Università di Roma
Tor Vergata

Strutture dati avanzate

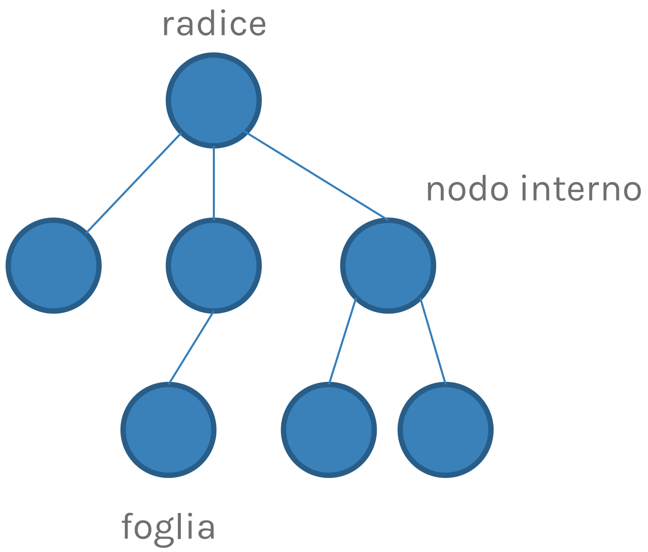
Alessandro Pellegrini
pellegrini@diag.uniroma1.it

Alberi

Definizioni di base

- Si tratta di una struttura collegata in cui i nodi sono organizzati in una gerarchia di differenti livelli

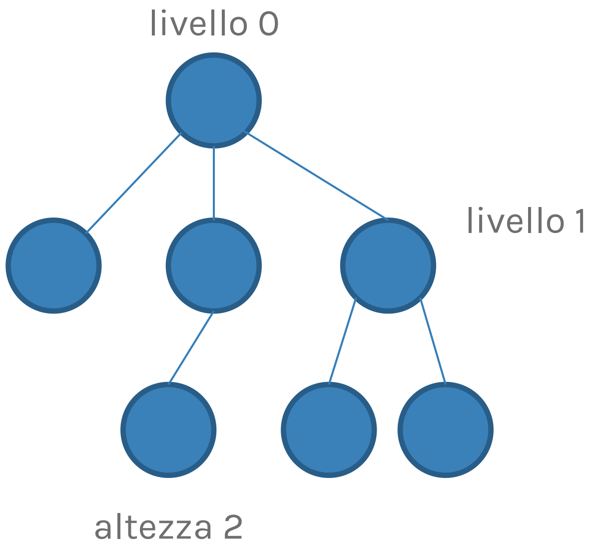
- Concetti intuitivi:
  - genitore
  - figlio
  - fratello



- Un nodo è:
  - interno, se ha almeno un figlio
  - foglia, se non ha figli
  - radice, se non ha genitori

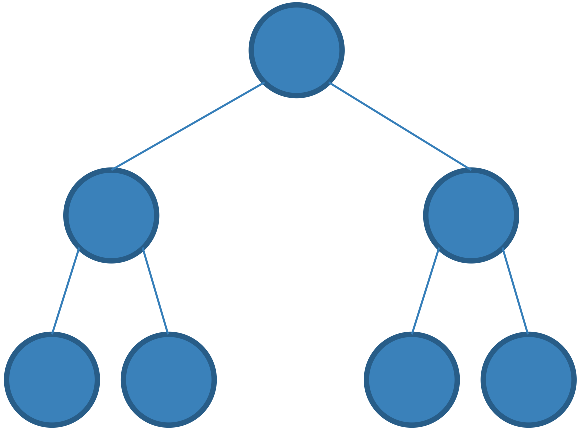
Definizioni di base

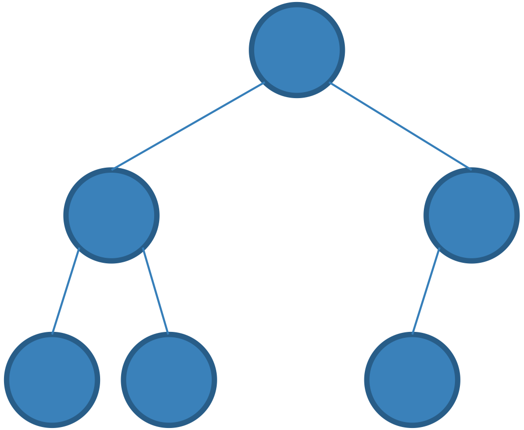
- Il *livello* di un nodo è la lunghezza (numero di nodi attraversati) per raggiungere quel nodo a partire dalla radice
- *ramo*: percorso dalla radice a una foglia
- *altezza*: lunghezza del ramo più lungo
- *grado* di un nodo: numero di figli
- *foresta*: un insieme di alberi



Alberi binari pieni e completi

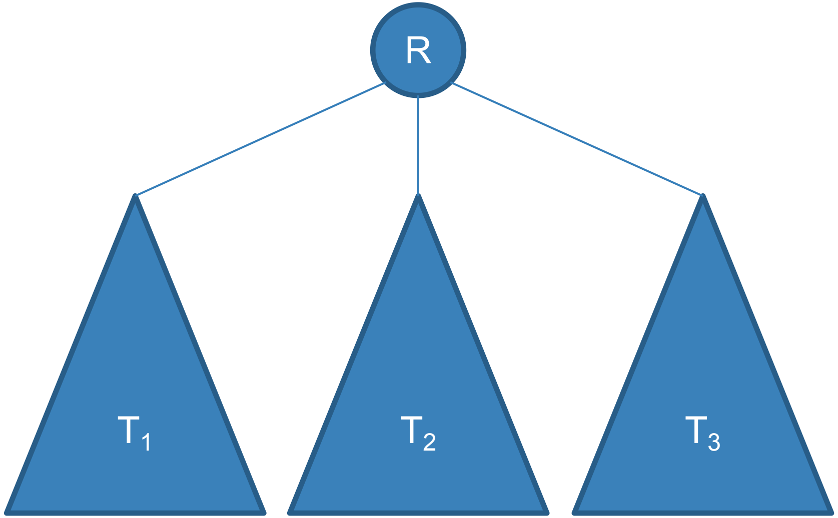
- Un albero binario è *pieno* se ogni suo nodo o è una foglia, o ha grado esattamente due (chiamati anche alberi binari propri, o 2-alberi)
- Un albero binario è *completo* se è riempito su tutti i livelli, con la possibile eccezione dell'ultimo, che è riempito fino ad un certo punto





