, se esiste, e ritorna **nil** altrimenti. Vediamo la procedura *BSTTreeSearch*.

```
proc BSTTreeSearch(x, k)
 { if ((x = nil ) or  (x.key = k))
     then  return x
   if (k ≤ x.key)
     then  return BSTTreeSearch(x.left, k)
     else  return BSTTreeSearch(x.right, k)
```

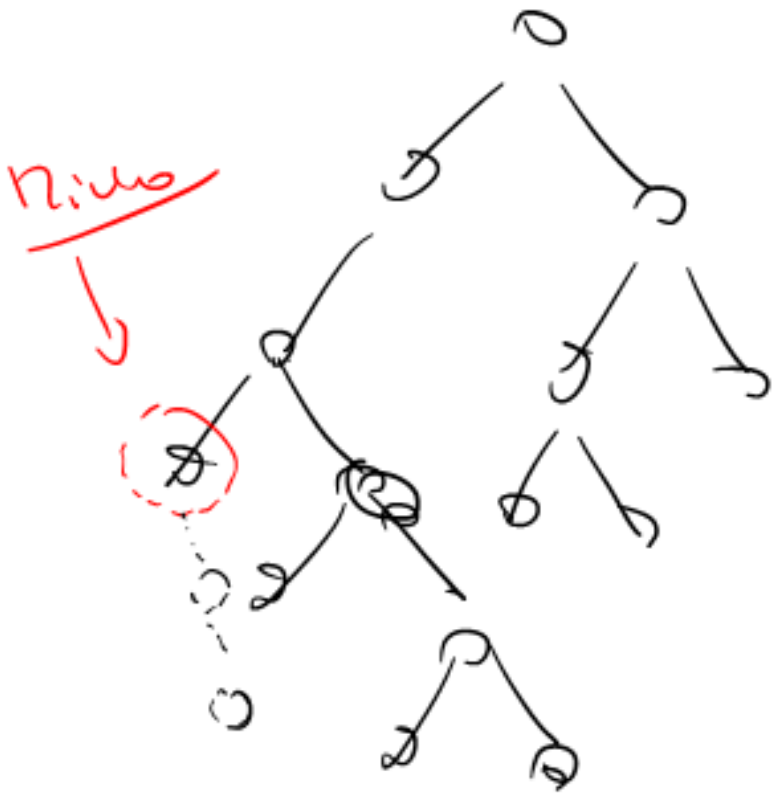
Correttezza e complessità di *BSTTreeSearch*

La **correttezza** di *BSTTreeSearch* è triviale: se la chiave esiste, questa viene trovata sicuramente, perché si visitano tutti i nodi nella parte dell'albero dove la chiave deve essere. Si puó mostrare formalmente per induzione sulla altezza dell'albero. La **complessitá** di *BSTTreeSearch* è direttamente proporzionale alla sua altezza. Nel caso peggiore, l'albero assume come sappiamo l'aspetto di una lista e la ricerca di una chiave che non esiste prende tempo $\Theta(n)$. Nel caso medio, l'albero ha altezza logaritmica, e la ricerca di una chiave, anche non esistente, ha tempo $\Theta(log(n))$.

BSTTreeMinimum, *BSTTreeMaximum*, correttezza e complessità

```
proc BSTTreeMinimum(x)
{ if (x.left = nil )
    then return x
  return BSTTreeMinimum(x.left)
```

Il nodo che contiene la chiave **minima** di un BST si trova sempre sull'ultimo nodo del ramo che dalla radice percorre sempre rami sinistri, mentre quello che contiene la chiave **massima** sull'ultimo nodo del ramo che dalla radice percorre sempre rami destri. La **correttezza** di *BSTTreeMinimum* e *BSTTreeMaximum* si ricava immediatamente da questa osservazione, e la loro **complessità** è proporzionale all'altezza dell'albero, cioè $\Theta(n)$ nel caso peggiore e $\Theta(log(n))$ nel caso medio.


