DIPARTIMENTO DI MATEMATICA E INFORMATICA

UNIVERSITÀ degli STUDI di CATANIA

Introduzione agli Alberi ed ai BST

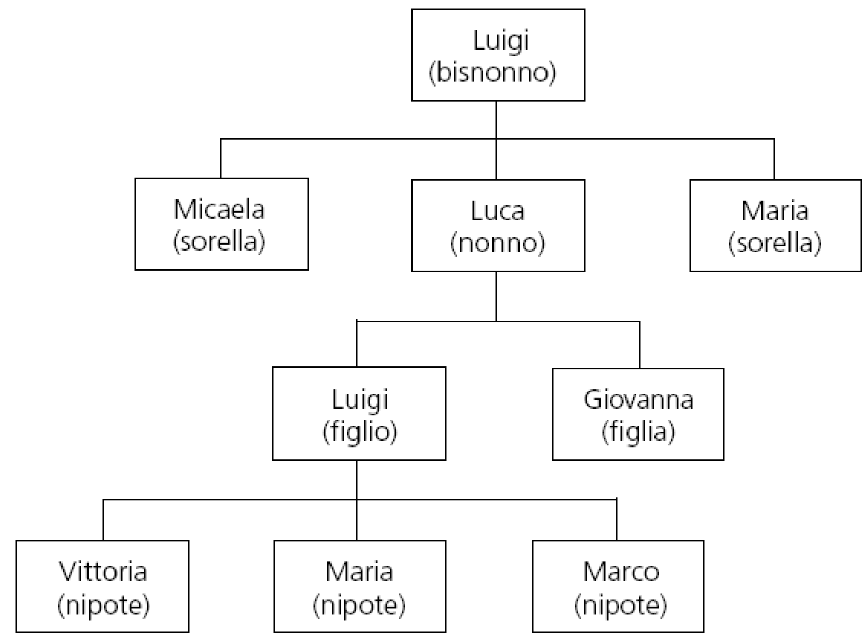
Alessandro Ortis

ortis@dmi.unict.it
www.dmi.unict.it/ortis/

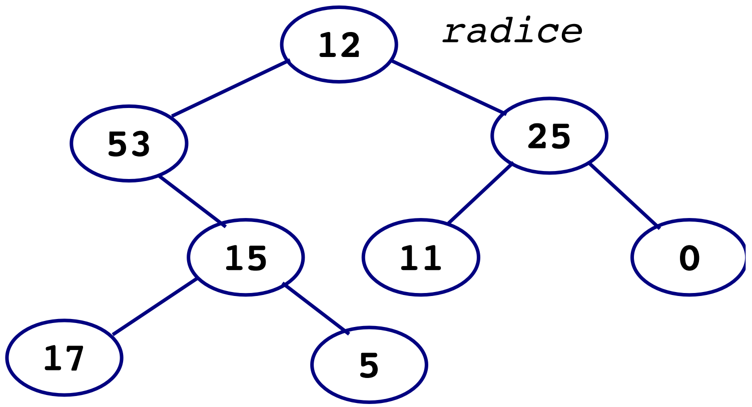
Albero

- si dice "grafo diretto" un insieme di nodi collegati mediante archi direzionati.

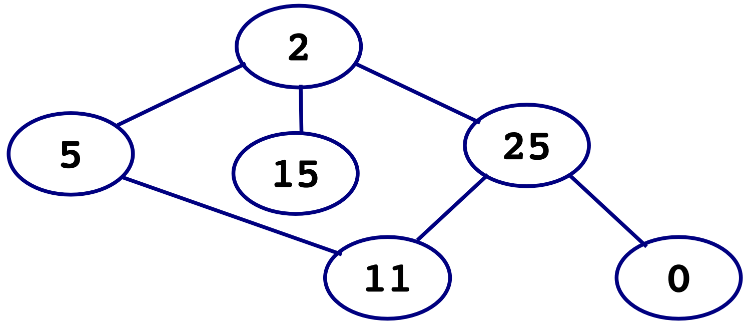
- un *albero* è un grafo diretto in cui ogni nodo può avere un solo arco entrante ed un qualunque numero di archi uscenti
- se un nodo non ha archi uscenti si dice "**foglia**"
- se un nodo non ha archi entranti si dice "**radice**"
- poiché in un albero non ci sono nodi con due o più archi entranti,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