Tipo di dato astratto

- Un insieme vuoto di nodi, oppure un insieme costituito da una *radice* R e da zero o più *sottoalberi*
- La radice di un sottoalbero è collegata al nodo padre da un arco orientato



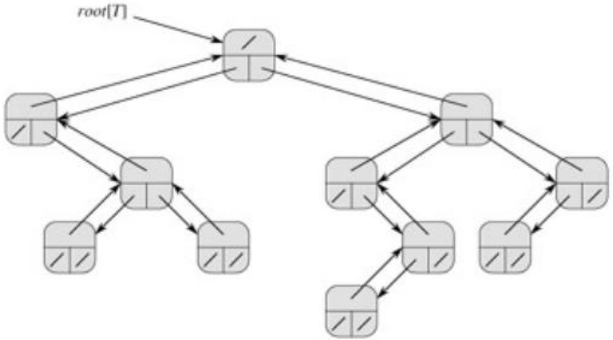
Tipo di dato astratto

- Operazioni tipiche che si desidera effettuare su un albero:
  - Inserimento di un nodo
  - Ricerca di un nodo
  - `parent(v)`: restituisce il genitore del nodo `v`
  - `children(v)`: restituisce l'insieme dei figli del nodo `v`
  - `isLeaf(v)`: restituisce True se il nodo `v` è una foglia
  - `isRoot(v)`: restituisce True se il nodo `v` è una foglia
  - `root()`: restituisce un riferimento al nodo radice
  - `level(v)`: restituisce il livello del nodo `v`
  - `height()`: restituisce l'altezza dell'albero
  - `len()`: restituisce il numero di nodi
  - `leaves()`: restituisce il numero di foglie
  - `arity()`: massimo numero di figli di un nodo dell'albero

- La particolare implementazione di queste operazioni dipende dalla specifica incarnazione dell'albero

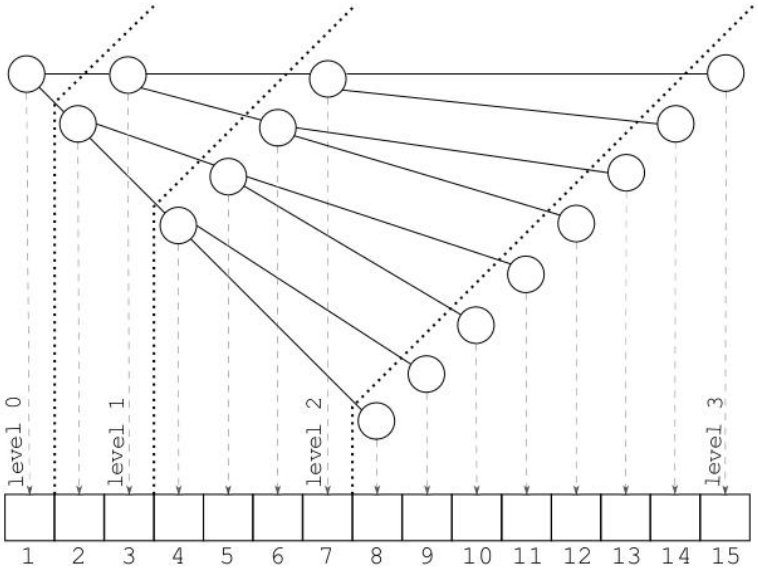
Alberi binari

- Gli alberi binari hanno molti utilizzi
- Sono la forma più semplice di albero, in cui in ciascun nodo viene mantenuto il riferimento ad (al più) due sottoalberi
- Per convenienza, ciascun nodo mantiene anche un riferimento al nodo padre



Utilizzo di array

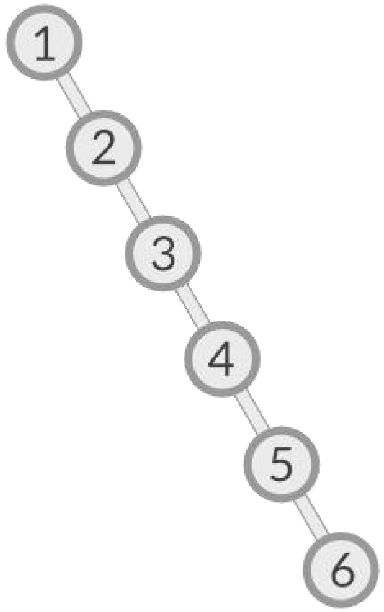
- È possibile rappresentare un albero in maniera compatta mediante un array
  - Ogni nodo è memorizzato in posizione p(v)
- Se v è la radice, p(v)=1
- Se v è figlio sinitro di u, allora p(v)=2p(u)
- Se v è figlio destro di u, allora p(v)=2p(u)+1



Utilizzo di array

- Problema nel caso di alberi incompleti:
  - ci può essere uno spreco di risorse
  - il caso degenere può richiedere un vettore di $2^n - 1$ elementi

- Si può utilizzare un array dinamico come classe base per implementare un albero in cui sia necessario effettuare molti inserimenti/eliminazioni