nivo

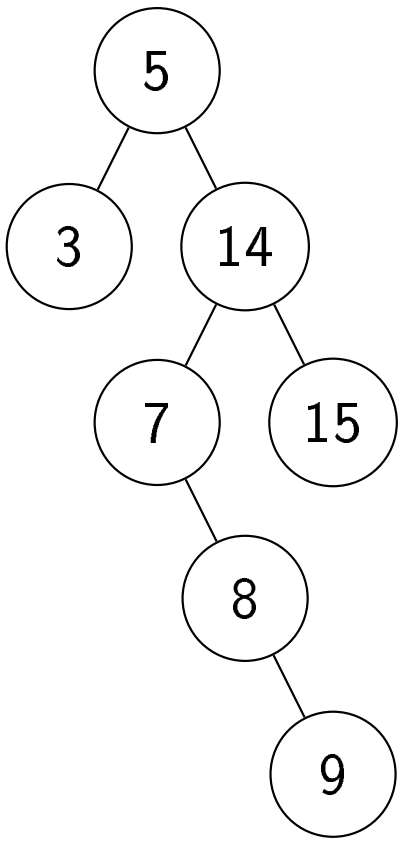
Alberi binari di ricerca: successore e predecessore

Ci poniamo adesso il problema, dato un nodo x in un BST, di trovare il nodo y, se esite, tale che $y.key$ è il **successore immediato** di $x.key$ nell'ordinamento naturale delle chiavi. Lo faremo con la procedura *BSTTreeSuccessor*, che deve tener conto di casi particolari.

il successore di 14 è 15

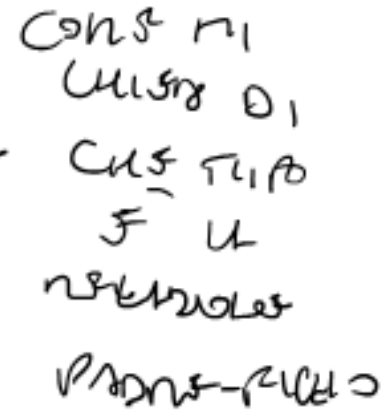
il successore di 3 è 5

il successore di 9 è 14



Alberi binari di ricerca: successore e predecessore

```
proc BSTTreeSuccessor (x)
  if (x.right ≠ nil )
     then return BSTTreeMinimum(x.right)
  y = x.p
  while ((y ≠ nil ) and (x = y.right))
    { x = y
      y = y.p
  return y
```



Andiamo per casi: se x ha figlio destro, allora il suo successore immediato é necessariamente il **minimo** del sottoalbero destro di x; se, invece, x non ha figlio destro, allora il suo successore immediato si trova tra i suoi antenati: in particolare, bisogna risalire finché la relazione padre-figlio è di tipo padre-figlio **sinistro**.

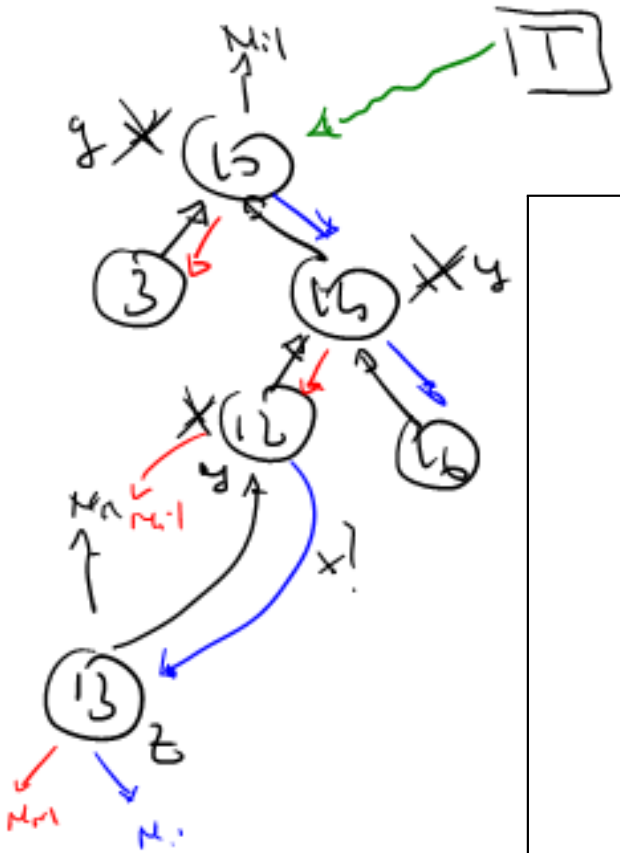
Correttezza e complessità di *BSTTreeSuccessor*