Esempi



Albero di interi

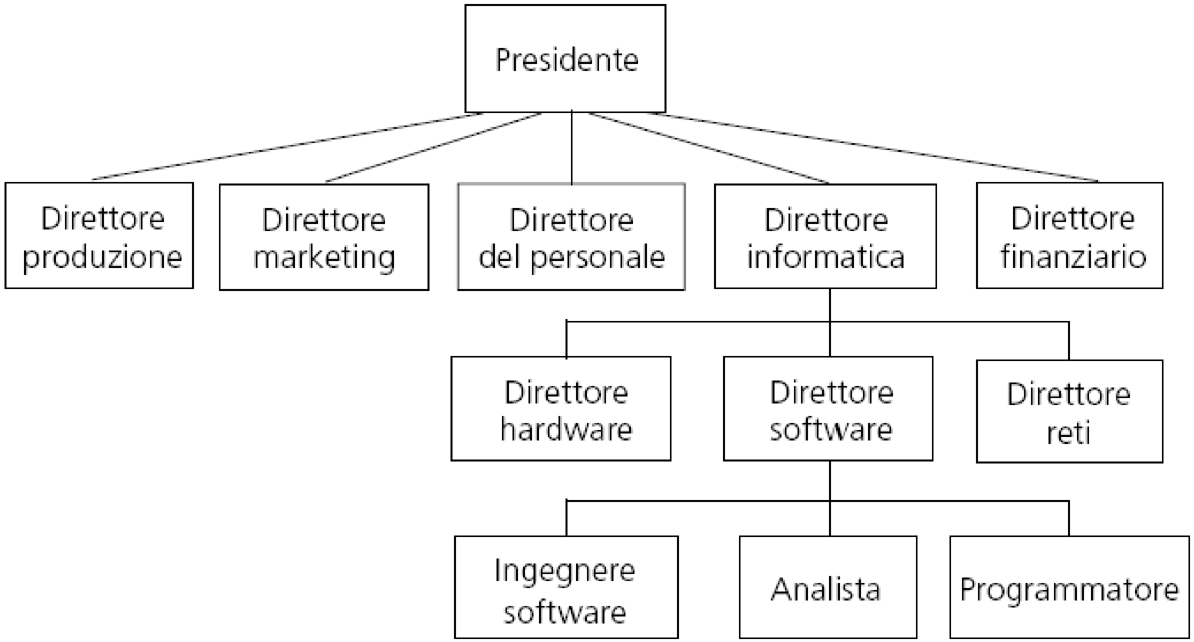


Struttura che non è un albero!

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

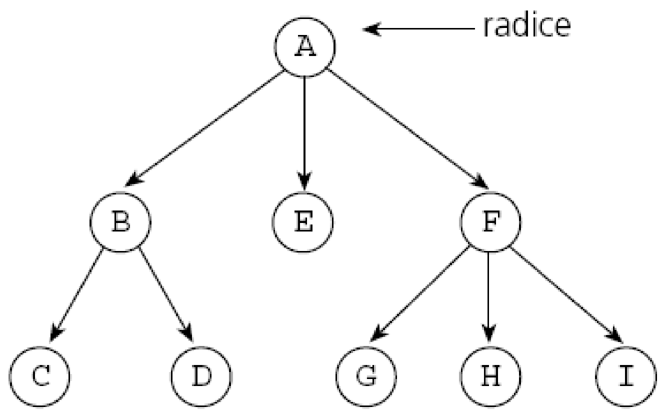


- il nodo da cui un arco parte si dice *padre*, un nodo a cui questo arriva si dice *figlio*
- due nodi con lo stesso padre sono detti fratelli
- da ogni nodo non-foglia di un albero si dirama un sottoalbero, quindi si intuisce la natura ricorsiva di questa struttura dati

Alberi come strutture ricorsive

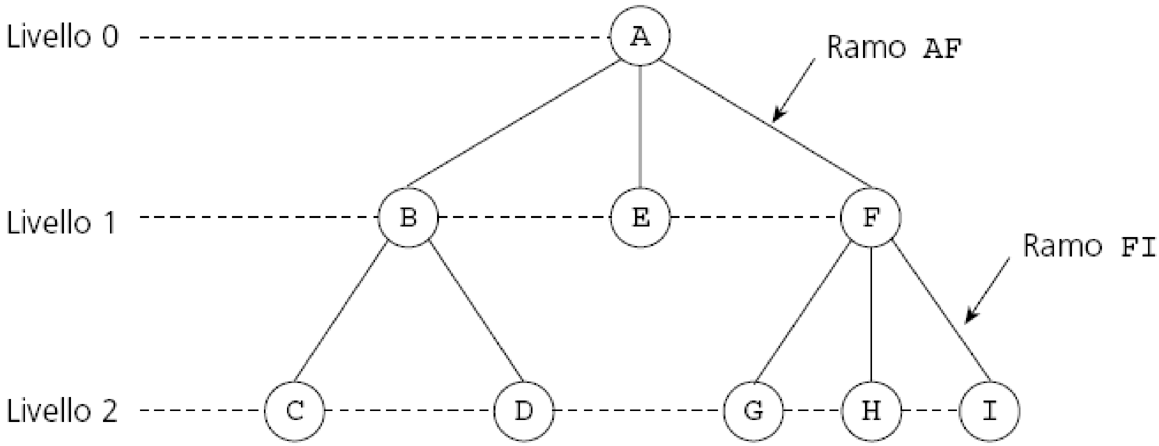
Definizione ricorsiva: un albero è un insieme di nodi che

- è vuoto
- oppure ha un nodo denominato *radice* da cui discendono zero o più sottoalberi che sono essi stessi alberi



- 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)

Cammini e livelli



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

- 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 nodo

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

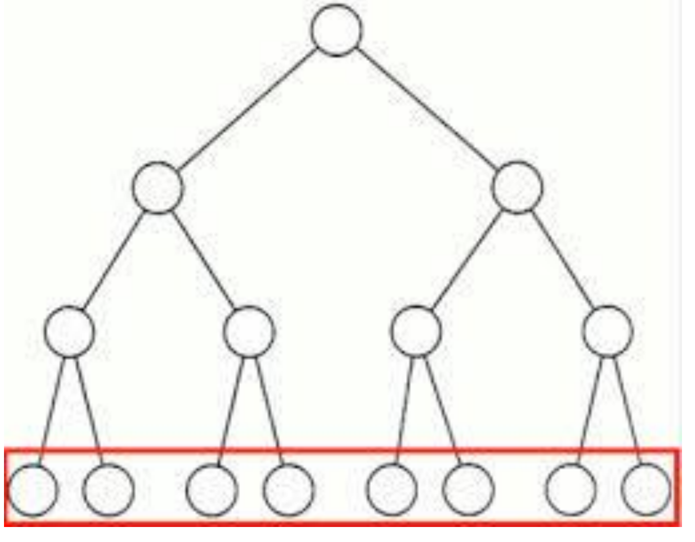
Profondità (u)