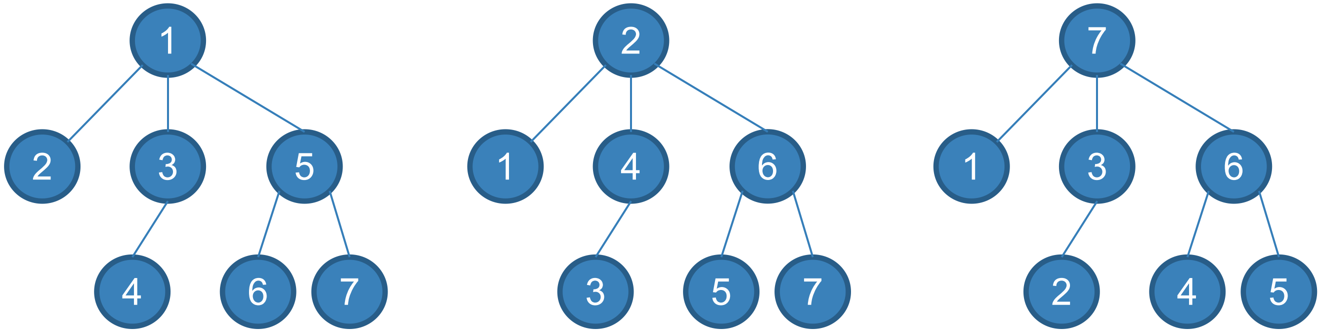


Visite di alberi

- Gli algoritmi di visita (o attraversamento) degli alberi “esplorano” tutti i nodi di un albero
  - Sono operazioni utili, ad esempio, per la ricerca di un elemento specifico contenuto all'interno dei nodi dell'abero
- Due grandi famiglie di operazioni, con differenti varianti
  - In profondità (depth-first search—DFS, a scandaglio)
    - in preordine (preorder, o ordine anticipato)
    - in ordine simmetrico (in order)
    - in postordine (postorder, o ordine posticipato)
  - In ampiezza (breadth-first search—BFS, a ventaglio)

Depth-First Search

- Vengono visitati i rami, uno dopo l'altro
- In *preordine*: dato un nodo v, visita v e poi visita i sottoalberi di v, da sinistra verso destra
- In *ordine simmetrico*: dato un nodo v, visita il primo sottoalbero, poi visita v, poi visita i restanti sottoalberi
- In *postordine*: visita i sottoalberi di v, da sinistra verso destra, poi visita v



DFS per alberi binari

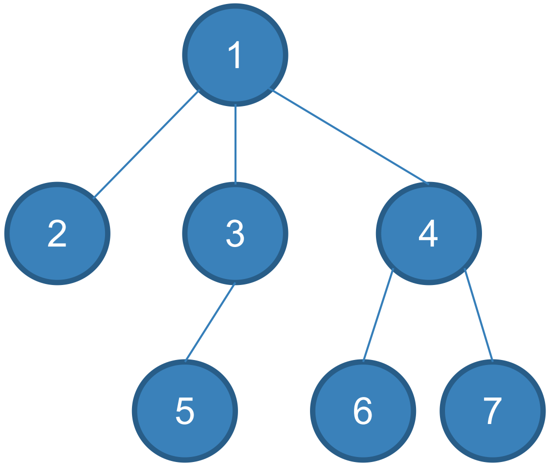
```
PREORDERDFS(v):
    if v ≠⊥ then:
        <do something with v>
        PREORDERDFS(v.left)
        PREORDERDFS(v.right)
```

```
POSTORDERDFS(v):
    if v ≠⊥ then:
        POSTORDERDFS(v.left)
        POSTORDERDFS(v.right)
        <do something with v>
```

```
INORDERDFS(v):
    if v ≠⊥ then:
        INORDERDFS(v.left)
        <do something with v>
        INORDERDFS(v.right)
```

Breadth-First Search

- Tipicamente implementata utilizzando delle strutture dati di appoggio (ad esempio, una coda)
- I nodi vengono visitati per livelli, partendo dalla radice

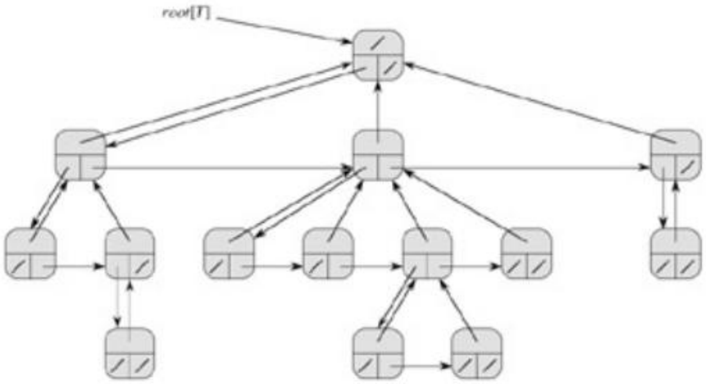


BFS

```
BFS(root):
    q ← Queue()
    node ← root
    while node != ⊥:
        <do something with node>
        for each child of node:
            q.enqueue(child)
        node ← q.dequeue()
```

Alberi n-ari

- Se si desidera avere più figli, è possibile incrementare il numero di puntatori
- Problema: cosa succede se il numero di figli non è noto a priori?
- È possibile utilizzare una lista (doppiamente) collegata per rappresentare i *siblings* (fratelli) di uno stesso livello
- Il nodo padre utilizzerà i puntatori head e tail per riferire i nodi iniziale e finale di tale lista

