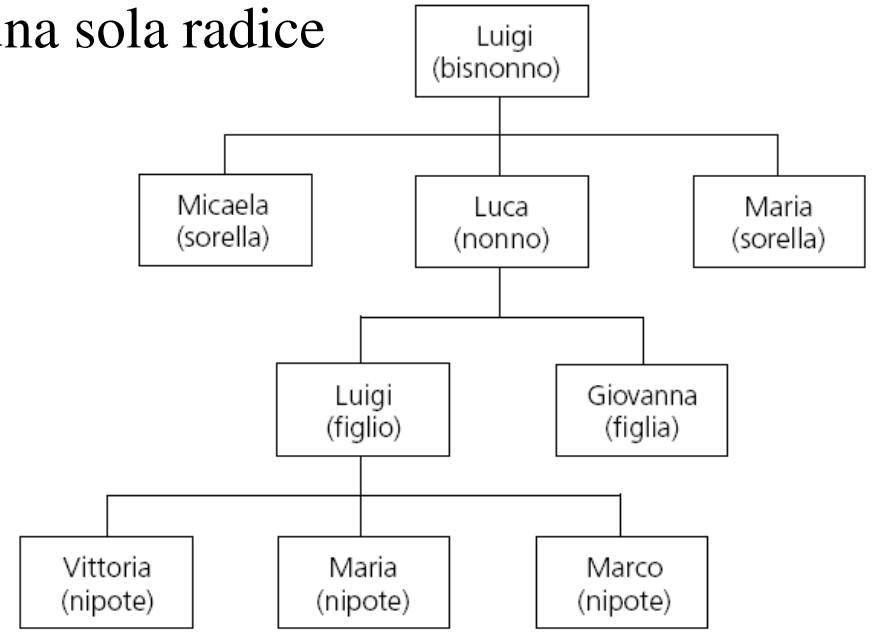
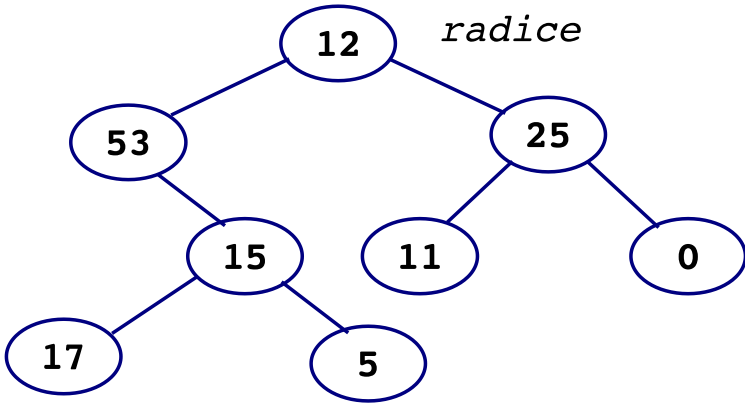
albero

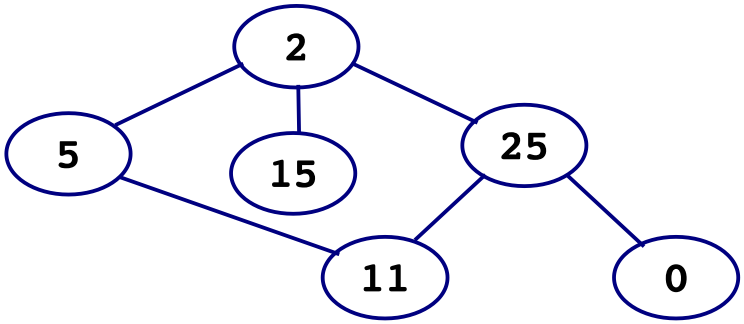
- si dice "grafo diretto" un insieme di nodi legati "a due a due" da archi direzionati (puntatori)
- un *albero* è un grafo diretto in cui ogni nodo può avere un solo arco entrante ed un qualunque numero di archi uscenti
- se un nodo non ha archi uscenti si dice "**foglia**"
- se un nodo non ha archi entranti si dice "**radice**"
- poiché in un albero non ci sono nodi con due o più archi entranti, per ogni albero vi deve essere una ed una sola radice