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

Alberi equilibrati

Un albero di profondità h è equilibrato (o bilanciato) se,

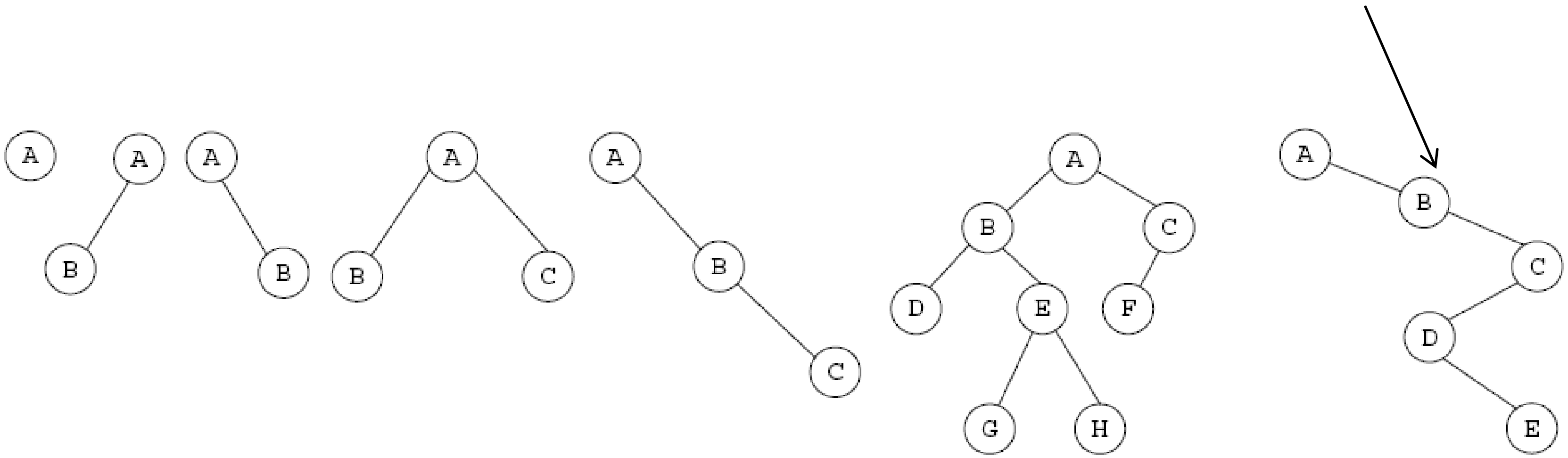
1. Tutte le foglie si trovano allo stesso livello.
2. Dato k numero massimo di figli per nodo, ogni nodo interno (inclusa la radice) ha esattamente k figli.



Albero binario

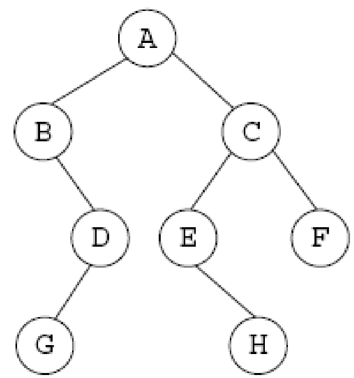
- un albero è *binario* se ogni nodo non ha più di due figli (sottoalberi), il sinistro ed il destro

Albero degenere!

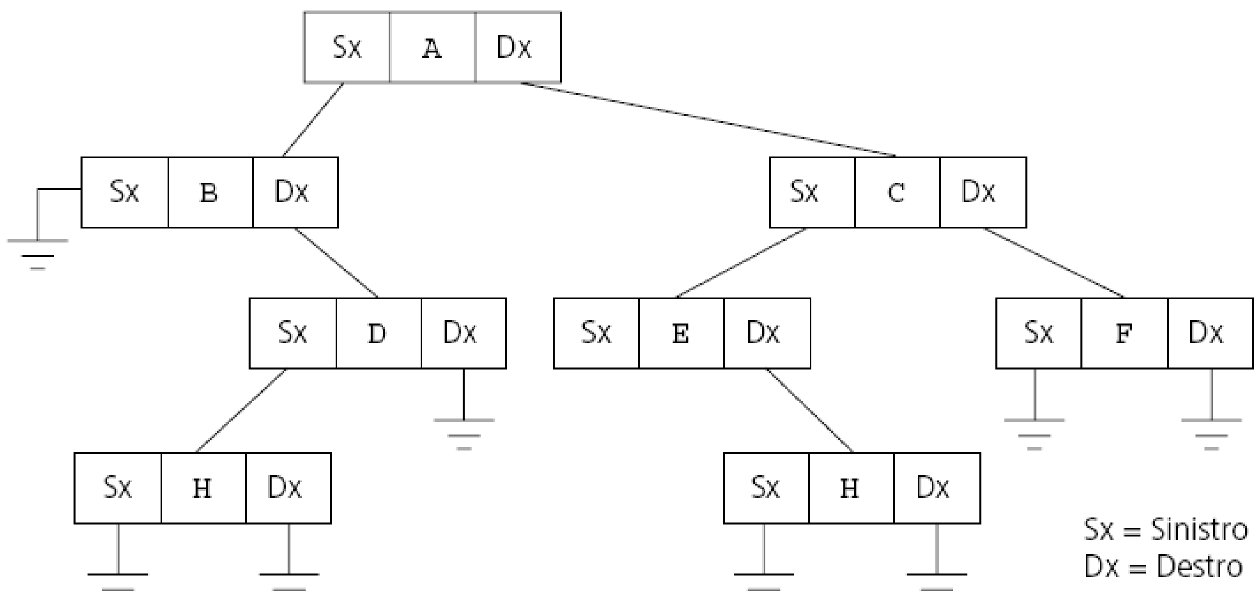


- Ad ogni livello n un albero binario può contenere (al più) 2^n nodi
- Il numero totale di nodi di un albero (incluse le foglie) di profondità n è al massimo $2^{(n+1)}-1$