La **correttezza** di *BSTTreeSuccessor* è evidente, considerato che implementa esattamente la strategia vista prima. La **complessitá** di *BSTTreeSuccessor* è proporzionale all'altezza dell'albero, cioè $\Theta(n)$ nel caso peggiore e $\Theta(log(n))$ nel caso medio. Una procedura simmetrica si puó scrivere per trovare il predecessore di un dato nodo, se esiste.

Alberi binari di ricerca: inserimento

L'operazione di **inserimento** di un nodo è quella che ci permette di costruire e modificare un dato BST. Stabiliamo che l'inserimento opera su un BST (possibilmente vuoto) denotato da T, tale che $T.root =$**nil** quando l'albero è vuoto, e punta alla radice di T in caso contrario. Si inserisce in T un nodo z, precedentemente creato, in maniera che $z.key$ contiene la chiave da inserire, e $z.left = z.right =$**nil**; si noti che il nuovo nodo inserito finisce sempre per essere una nuova foglia di T.

Alberi binari di ricerca: inserimento



```
proc BSTTreeInsert (T, z)
  y = nil
  x = T.root
  while (x ≠ nil )
    y = x
    if (z.key ≤ x.key)
      then x = x.left
      else x = x.right
  z.p = y
  if (y = nil )
    then T.root = z
  if ((y ≠ nil ) and (z.key ≤ y.key)
    then y.left = z
  if ((y ≠ nil ) and (z.key > y.key)
    then y.right = z
```

Correttezza e complessità di *BSTTreeInsert*

Vogliamo mostrare che *BSTTreeInsert* é **corretta**, cioè se T è un BST e T' è il risultato di un inserimento, allora T' é un BST. Se T é vuoto, il ciclo **while** non si esegue, e tra le istruzioni restanti si solo esegue la prima, mettendo z come radice di T', che diventa un albero con un solo nodo e quindi corretto. Sia quindi T un BST corretto non vuoto. Vogliamo mostrare che l'**invariante** del ciclo é: la posizione corretta di z é nel sottoalbero radicato in x, e y ne mantiene il padre. Questa è:

- vera all'inizio (**caso base**), perchè $x = T.root$;
- vera anche dopo l'i-esima esecuzione del ciclo (**caso induttivo**): assumendola vera dopo la $(i-1)$-esima esecuzione, z viene confrontato con x (che non é ancora una foglia) e x viene spostato coerentemente, cosí come y, mantenendo vera la proprietá.

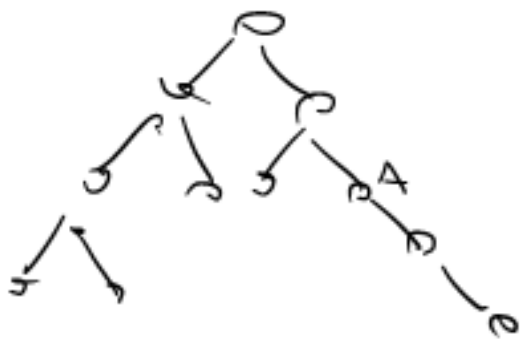
Alla fine del ciclo, $x =$**nil**, ed é precisamente la posizione di z: poiché si é persa la relazione padre-figlio tra y ed x, le ultime due istruzioni recuperano questa relazione per ottenere la posizione corretta.