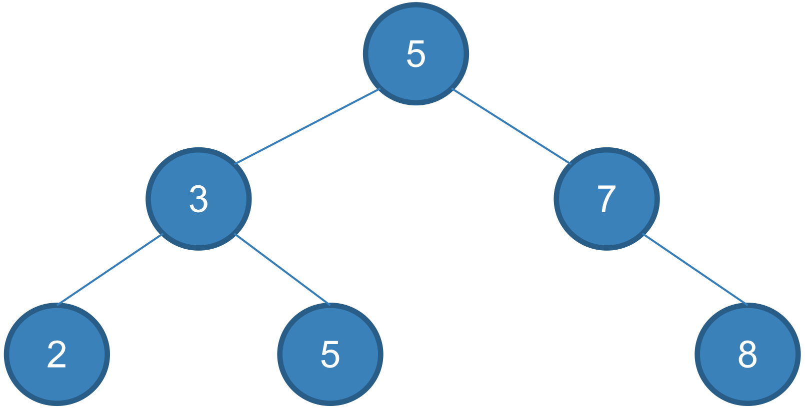


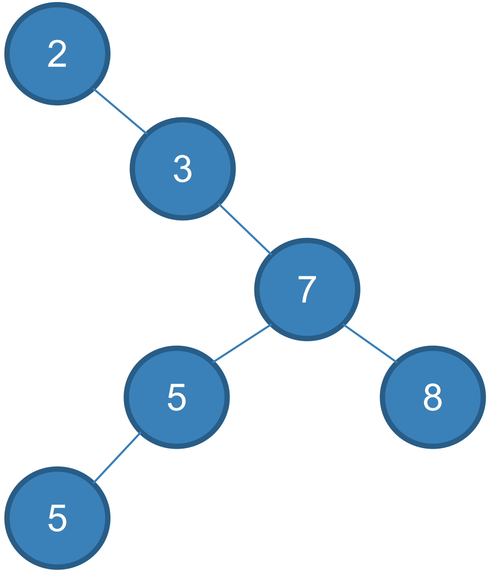
Alberi binari di ricerca

- Gli alberi binari di ricerca sono strutture dati che supportano alcune operazioni di tipo insiemistico:
  - `search()`: ricerca un nodo associato ad una specifica chiave nell'albero
  - `minimum()`: restituisce il nodo con il valore più piccolo della chiave
  - `maximum()`: restituisce il nodo con il valore più grande della chiave
  - `predecessor()`: dato un nodo, restituisce quello con la chiave immediatamente precedente
  - `successor()`: dato un nodo, restituisce quello con la chiave immediatamente successiva
  - `insert()`: inserisce un nuovo nodo con una determinata chiave
  - `delete()`: rimuove un nodo dall'albero
- la visita in profondità in ordine simmetrico consente di accedere ai nodi dell'albero *in ordine di chiave*

Alberi binari di ricerca

- Proprietà fondamentale:
  - sia v un nodo dell'albero binario di ricerca. Se w è un nodo nel sottoalbero sinistro di v, allora w.key ≤ x.key. Viceversa, se w appartiene al sottoalbero destro, allora w.key ≥ x.key.





Ricerca in un albero binario

- Per cercare un nodo con una data chiave in un albero binario di ricerca, si può effettuare una visita sfruttando la proprietà fondamentale per decidere se proseguire la visita nel sottoalbero destro o sinistro

```
TREESEARCH(v, key):
    if v == ⊥ or key == key[v] then:
        return v
    if key < key[v] then
        return TREESEARCH(v.left, key)
    return TREESEARCH(v.right, key)
```

- Il costo è $O(h)$, con h altezza dell'albero

Massimo e minimo

TREEMINIMUM(v):

while v.left ≠⊥:

v ← v.left

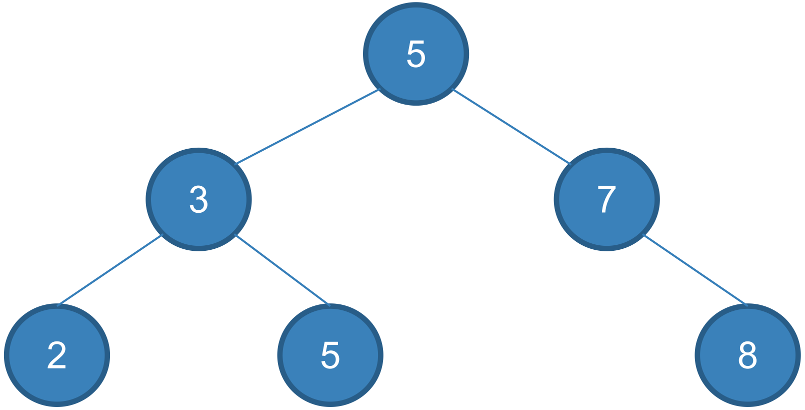
return v

TREEMAXIMUM(v):

while v.right ≠⊥:

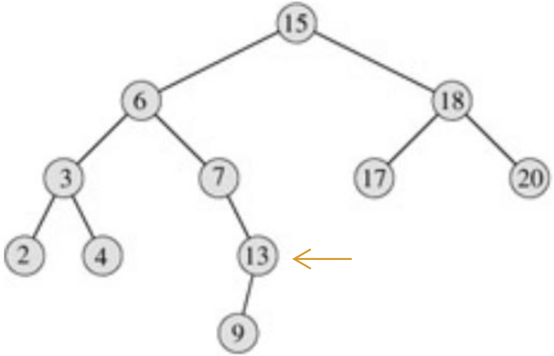
v ← v.right

return v



Successore

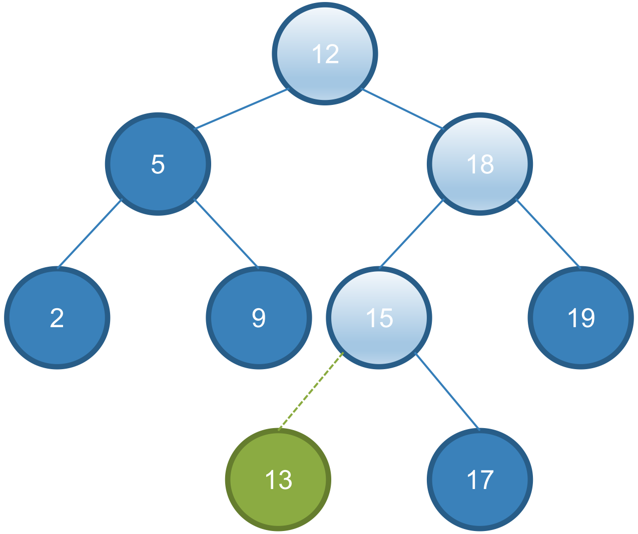
```
TREESUCCESSOR(v):
    if v.right ≠⊥ then:
        return treeMinimum(v.right)
    y ← v.parent
    while y ≠⊥ and x = y.right:
        x ← y
        y ← y.parent
    return y
```



- Il primo caso dell'algoritmo copre la presenza di un sottoalbero destro, di cui ci interessa un minimo
- Il secondo caso richiede invece di trovare il nodo che sia il più vicino antenato di v il cui figlio sinistro è anche un antenato di v