Implementazione di un albero



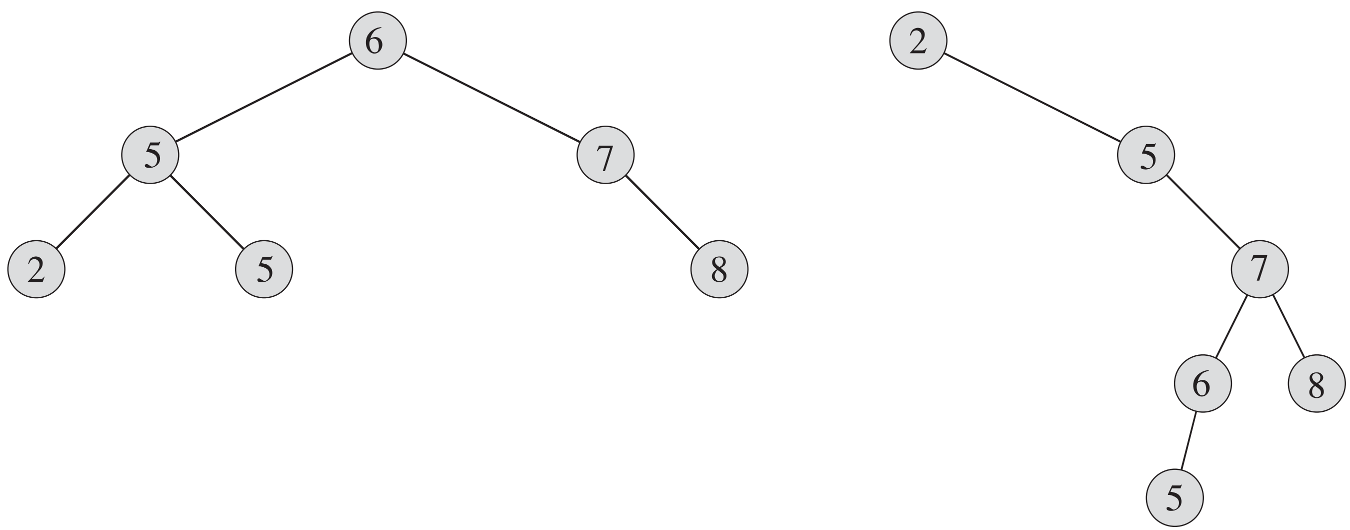
(a) Albero



(b) Struttura

Albero binario di ricerca

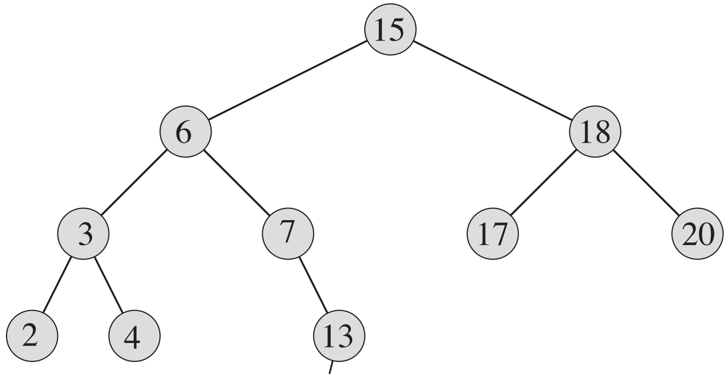
- Proprietà fondamentale:
  - x nodo generico. Se y è un nodo nel sottoalbero sinistro di radice x allora y.valore < x.valore. Altrimenti y.valore ≥ x.valore

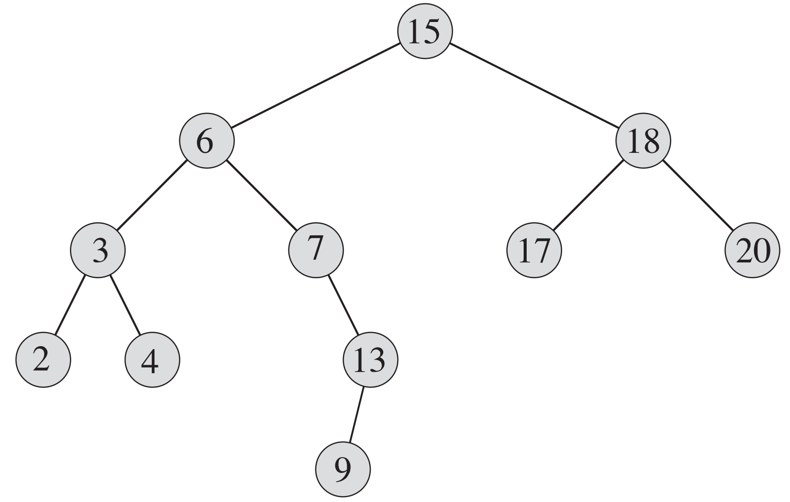


(a) (b)

Inserimento in un albero binario di ricerca

I nuovi elementi vengono sempre inseriti come nuove foglie.
In che punto inseriremo l'elemento 9?