Correttezza e complessità di *BSTTreeInsert*

Osserviamo che è l'operazione di inserimento che si occupa di decidere se le chiavi uguali vanno a sinistra o a destra (proprietà che abbiamo arbitrariamente deciso al principio). Tutte e due le scelte sono buone: bisogna peró controllare l'implementazione di tutte le altre operazioni in base alla scelta fatta sull'inserimento in maniera da garantire la coerenza. La **complessità** di *BSTTreeInsert*, come tutte le altre operazioni che abbiamo visto, è proporzionale alla altezza dell'albero, e pertanto è $\Theta(n)$ nel caso peggiore e $\Theta(log(n))$ nel caso medio.

Alberi binari di ricerca: eliminazione

Eliminare un elemento da un BST dato è una operazione leggermente più difficile delle altre. Infatti, considerando un nodo z qualsiasi, se z è foglia, si puó eliminare semplicemente, e se z ha un solo figlio, allora l'operazione di eliminazione coincide con l'operazione di eliminazione in una lista. Ma se z ha due figli, allora dobbiamo trovare il modo di ricostruire l'albero dopo l'eliminazione: questo è il caso complesso. Un' osservazione ci viene in aiuto: se z ha due figli, il nodo che contiene la chiave successore di quella contenuta in z certamente non ha mai figlio sinistro. Questa osservazione è importante perchè ci permette di ridurre il caso difficile ad uno piú semplice che sappiamo trattare.