Esempi



Albero di interi

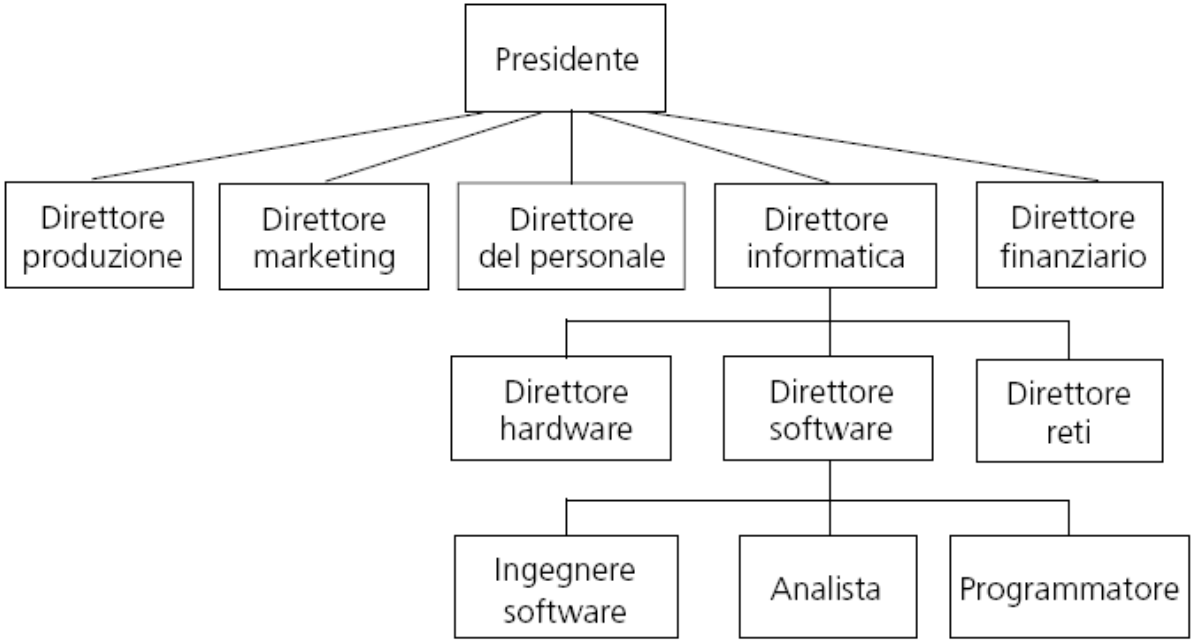


Struttura che non è un albero!

Alberi

- Generalizzazione delle liste.
  - Ogni elemento ha più di un solo successore.
  - Utili per rappresentare partizioni ricorsive di insiemi e strutture gerarchiche

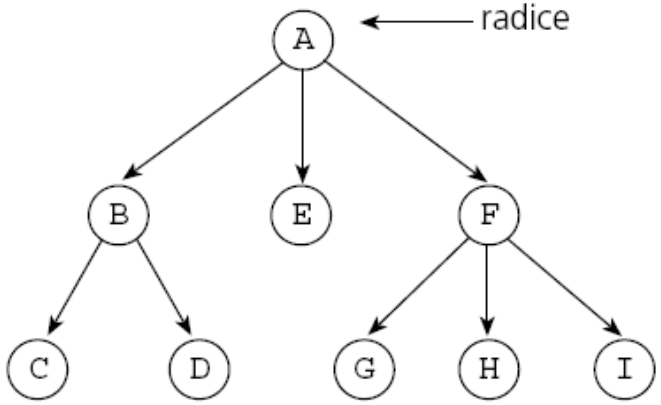
alberi e strutture gerarchiche



- il nodo da cui un arco parte si dice *padre*, un nodo a cui questo arriva si dice *figlio*
- due nodi con lo stesso padre sono detti fratelli
- da ogni nodo non-foglia di un albero si dirama un sottoalbero, quindi si intuisce la natura ricorsiva di questa struttura dati

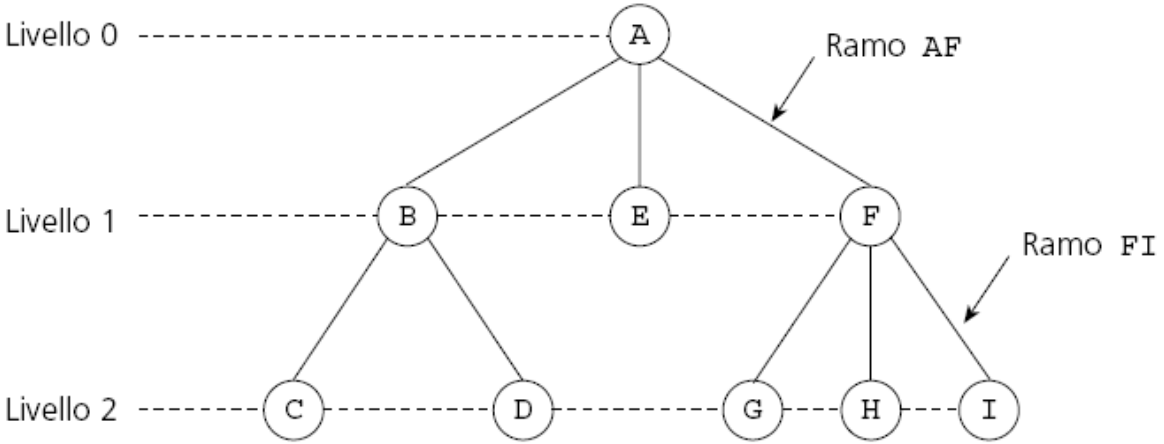
alberi come strutture ricorsive

- da ogni nodo non-foglia di un albero si dirama un sottoalbero, quindi si intuisce la natura ricorsiva di questa struttura dati



- dato un nodo, i nodi che appartengono al suo sottoalbero si dicono suoi *discendenti*
- dato un qualunque nodo, i nodi che si trovano nel *cammino* dalla radice ad esso sono i suoi *ascendenti* (per esempio, B ed A sono ascendenti di C)
- un albero è un insieme di nodi che:
  - o è vuoto
  - oppure ha un nodo denominato *radice* da cui discendono zero o più sottoalberi che sono essi stessi alberi

cammini e livelli



Padri: A, B, F	Foglie: C, D, E, G, H, I
Figli: B, E, F, C, D, G, H, I	Nodi interni: B, F
Fratelli: {B, E, F}, {C, D}, {G, H, I}	

- il **livello** di un nodo è la sua distanza dalla radice
- la radice ha livello 0, i suoi figli livello 1, i suoi nipoti livello 2 e così via
- i fratelli hanno lo stesso livello ma non tutti i nodi dello stesso livello sono fratelli
- La **profondità** di un albero è la lunghezza del cammino più lungo dalla radice ad una foglia