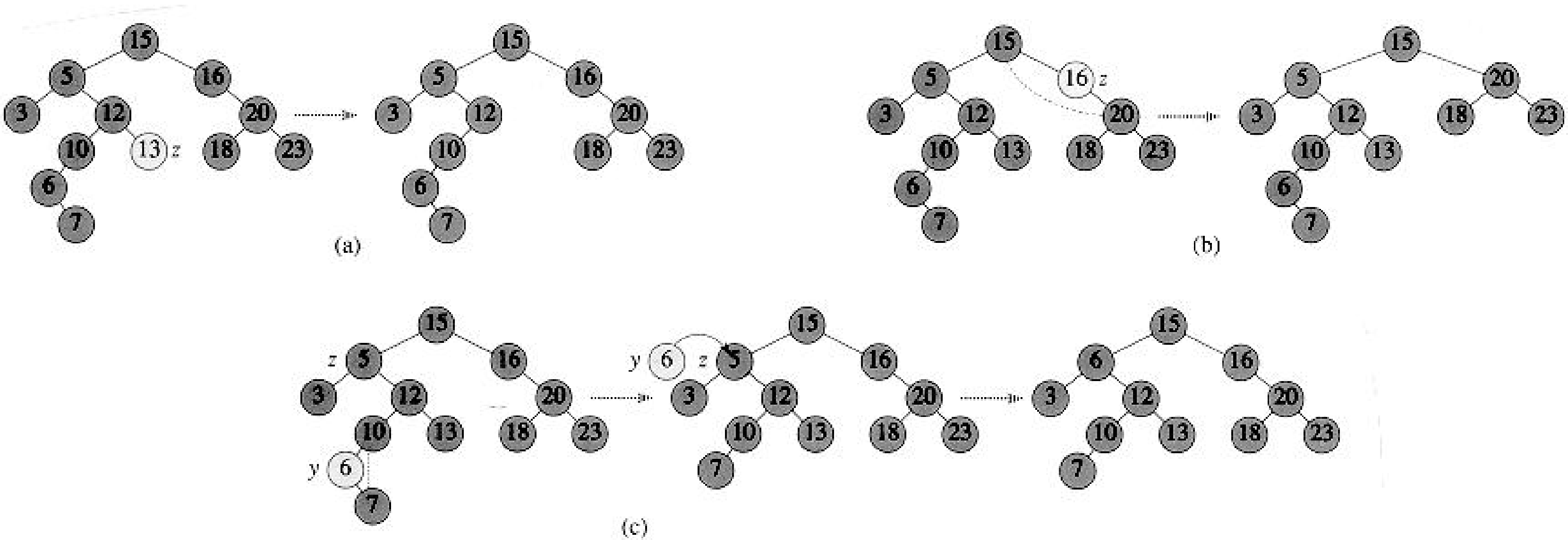
Inserimento

```
TREEINSERT(T, z):
    y ← ⊥                 ▸ genitore di x
    x ← T.root            ▸ Puntatore del percorso
    while x ≠⊥:
        y ← x
        if z.key < x.key then:
            x ← x.left
        else:
            x ← x.right
    z.parent ← y
    if y = ⊥ then:        ▸ Albero vuoto
        T.root ← z
    else if z.key < y.key then:
        y.left ← z
    else:
        y.right ← z
```



Eliminazione

- Tre casi differenti da coprire:
  a) il nodo da eliminare non ha figli
  b) il nodo da eliminare ha un solo figlio
  c) il nodo da eliminare ha due figli



(a)

(b)

(c)

Eliminazione

```
TREEDELETE(T, z):
    if z.left = ⊥ or z.right = ⊥ then:
        y ← z
    else:
        y ← TREESUCCESSOR(z)
    if y.left ≠⊥ then:
        x ← y.left
    else:
        x ← y.right
    if x ≠⊥ then:
        x.parent ← y.parent
    [...]
```

```
    [...]
    if y.parent = ⊥ then:
        T.root ← x
    else if y = y.parent.left then:
        y.parent.left ← x
    else:
        y.parent.right ← x
    if y ≠ z then:
        z.key ← y.key    ▸ Copia dati aggiuntivi
return y
```

Eliminazione

Sgancia il nodo y

TREEDELETE(T, z): Trova il nodo y da sganciare

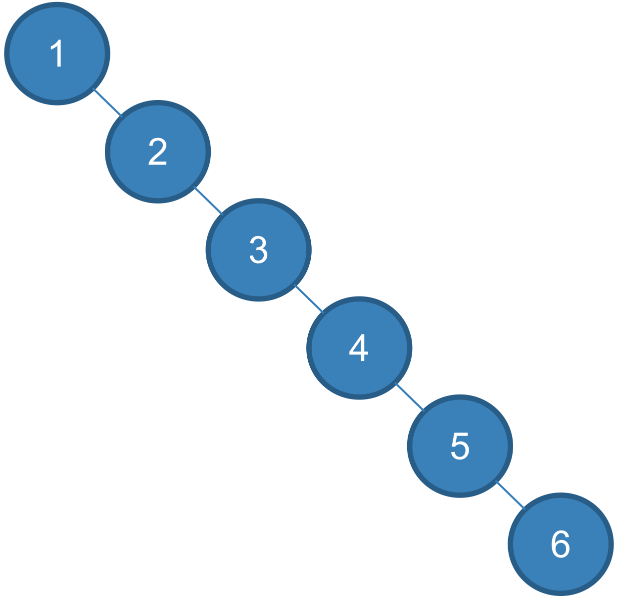
```
if z.left = ⊥ or z.right = ⊥ then:
    y ← z
else:
    y ← TREESUCCESSOR(z)
if y.left ≠⊥ then:
    x ← y.left
else:
    x ← y.right        Trova un figlio di y
if x ≠⊥ then:
    x.parent ← y.parent
[...]
```

```
[...]
if y.parent = ⊥ then:
    T.root ← x
else if y = y.parent.left then:
    y.parent.left ← x
else:
    y.parent.right ← x
if y ≠ z then:
    z.key ← y.key    ▸ Copia dati aggiuntivi
return y
```

Se è stato sganciato un figlio, copia i dati all'interno del nodo che doveva essere in realtà eliminato

Alberi binari di ricerca degeneri

- Quello in figura è un albero binario di ricerca?
- Come è possibile che si sia arrivati ad un albero del genere?
- Qual è il costo delle operazioni viste in precedenza?