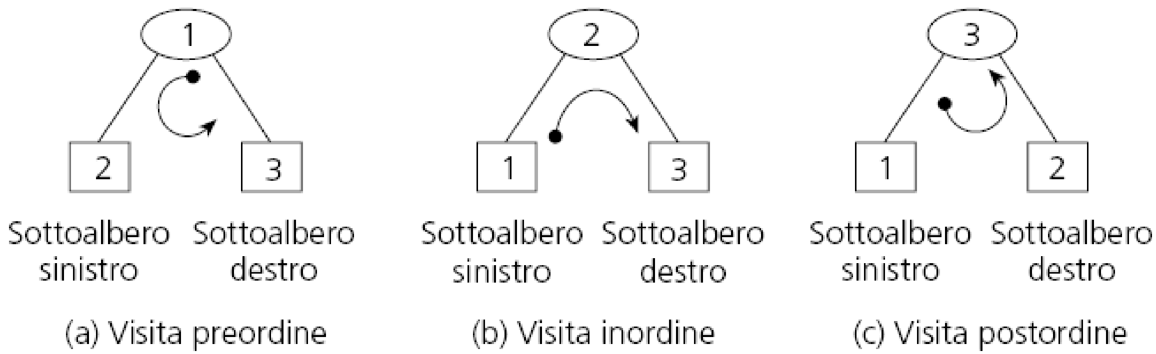


Inserimento in un albero binario di ricerca

```
Insert(T, elemento)
    x=T.root;
    y=NULL;
    while (x!= NULL)
        y=x;                    // y serve a mantenere il padre di x
        if (elemento < x.val)   // confronto con la chiave di x
            then x=x.left;
            else  x=x.right;
    new nodo;        // x è occupa la posizione dove inserire il nodo
    nodo.padre=y; nodo.val=elemento;
    if (y==NULL) // L'albero è vuoto
            T.root=nodo;
    else if (nodo.val < y.val) y.left=nodo;
    else y.right=nodo;
```

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

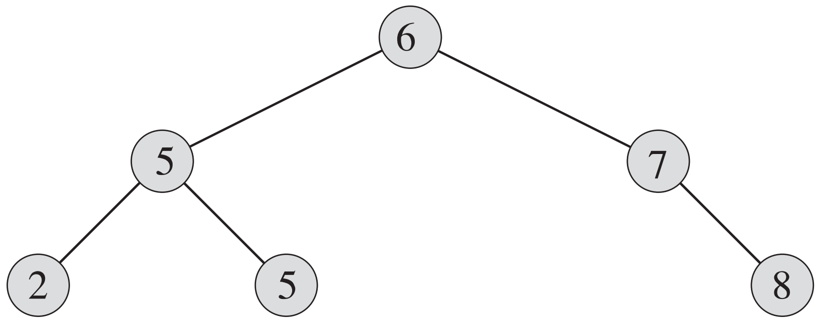
Visita di un albero binario in C++

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

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

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

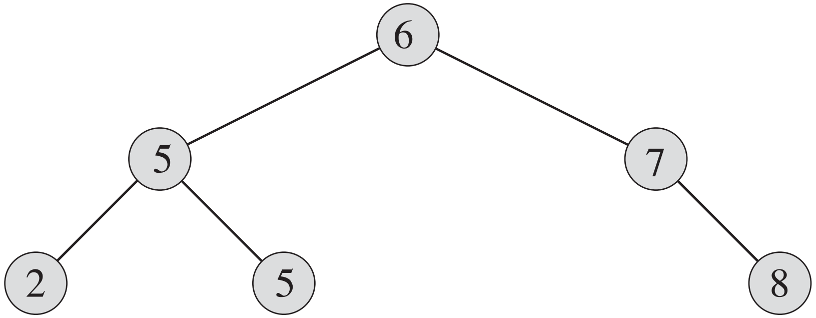
Esempio



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

- Nel caso in questione l’ordine di lettura sarebbe 2, 5, 5, 6, 7, 8

Costo della visita




```
void inOrder(Nodo *p)
{
  if (p!=nullptr)
  {
    inOrder(p -> sinistro);
    cout << p -> valore << ‘ ‘;
    inOrder(p -> destro);
  }
}
```

Complessità: $T(n)=\Theta(n)$

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

- Se $n=0$

  $T(0)=c$ ($c>0$) (la procedura deve testare se il puntatore è diverso da NULL)

- Se $n>0$

  …

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
x=T;
while ((x!= NULL) and (elemento!=x.val))
        if (elemento < x.val)  x=x.left;
        else x=x.right;
return x
```

Massimo e minimo

Massimo(T) // *T è tipicamente un puntatore alla radice*
x=T;
while (x.right!= NULL) x=x.right;
return x

Minimo(T) // *T è tipicamente un puntatore alla radice*
x=T;
while (x.left!= NULL) x=x.left;
return x

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

Successore di x

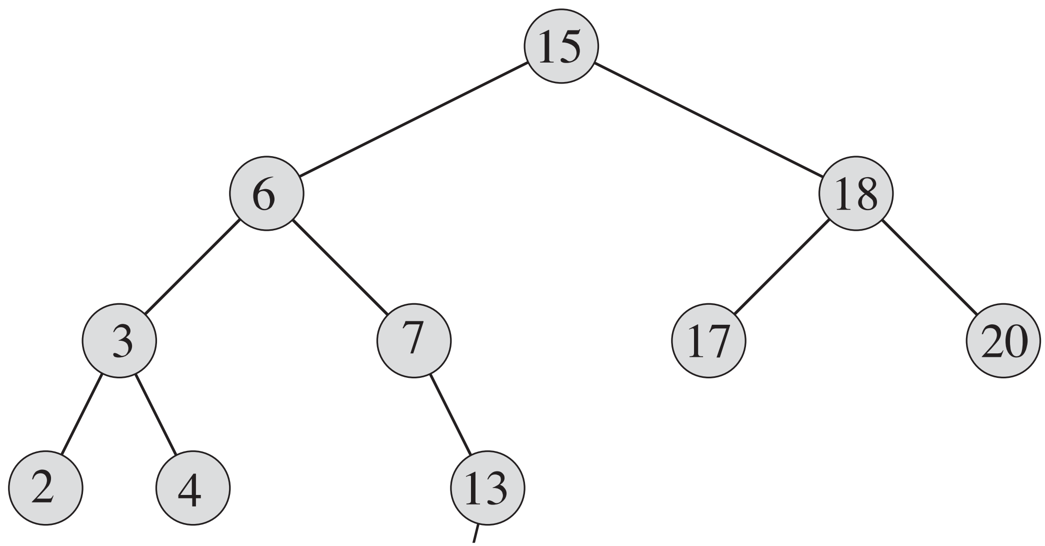
Due casi da considerare

1. Il sottoalbero destro di x è non vuoto: caso banale

2. 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 di x