Profondità di un albero

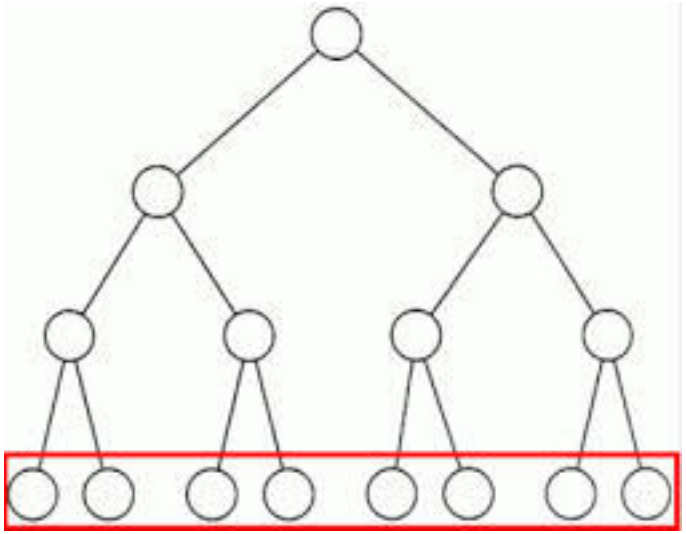
- Definita ricorsivamente
  - La radice ha profondità 0 i figli (della radice) profondità 1, etc.

Profondità (u)

```
p=0;
WHILE (u.padre != null) {
 p=p +1;
 u=u.padre;
}
```

Alberi equilibrati

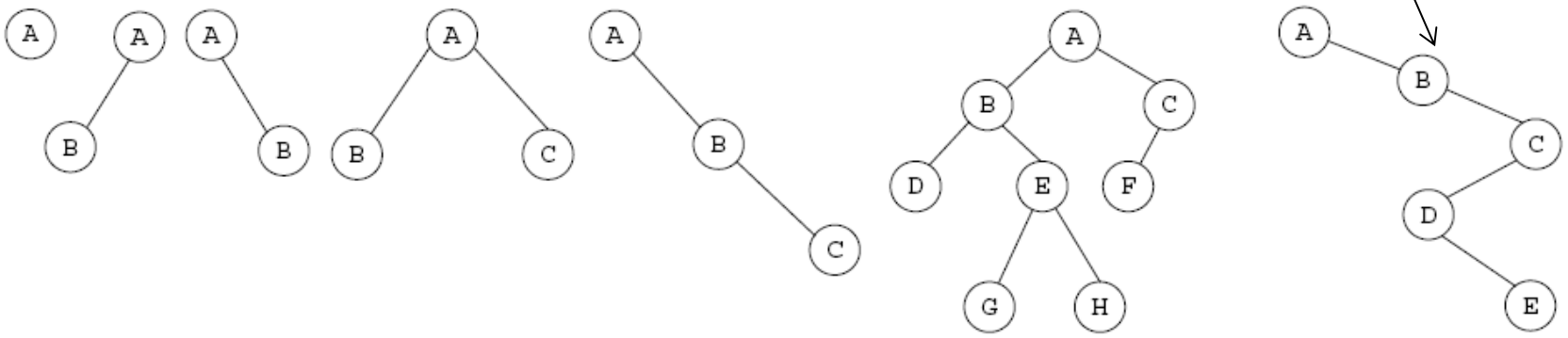
- Un albero di profondità h è equilibrato (o bilanciato) se, dato k numero massimo di figli per nodo, ogni nodo interno (inclusa la radice) ha esattamente k figli.



albero binario

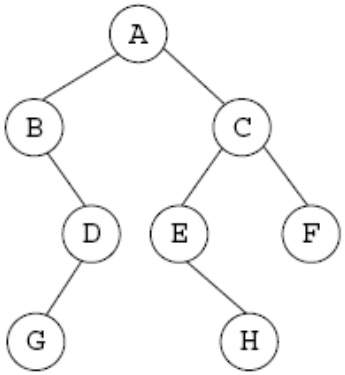
- un albero è *binario* se ogni nodo non ha più di due figli (sottoalberi), il sinistro ed il destro

Albero degenere!