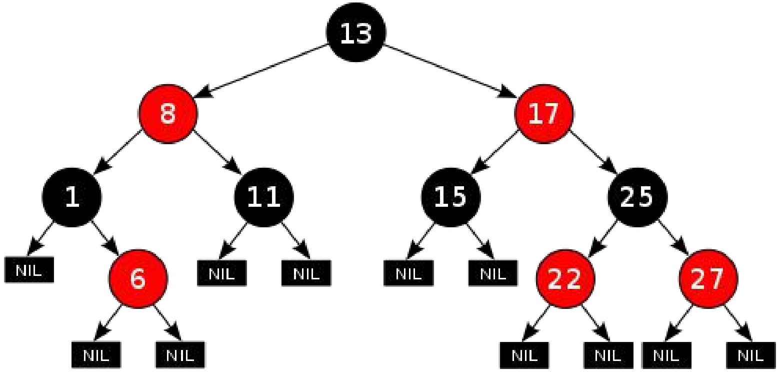


Alberi binari di ricerca auto-bilanciati

- Gli alberi auto-bilanciati (self-balancing trees) sono degli alberi binari di ricerca che tentano di non far verificare mai il caso degenere, anche in caso di inserimenti in ordine svantaggioso
- Verificano automaticamente se un'operazione di inserimento o di eliminazione ha provocato un forte “sbilanciamento” dell'albero
- In caso affermativo, modificano le relazioni tra i nodi (ribilanciamento) per riportarsi il più possibile ad un caso bilanciato
- Molte varianti:
  - AA Tree, AVL Tree, B-tree, Red-Black Tree, Splay tree...

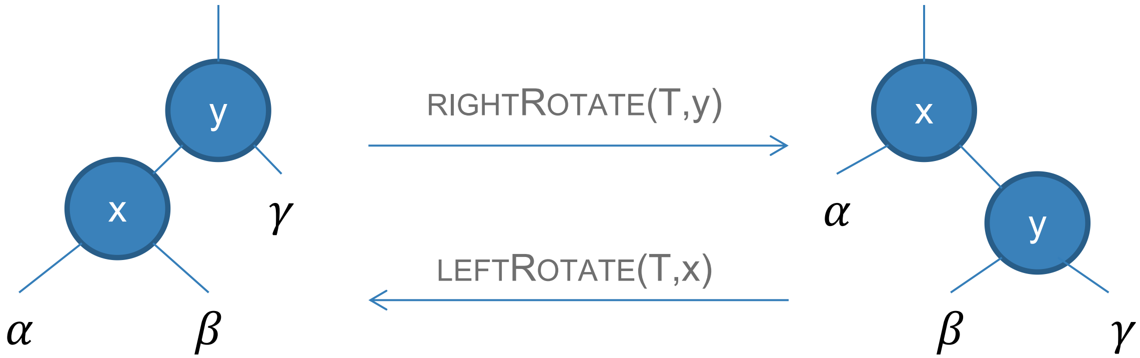
Red-Black Tree

- Gli alberi rosso-neri sono alberi binari di ricerca che assicurano che, in nessun caso, un percorso radice-foglia sia più lungo del doppio di ogni altro percorso radice-foglia
- Si basa sul colore dei nodi, che possono essere rossi o neri:
  - la radice è sempre nera
  - Ogni foglia (NIL) è sempre nera
  - Se un nodo è rosso, allora entrambi i figli sono neri
  - Dato qualsisi nodo, tutti i percorsi verso le foglie contengono lo stesso numero di nodi neri



Rotazioni

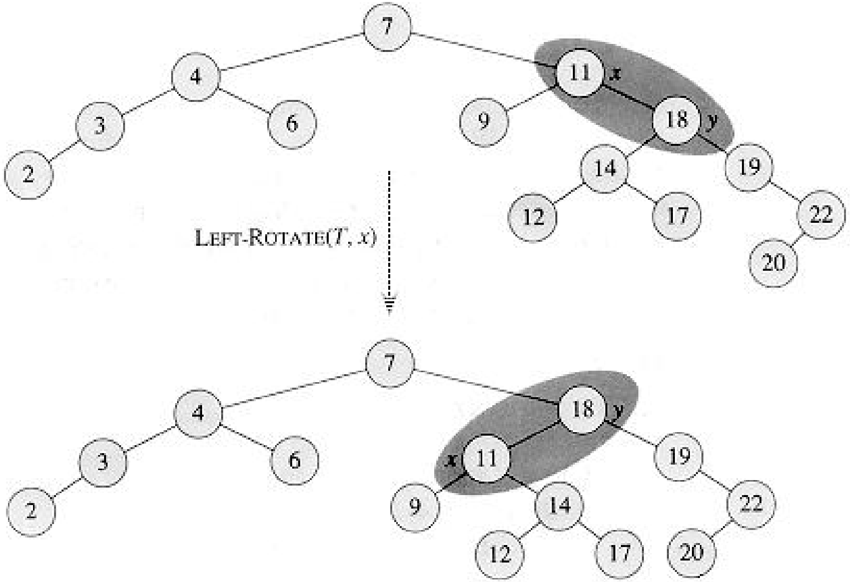
- I red-black tree reagiscono al verificarsi di situazioni degeneri applicando le operazioni di “rotazione”
  - È un'operazione locale che conserva la proprietà fondamentale degli alberi binari di ricerca
- Le rotazioni coinvolgono una coppia di nodi e tutti i loro sottoalberi destro e sinistro



Rotazione a sinistra

```
LEFTROTATE(T, x):
    y ← x.right
    x.right ← y.left                ▸ Scambia i sottoalberi
    y.left.parent ← x
    y.parent ← x.parent             ▸ Collega il genitore di x ad y
    if x.parent = ⊥ then:
        T.root ← y
    else if x = x.parent.left then:
        x.parent.left ← y
    else:
        x.parent.right ← y
    y.left ← x                      ▸ Metti x a sinistra di y
    x.parent ← y
```