```
Successore(x)
if (x.right!=NULL)
        return Minimo(x.right);   // Caso banale
y=x.padre;   // risaliamo l'albero fino a trovare un figlio sx di suo padre
while (y!=NULL) && (x==y.right)
        x=y;
        y=y.padre;
return y;
```



Cancellazione di un nodo z

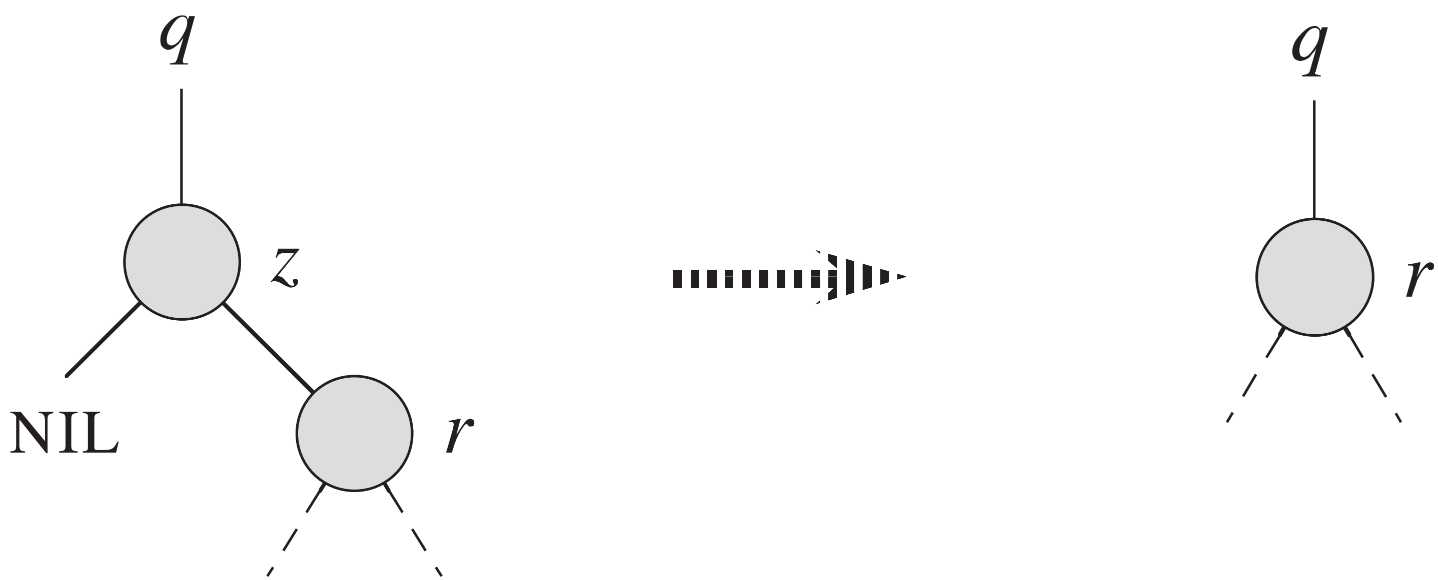
- Procedura più complicata delle precedenti

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

- 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.

- Il caso più complesso da gestire è il 3

Caso 2: z ha un solo figlio



Caso 3: z ha due figli

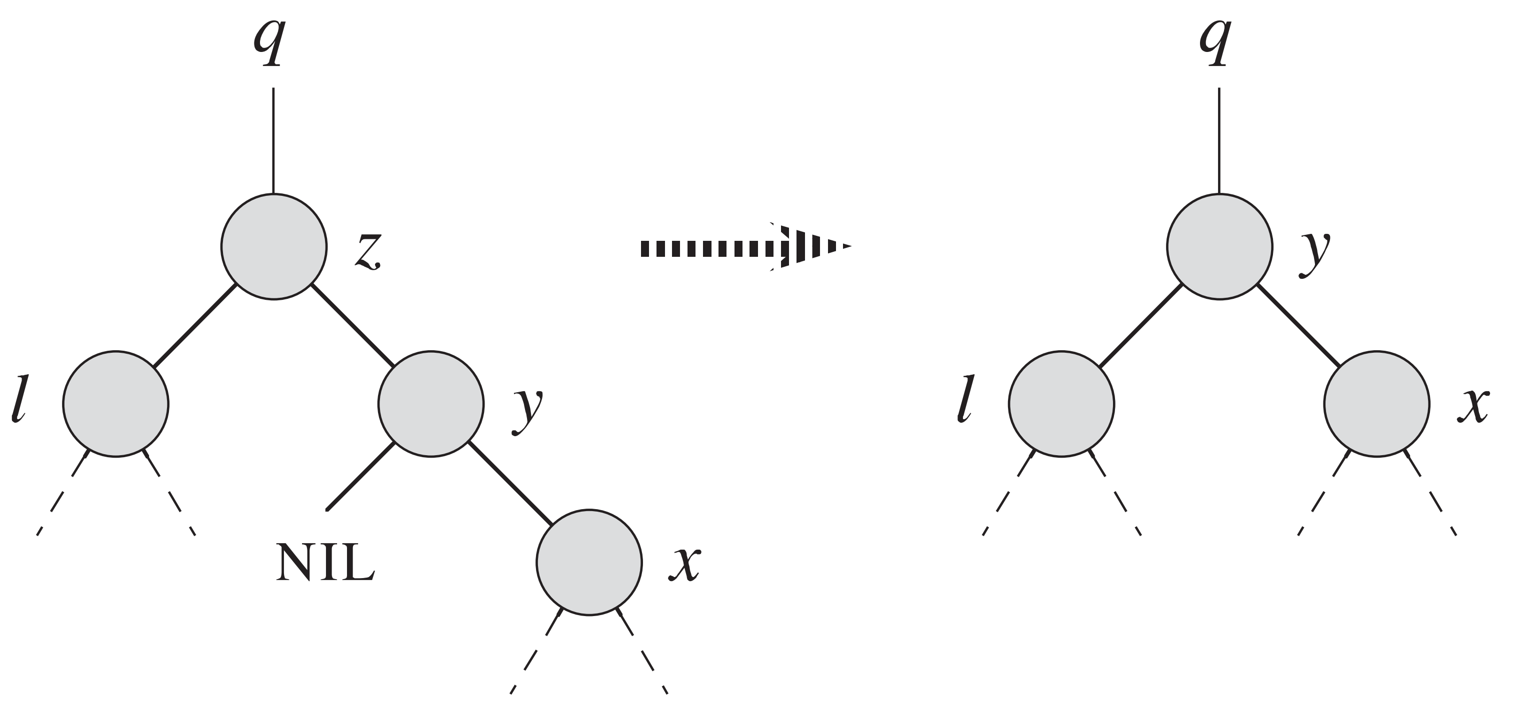
- Cerchiamo il successore y di z
  - Si troverà certamente nel sottoalbero destro di radice z

- y prende la posizione di z nell'albero