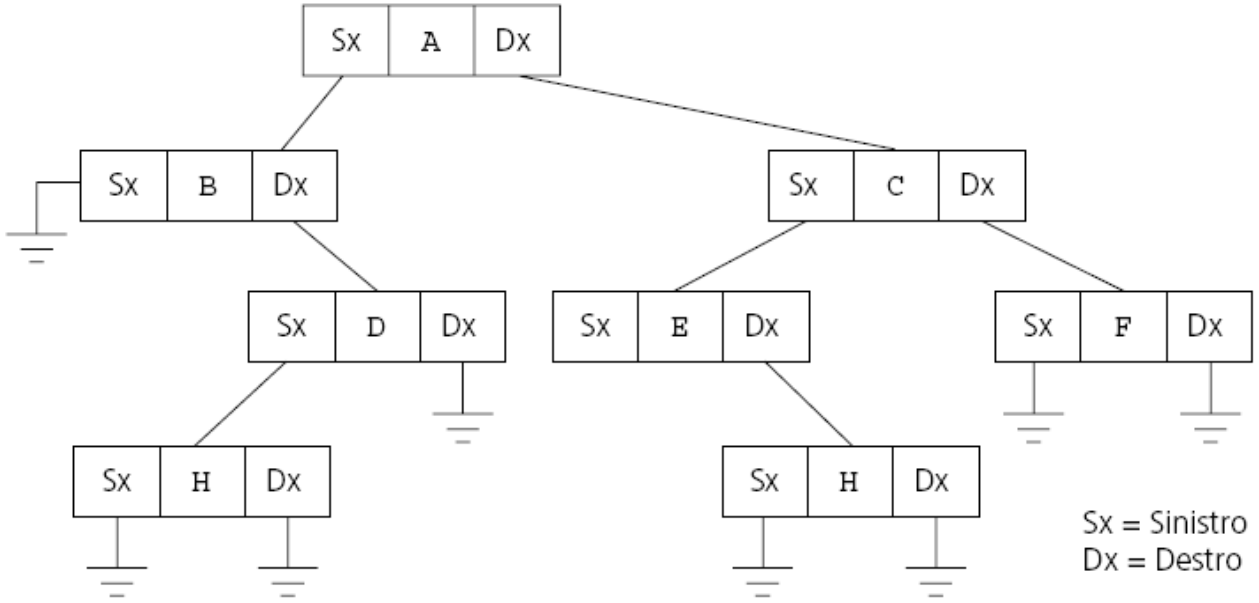


- Ad ogni livello n un albero binario può contenere (al più) 2^n nodi
- Il numero totale di nodi di un albero (incluse le foglie) di profondità n è al massimo $2^{(n+1)}-1$

implementazione di un albero



(a) Albero



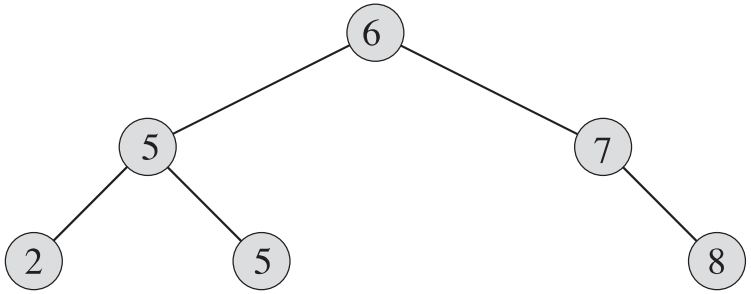
(b) Struttura

visita di un albero

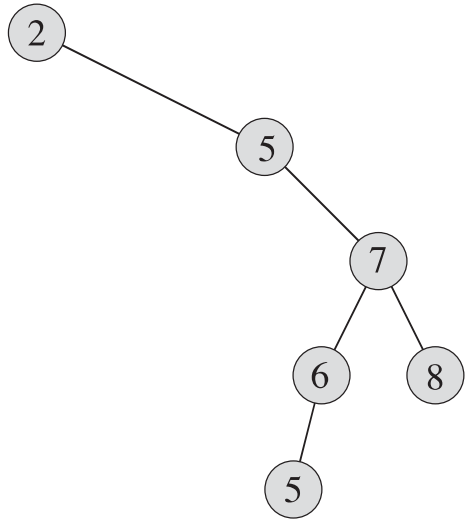
- partendo dalla radice, in linea generale un albero può essere visitato o "in profondità" oppure "in ampiezza"
- nella visita in profondità, dopo il nodo corrente viene esaminato un suo figlio, quale dipende dal particolare algoritmo; se non c'è alcun figlio si ritorna dal padre ("backtraking" semplice) verso un eventuale altro figlio e così via; così facendo tutti i discendenti d'ogni nodo vengono visitati prima dei suoi eventuali fratelli o pari livello
- nella visita in ampiezza, dopo il nodo corrente viene visitato un altro nodo di pari livello; così facendo tutti i nodi di un livello verranno visitati prima di tutti i nodi del livello successivo

Albero binario di ricerca

- Proprietà fondamentale:
  - x nodo generico. Se y è un nodo nel sottoalbero sinistro di radice x allora y.valore ≤ x.valore. Altrimenti y.valore ≥ x.valore



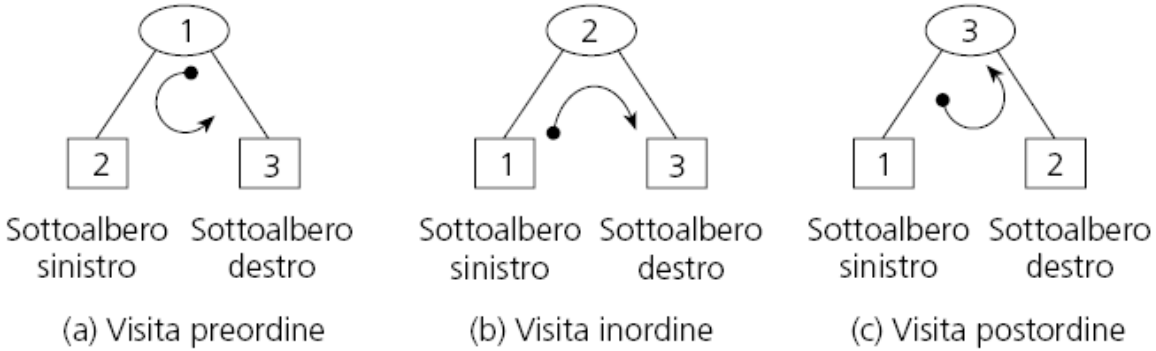
(a)



(b)

visita di un albero binario

- esistono tre strategie di visita notevoli



- la visita *preorder* visita prima la radice, quindi il sottoalbero sinistro e da ultimo quello destro
- la visita *inorder* processa prima il sottoalbero sinistro, quindi la radice ed infine il sottoalbero destro
- la visita *postorder* processa prima il sottoalbero sinistro, poi quello destro ed infine la radice