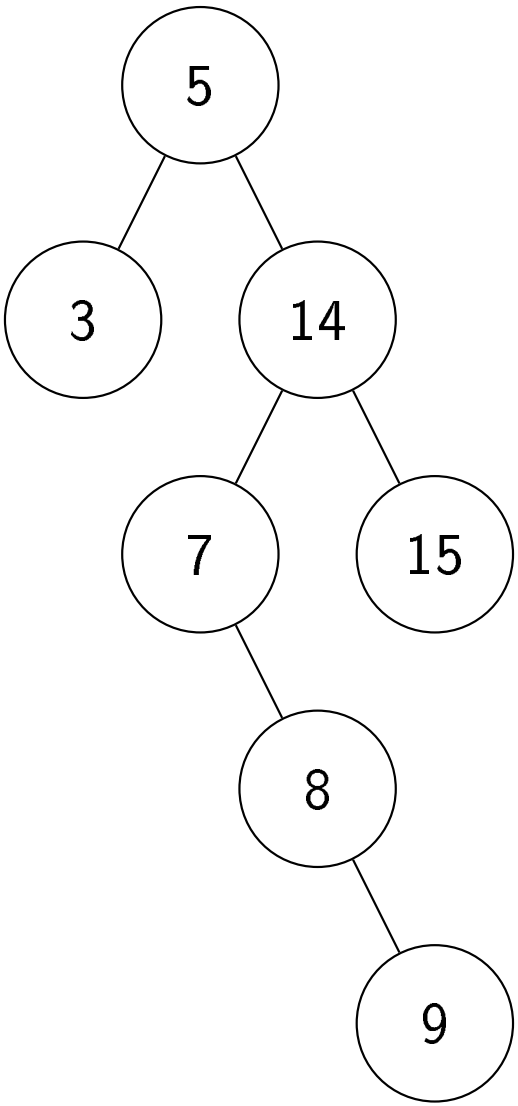


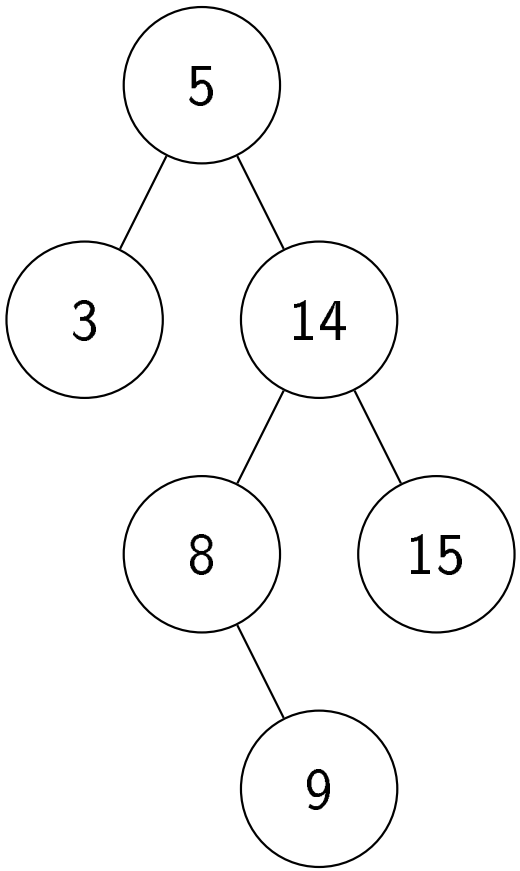
Alberi binari di ricerca: eliminazione

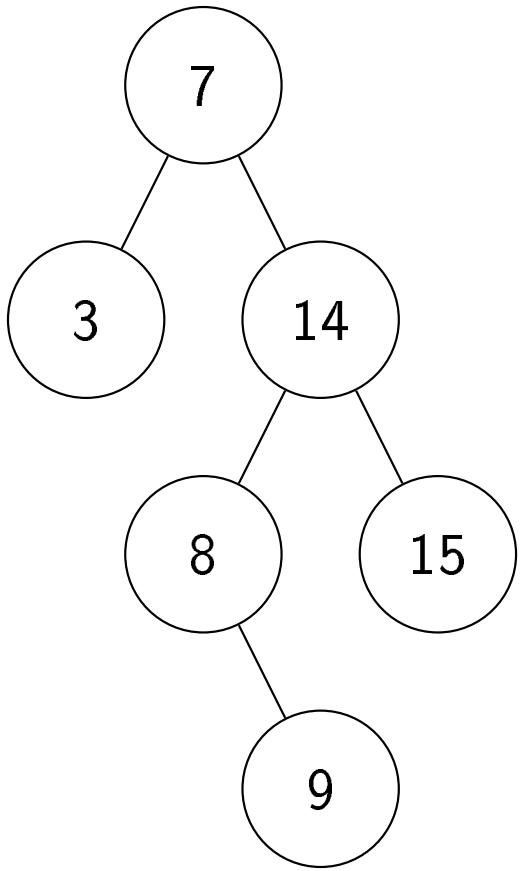
Quindi procediamo cosi. Se z non ha figli sinistri, o è una foglia, allora **trapiantiamo** il sotto-albero $z.right$ al posto di z (anche se $z.right$ è **nil**: caso z senza figli). Se z ha figlio sinistro, ma non destro, allora **trapiantiamo** il sotto-albero $z.left$ al posto di z ($z.left$ non puó essere **nil**, altrimenti saremmo nel caso anteriore). Se invece z ha due figli, allora andiamo a prendere il suo successore immediato y, che si trova nel sotto-albero destro di z e non ha **al piú un figlio**. Il nodo y va a prendere il posto di z e se y è figlio immediato di z allora il figlio destro di z diventa il figlio destro di y, e il resto rimane invariato, altrimenti (y è nel sotto-albero destro di z ma non è suo figlio immediato) allora prima rimpiazziamo y con il suo figlio destro, e poi rimpiazziamo z con y.