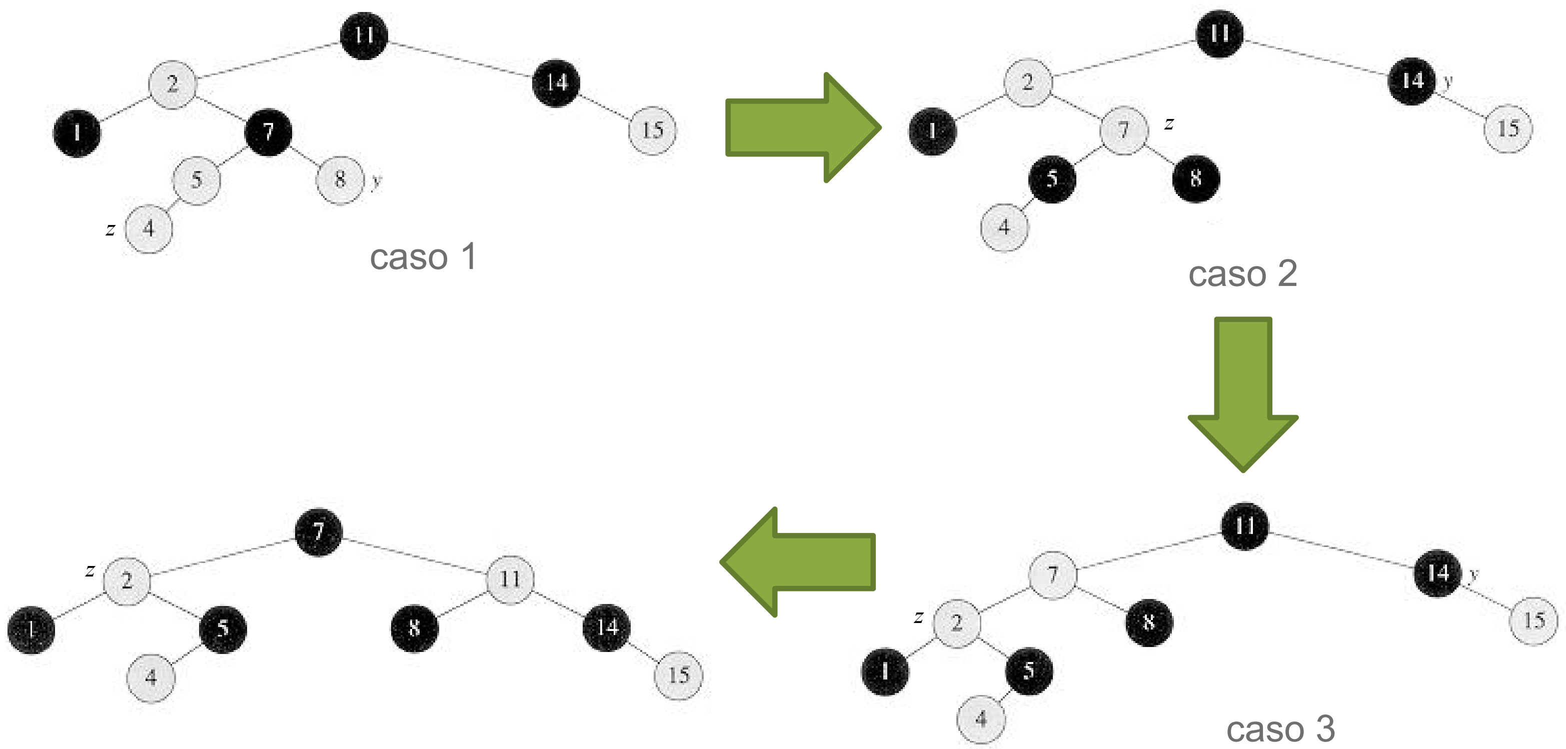
Rotazione a sinistra



Inserimento

```
RB-INSERT(T, z):
    y ← ⊥
    x ← T.root
    while x ≠ ⊥:
        y ← x
        if z.key < x.key then:
            x ← x.left
        else:
            x ← x.right
    z.parent ← y
    if y = ⊥ then:
        T.root ← z
    else if z.key < y.key then:
        y.left ← z
    else:
        y.right ← z
    z.left ← ⊥
    z.right ← ⊥
    z.color ← RED
    RB-INSERTFIXUP(T, z)
```

Violazioni dell'inserimento



RB-Tree Fixup

- La procedura di fixup cerca di spostarsi tra i tre casi verso la condizione di bilanciamento, cercando di garantire sempre che *dato qualsisi nodo, tutti i percorsi verso le foglie contengono lo stesso numero di nodi neri*
- Ci sono in realtà sei casi da coprire, ma tre sono simmetrici
  - riguardano le differenti violazioni che coinvolgono i sottoalberi destro o sinistro

RB-Tree Fixup

```
RB-INSERTFIXUP(T, z):
  while z.parent.color = RED:
    if z.parent = z.parent.parent.left then:
      y ← z.parent.parent.right
      if y.color = RED then:
        z.parent.color ← BLACK
        y.color ← BLACK
        z.parent.parent.color ← RED
        z ← z.parent.parent
      else if z = z.parent.right then:
        z ← z.parent
        LEFTROTATE(T, z)
      z.parent.color ← BLACK
      z.parent.parent.color ← RED
      RIGHTROTATE(T, z.parent.parent)
    else:
      [come l'altro ramo, ma con
      "right" e "left" scambiati]
  T.root.color ← BLACK
```