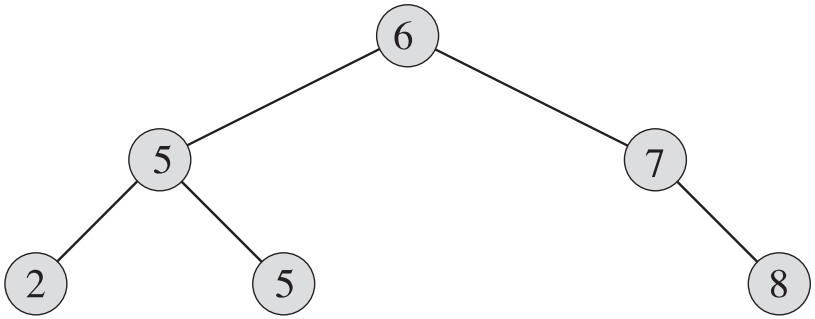
visita di un albero binario in C++

```
void PreOrdine(Nodo* p)
{
   if (p)
   {
      cout << p -> valore << ";     // visita la radice
      PreOrdine(p -> sinistro);    // visita il sottoalbero sinistro
      PreOrdine(p -> destro);      // visita il sottoalbero destro
   }
}
```

```
void InOrdine(Nodo *p)
{
   if (p)
   {
      InOrdine(p -> sinistro);    // visita il sottoalbero sinistro
      cout << p -> valore << '';    // visita la radice
      InOrdine(p -> destro);      // visita il sottoalbero destro
   }
}
```

```
void PostOrdine(Nodo *p)
{
   if (p)
   {
      PostOrdine(p -> sinistro);    // visita il sottoalbero sinistro
      PostOrdine(p -> destro);      // visita il sottoalbero destro
      cout << p -> valore << '';      // visita la radice
   }
}
```

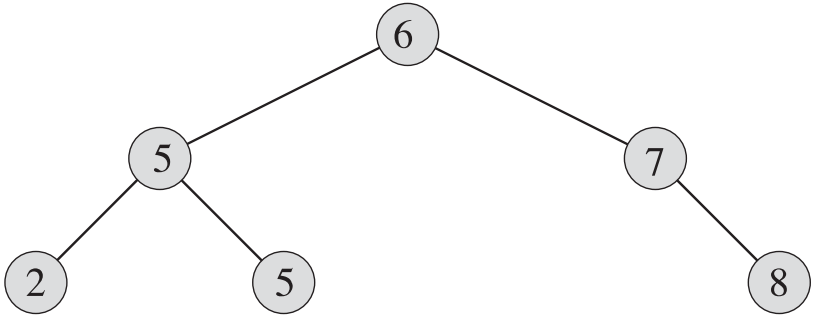
Esempio



- Visita Inorder:
  - Visitiamo prima il sottoalbero sinistro, quindi la radice ed infine il sottoalbero destro.

- Nel caso il questione l'ordine di lettura sarebbe 2, 5, 5, 6, 7, 8

Costo della visita




```
void InOrdine(Nodo *p)
{
   if (p!=NULL)
   {
     InOrdine(p -> sinistro);
     cout << p -> valore << '';
     InOrdine(p -> destro);
   }
}
```

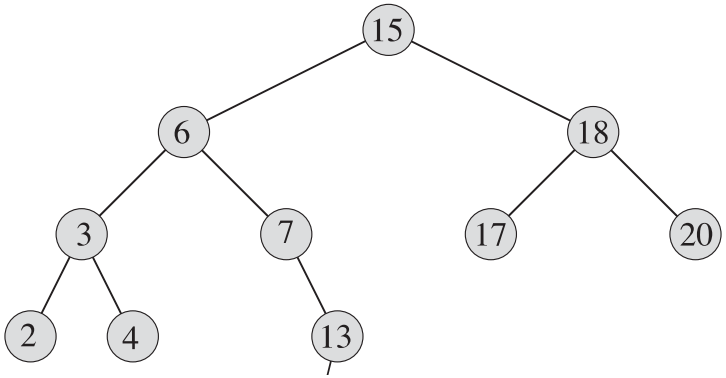
Complessità: T(n)=Θ(n)

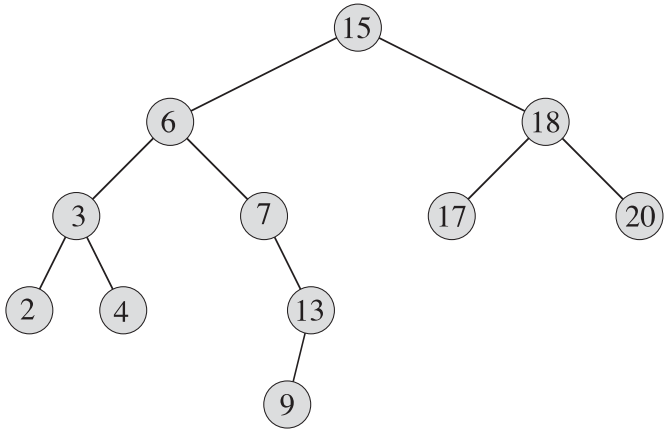
- Devono essere fatti almeno n passi, dunque T(n)= Ω(n)
- Rimane da provare che T(n)=O(n)

- T(0)=c (c>0) (la procedura deve testare se il puntatore è diverso da NULL)
- Guardiamo il caso n>0

Inserimento in un albero binario di ricerca

- I nuovi elementi vengono sempre inseriti come nuove foglie





Inserimento in un albero binario di ricerca

```
Insert(T,elemento)
1. x=T.root; y=NULL;
2. while (x!= NULL)
3.       y=x;
4.       if (elemento < x.val)    x=x.left;
5.     else x=x.right;
6. new nodo;
7. nodo.padre=y; nodo.val=elemento;
8. if (y==NULL) // L'albero è vuoto
9.     T.root=nodo;
10. else if (nodo.val < y.val) y.left=nodo;
11. else y.right=nodo;
```