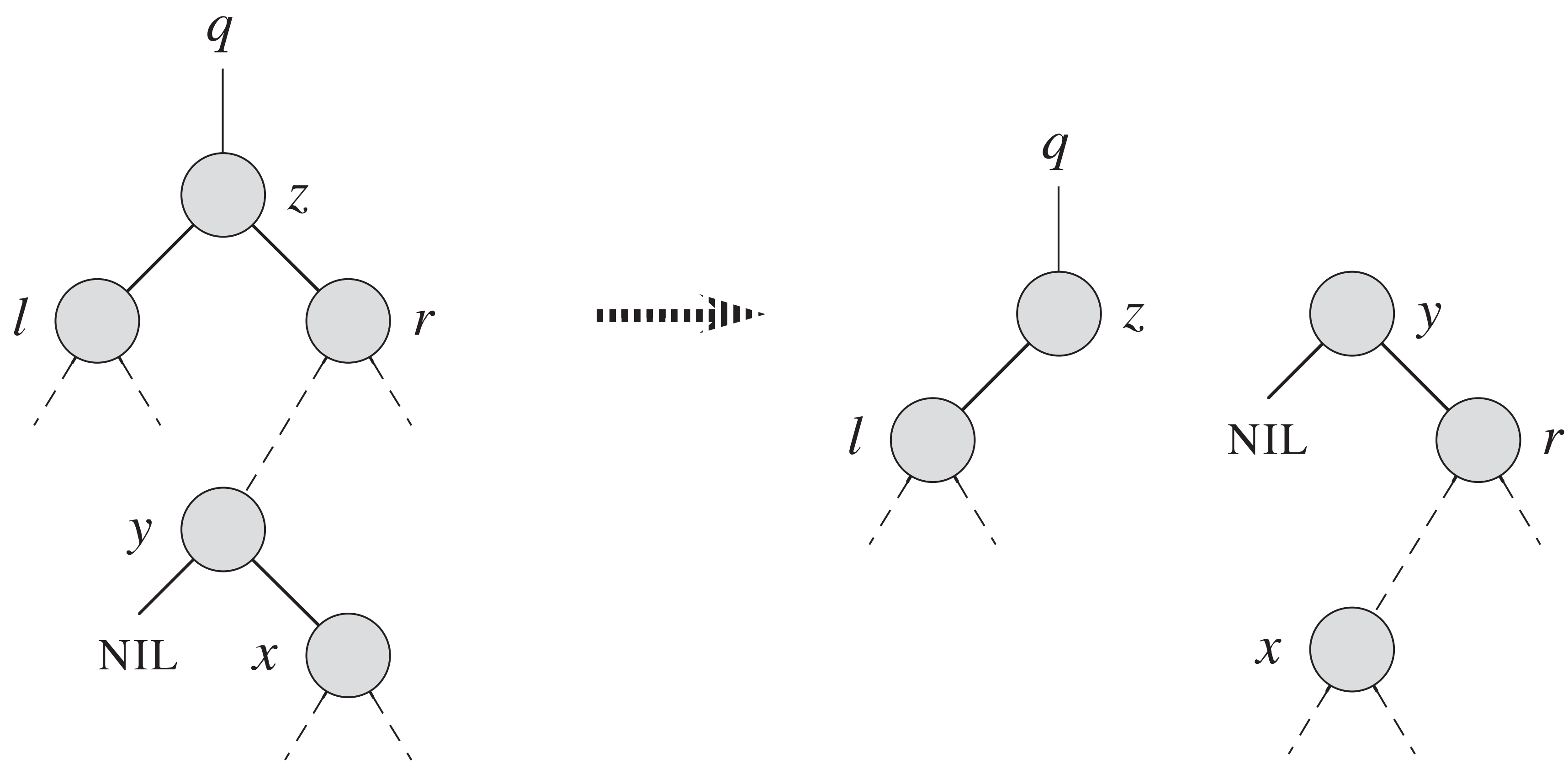
- La rimanente parte del sottoalbero destro (di z) diventa il nuovo albero destro di y

- Ulteriore complicazione: che succede se y non è figlio di z?

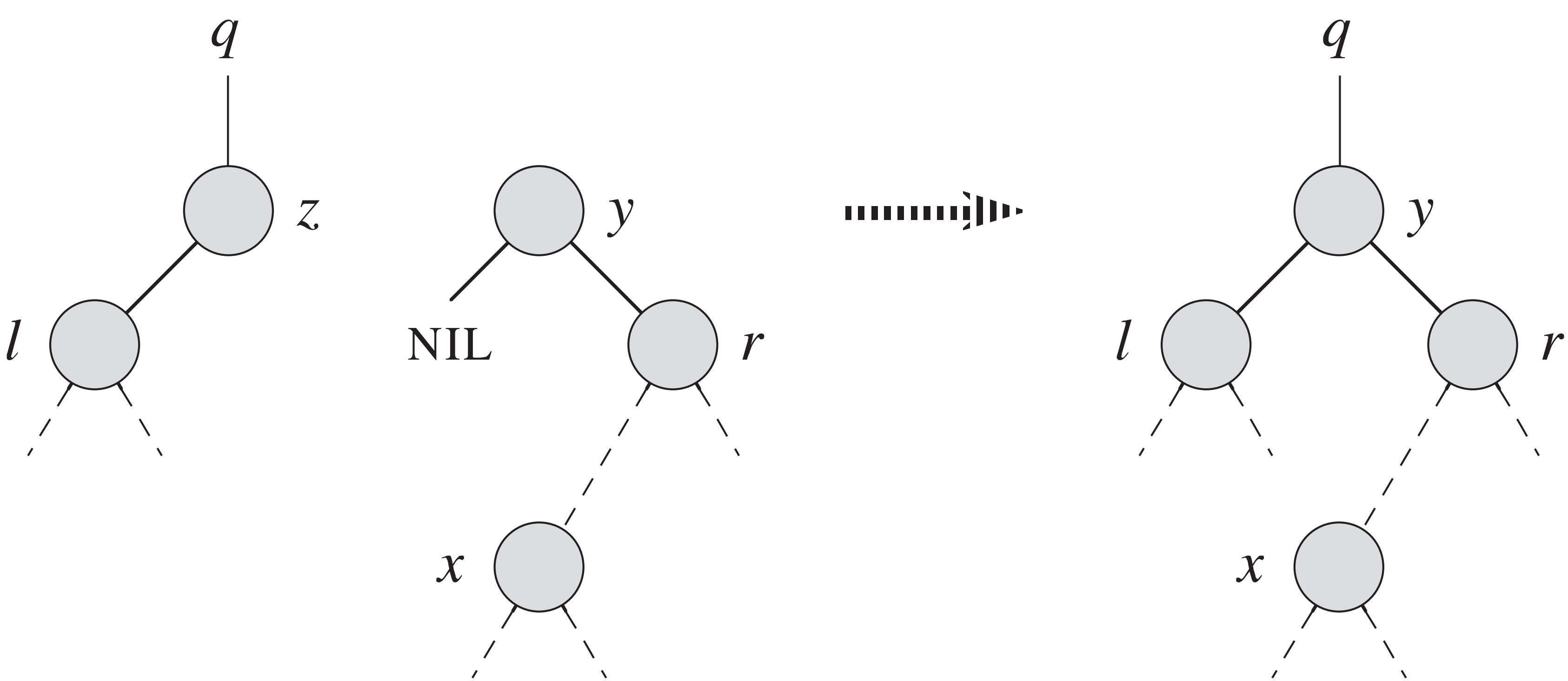
Caso 3a: y è figlio di z



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)
if u.padre==NULL
        root=v;                        // u è proprio la radice
else if (u==u.padre.left)              // u è figlio sinistro di suo padre
        u.padre.left=v;
else
        u.padre.right=v;
if (v!=NULL)                           // aggioniamo il padre di v
         v.padre=u.padre;
```

Cancellazione

```
Delete(root, z)
if (z.left==NULL)
        Trapianta(T, z , z.right);    // z non ha figlio sinistro -> sposto il suo
                                      figlio destro ‘in alto’ al posto di z
else if (z.right==NULL)               // analogamente se z non ha figlio destro
        Trapianta(T,z,z.left);
else
        y=Minimo(z.right);   //uguale a successor poiché z.right!=NULL
        if (y.padre!=z)      // se y non è figlio (destro) di z …
                Trapianta(T, y , y.right); // sposto y.right in y …
                y.right=z.right;   // … e z.right come figlio dx di y
                y.right.padre=y;
        Trapianta(T, z , y);   // ora sposto y in z (dopo aver settato y.right)
        y.left=z.left;         // sistemo il sottoalbero sx di z in y.left
        y.left.padre=y;
```