Alberi binari di ricerca: eliminazione

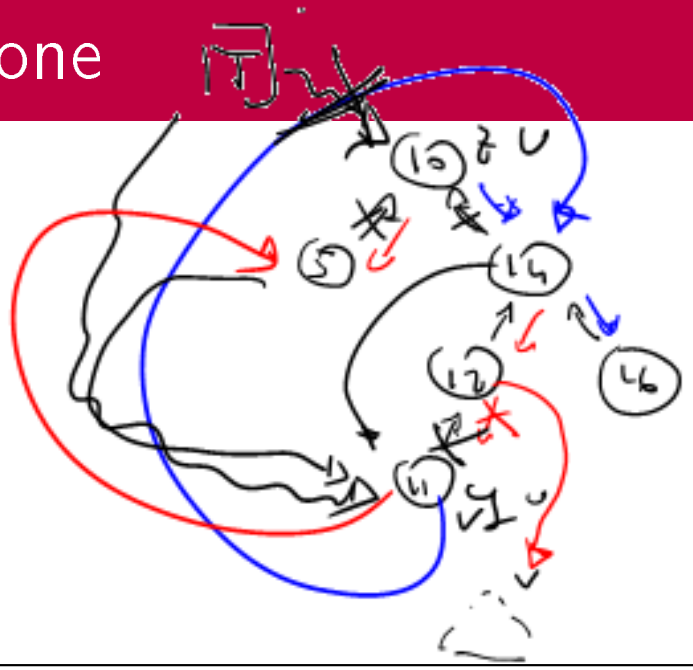
Ad esempio, eliminiamo 5 (caso 3.2):





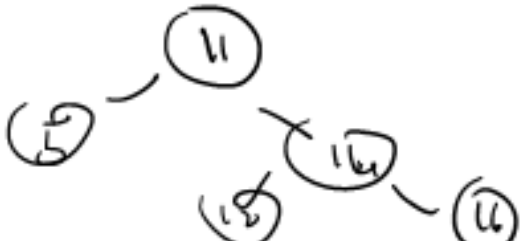


Alberi binari di ricerca: eliminazione



```
proc BSTTreeDelete (T, z)
  if (z.left = nil )
    then BSTTransplant(T, z, z.right)
  if ((z.left ≠ nil ) and (z.right = nil ))
    then Transplant(T, z, z.left)
  if ((z.left ≠ nil ) and (z.right ≠ nil ))
    then
      y = BSTTreeMinimum(z.right)
      if (y.p ≠ z)
        then
          BSTTransplant(T, y, y.right)
          y.right = z.right
          y.right.p = y
      BSTTransplant(T, z, y)
      y.left = z.left
      y.left.p = y
```

```
proc BSTTreeTransplant (T, u, v)
  if (u.p = nil )
    then T.root = v
  if ((u.p ≠ nil ) and (u = u.p.left))
    then u.p.left = v
  if ((u.p ≠ nil ) and (u = u.p.right))
    then u.p.right = v
  if (v ≠ nil )
    then v.p = u.p
```



Correttezza e complessità di *BSTTreeDelete*

La **correttezza** di *BSTTreeDelete* è implicita nell'algoritmo stesso, la cui casistica, studiata prima, riflette i passi necessari. Quindi consideriamo *BSTTreeDelete* corretto per progettazione. La **complessità** di *BSTTreeDelete*, che non contiene cicli, è, nuovamente, $\Theta(n)$ nel caso peggiore e $\Theta(log(n))$ nel caso medio; questo si deve naturalmente alla presenza di una chiamata a *BSTTreeMinimum*.

Alberi binari di ricerca e liste: confronto

	Liste	BST c. medio	BST c. peggiore
Inserimento	$\Theta(1)$	$\Theta(log(n))$	$\Theta(n)$
Cancellazione	$\Theta(1)$	$\Theta(log(n))$	$\Theta(n)$
Visita	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Ricerca	$\Theta(n)$	$\Theta(log(n))$	$\Theta(n)$
Successore	$\Theta(n)$	$\Theta(log(n))$	$\Theta(n)$
Predecessore	$\Theta(n)$	$\Theta(log(n))$	$\Theta(n)$
Massimo	$\Theta(n)$	$\Theta(log(n))$	$\Theta(n)$
Minimo	$\Theta(n)$	$\Theta(log(n))$	$\Theta(n)$

Conclusione

Gli alberi, come abbiamo detto, sono una struttura dati molto versatile. Gli alberi binari di ricerca sono una delle specializzazioni di questa struttura dati più comuni ed utilizzate. Oltre al loro uso naturale, queste sono utilizzate come base per strutture più complesse, alcune delle quali vedremo più avanti.