Ricerca in un albero binario di ricerca

```
Ricerca(T,elemento)      // T è tipicamente un puntatore alla radice
1. if (T== NULL) or (elemento==T.val))  return T;
2. if (elemento < T.val) return Ricerca(T.left, elemento)
3. else return Ricerca(T.right, elemento)
```

Complessità: T(n)=O(h) (h profondità dell'albero)

Ricerca iterativa

```
Ricerca(T,elemento)  // T tipicamente è un puntatore alla radice
1. x=T;
2. while ((x!= NULL) and (elemento!=x.val))
3.     if (elemento < x.val)  x=x.left;
4.     else x=x.right;
5. return x
```

Massimo e minimo

Massimo(T) *// T è tipicamente un puntatore alla radice*

1. x=T;
2. **while** (x.right!= NULL) x=x.right;
3. **return** x

Minimo(T) *// T è tipicamente un puntatore alla radice*

1. x=T;
2. **while** (x.left!= NULL) x=x.left;
3. **return** x

Complessità: T(n)=O(h) (h profondità dell'albero)

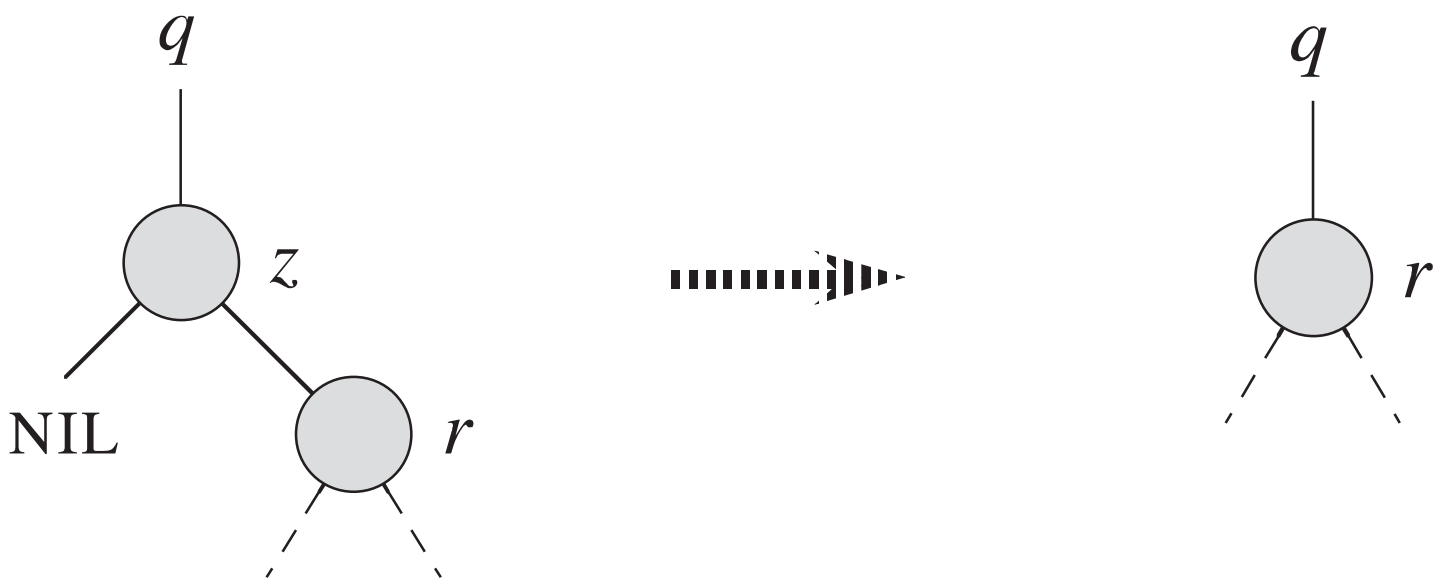
Cancellazione di un nodo z

- Procedura più complicata delle precedenti

- Tre casi da considerare
  1. z non ha figli
  2. z ha un solo figlio
  3. z ha due figli

- Il caso più complesso da gestire è il 3
- Nel caso in cui z non abbia figli la procedura è banale:
  - Eliminiamo z e modifichiamo il padre in modo che il puntatore a z diventi un puntatore a NULL.

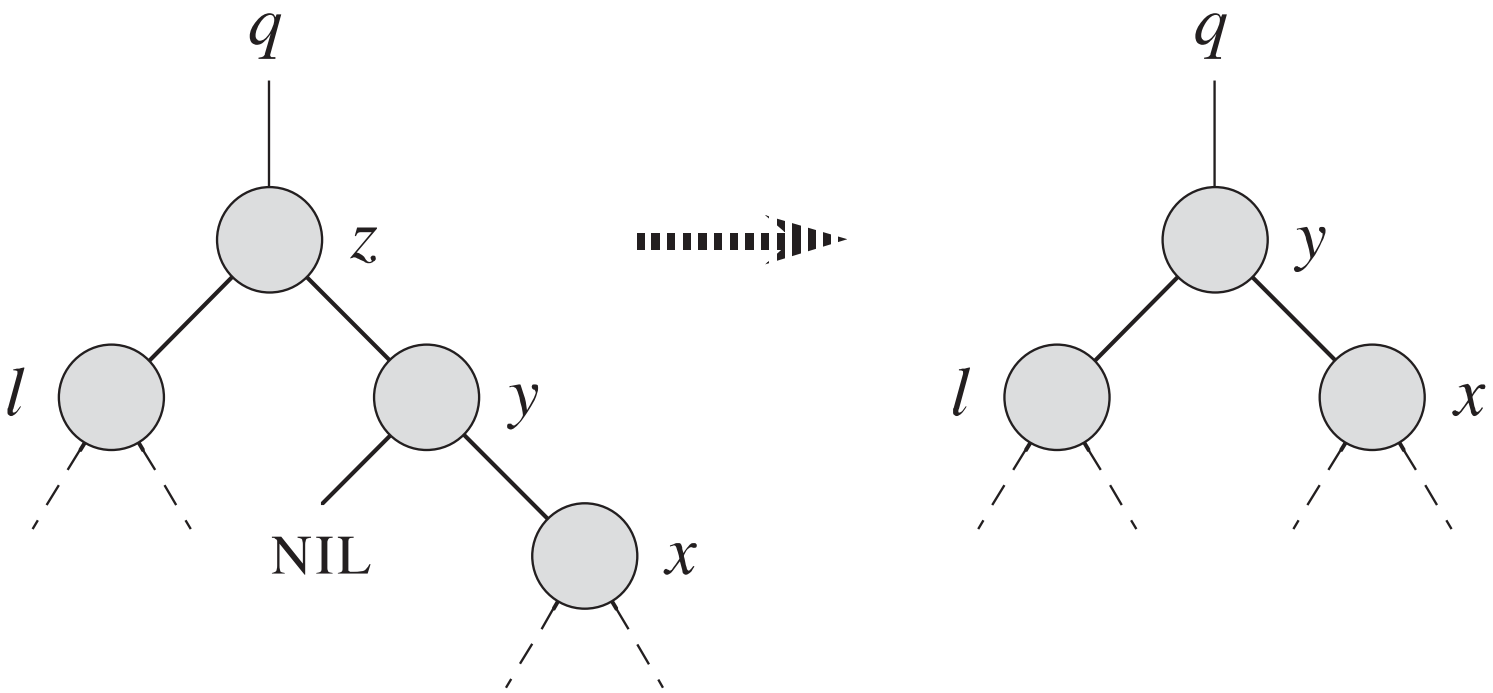
Caso 2: z ha un solo figlio



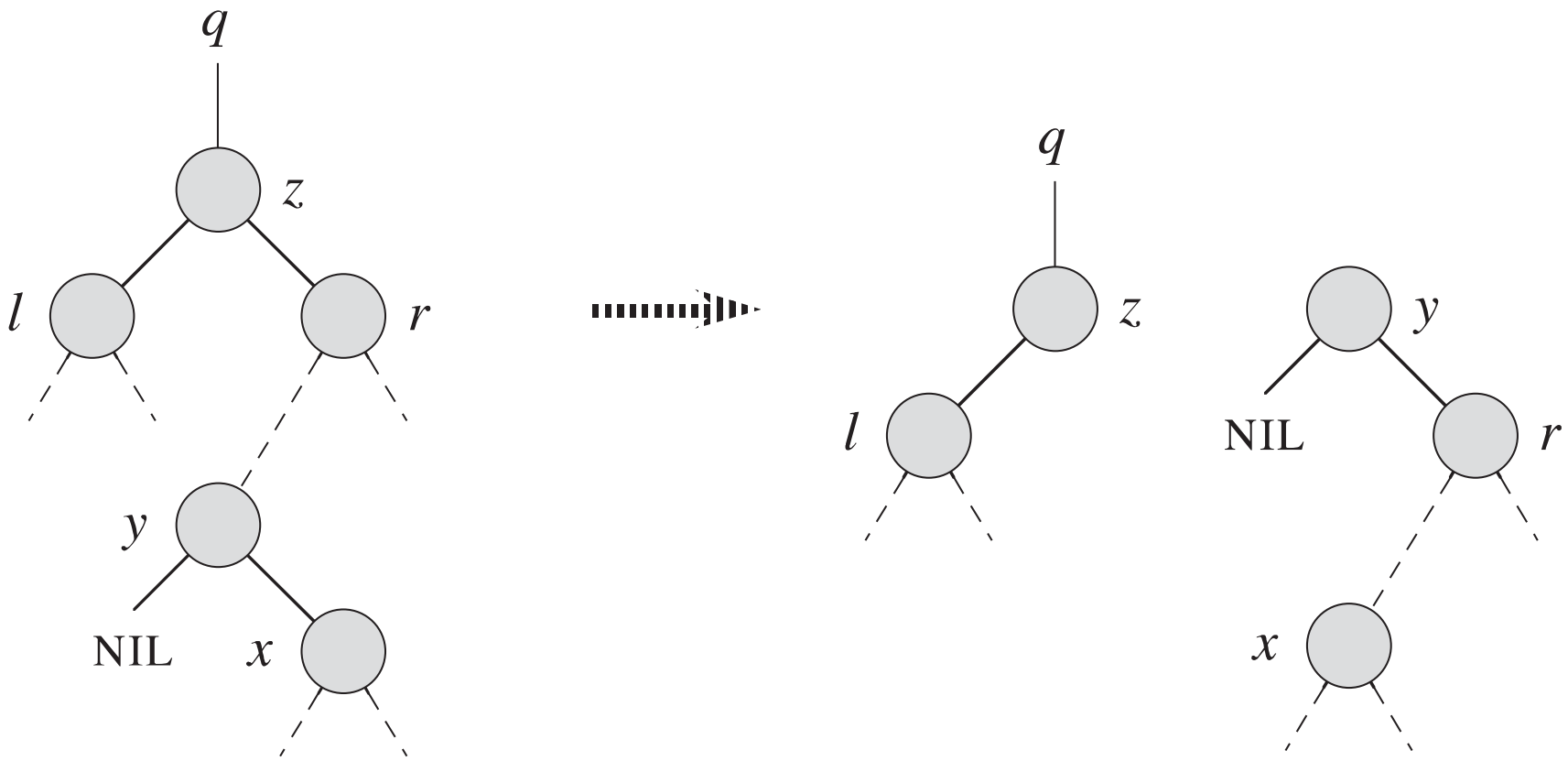
Caso 3: z ha due figli

- Cerchiamo il successore y di z
  - Si troverà certamente nel sottoalbero destro di radice z
- y prende la posizione di z nell'albero
- La rimanente parte del sottoalbero destro (di z) diventa il nuovo albero destro di y
- Ulteriore complicazione che succede se y non è figlio di z?

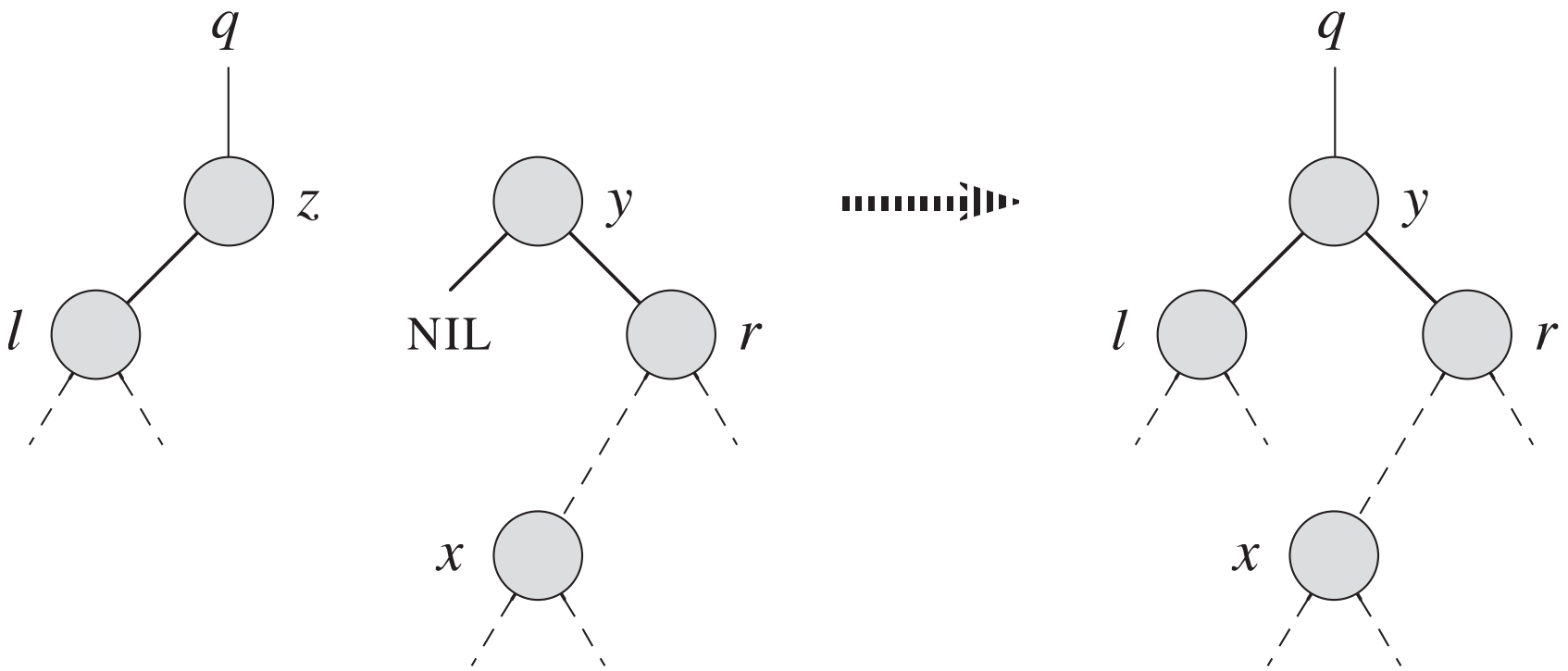
Caso 3a: y è figlio di z



Caso 3 più generale



Caso 3 più generale



Cancellazione: procedura “Trapianta”

- Permette di spostare sottoalberi
  - Rimpiazza un sottoalbero di radice u con un altro di radice v

```
Trapianta(root,u,v)
1. if u.padre==NULL root=v;              // u è proprio la radice
2. else if (u==u.padre.left)             // u è figlio sinistro
3.         u.padre.left=v;
4. else u.padre.right=v;
5. if (v!=NULL)                          // aggiorniamo il padre di v
6.   v.padre=u.padre;
```

Cancellazione

```
Delete(root,z)
1. if (z.left==NULL) Trapianta(T,z,z.right);  // z non ha figlio sinistro
2. else if (z.right==NULL)
3.       Trapianta(T,z,z.left);
4. else
5.     y=Minimo(z.right);
6.     if (y.padre!=z)                          // Caso 3 più generale
7.           Trapianta(T,y,y.right);
8.           y.right=z.right;
9.           y.right.padre=y;
10.    Trapianta(T,z,y);
11.    y.left=z.left;
12.    y.left.padre=y;
```

Successore di x

- Due casi da considerare
  - Il sottoalbero destro di x è non vuoto
    - Caso banale!
  - Il sottoalbero destro di x è vuoto
    - Minimo antenato di x il cui figlio sinistro è anche un antenato di x
    - Ogni nodo è antenato di se stesso

Successore

```
Successore(x)
1.  if (x.right!=NULL) return Minimo(x.right);      // Caso banale
2.  y=x.padre;
3.  while (y.right!=NULL) && (x==y.right)
4.        x=y;
5.        y=y.padre;
6.  return y;
```