Ci fa muovere tra i vari casi

Caso in cui operiamo su un sottoalbero sinistro

Caso 1

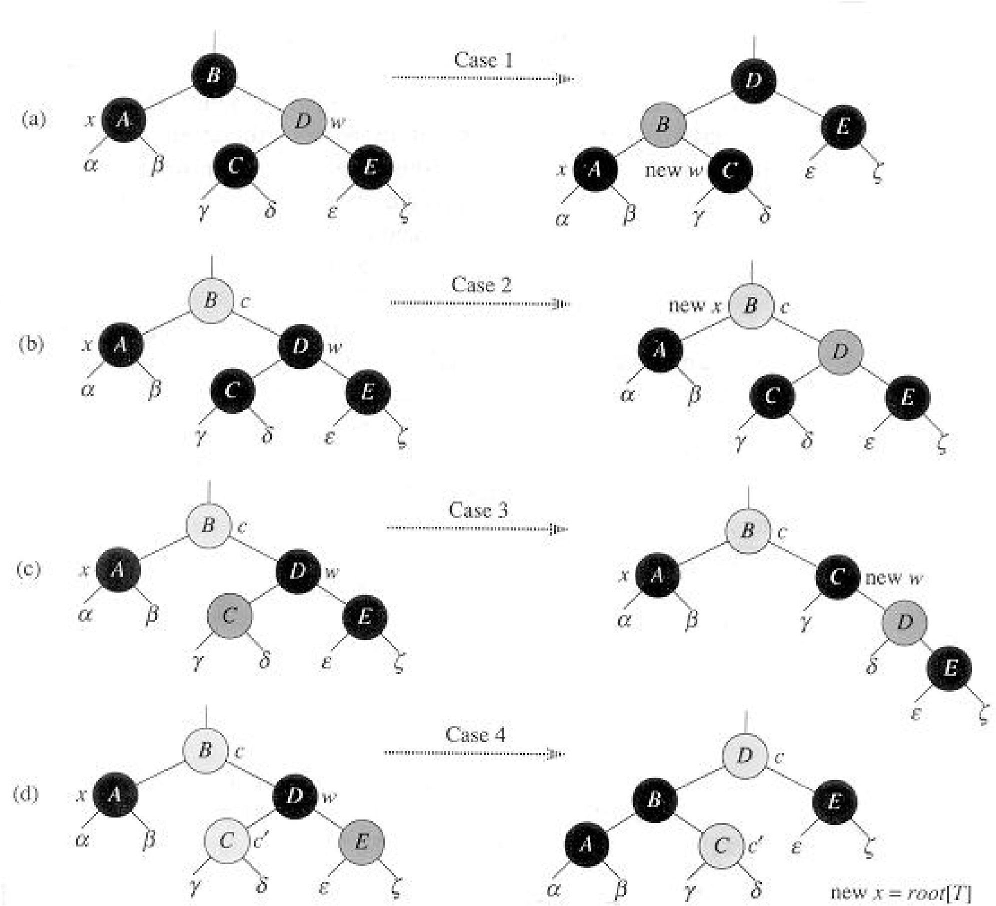
Caso 2

Caso 3

Eliminazione

```
RB-DELETE(T, z):
  if z.left = ⊥ or z.right = ⊥ then:
    y ← z
  else:
    y ← TREESUCCESSOR(z)
  if y.left ≠ ⊥ then:
    x ← y.left
  else:
    x ← y.right
  x.parent ← y.parent
  if y.parent = ⊥ then:
    T.root ← x
  else if y = y.parent.left then:
    y.parent.left ← x
  else:
    y.parent.right ← x
  if y ≠ z then:
    z.key ← y.key
  if y.color = BLACK
    then RB-DELETEFIXUP(T,x)
  return y
```

Violazioni nell'eliminazione



Nodi neri

Nodi rossi

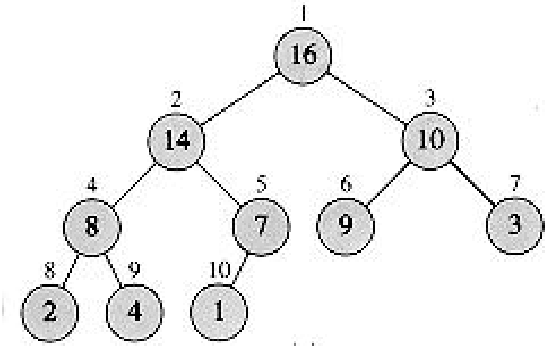
Nodi neri o rossi

Mucchi

Heap

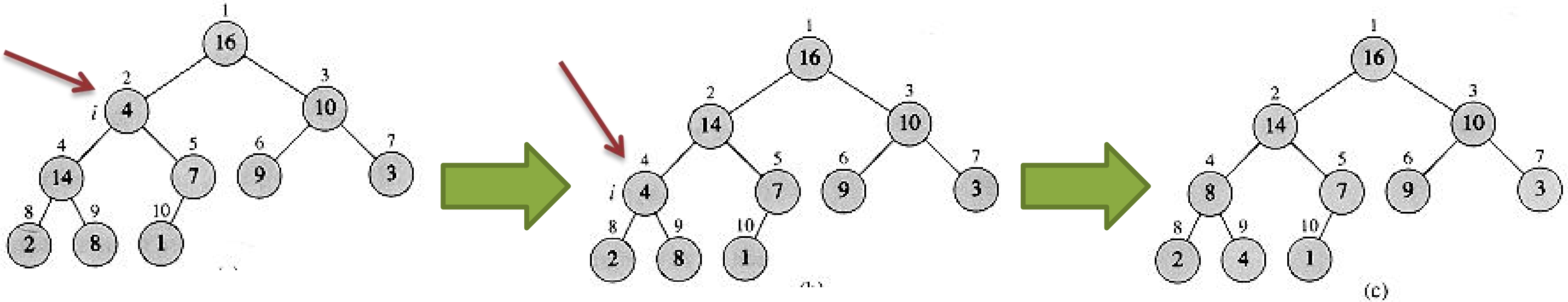
- L'heap (binario) è una struttura dati composta da un *albero binario completo* (non necessariamente *pieno*)

- Gli heap soddisfano la *heap property*:
  - per ciascun nodo v differente dalla radice, v.parent.key ≥ v.key
  - Questo vuol dire che la chiave di un nodo è al più il valore del suo genitore.



Garantire la proprietà di heap: `heapify()`

- Si tratta di un'importante procedura sugli heap:
  - Dato un nodo v di un heap, `heapify(v)` assume che i sottoalberi destri e sinistri siano degli heap
  - Tuttavia, il nodo v potrebbe avere una chiave più piccola, violando quindi la proprietà di heap
  - `heapify()` ristruttura l'albero in maniera tale da ripristinare tale proprietà



Garantire la proprietà di heap: `heapify()`

```
HEAPIFY(v):
   l ← v.left
   r ← v.right
   if l ≠ ⊥ and l.key > v.key then:
      largest ← l
   else:
      largest ← v
   if r ≠ ⊥ and r.key > largest.key then:
      largest ← r
   if largest ≠ v then:
      scambia i nodi v e largest nell'albero
      HEAPIFY(v)
```

Coda di priorità

- La variante che abbiamo visto fin'ora è chiamata anche MaxHeap
- Il MinHeap sfrutta una proprietà di heap duale:
  - per ciascun nodo v differente dalla radice, v.parent.key ≤ v.key

- In maniera intuitiva, è possibile utilizzare un Min/Max Heap per realizzare una coda di priorità

- Occorre però prevedere due ulteriori operazioni sugli heap:
  - estrazione dell'elemento a più alta priorità
  - inserimento di un nuovo elemento nell'heap

Inserimento di un elemento nell'heap

- È possibile trattare inizialmente un heap come un albero binario tradizionale
- Il nuovo nodo viene inserito nell'ultimo livello dell'albero
- Può poi essere fatto risalire fino alla corretta posizione

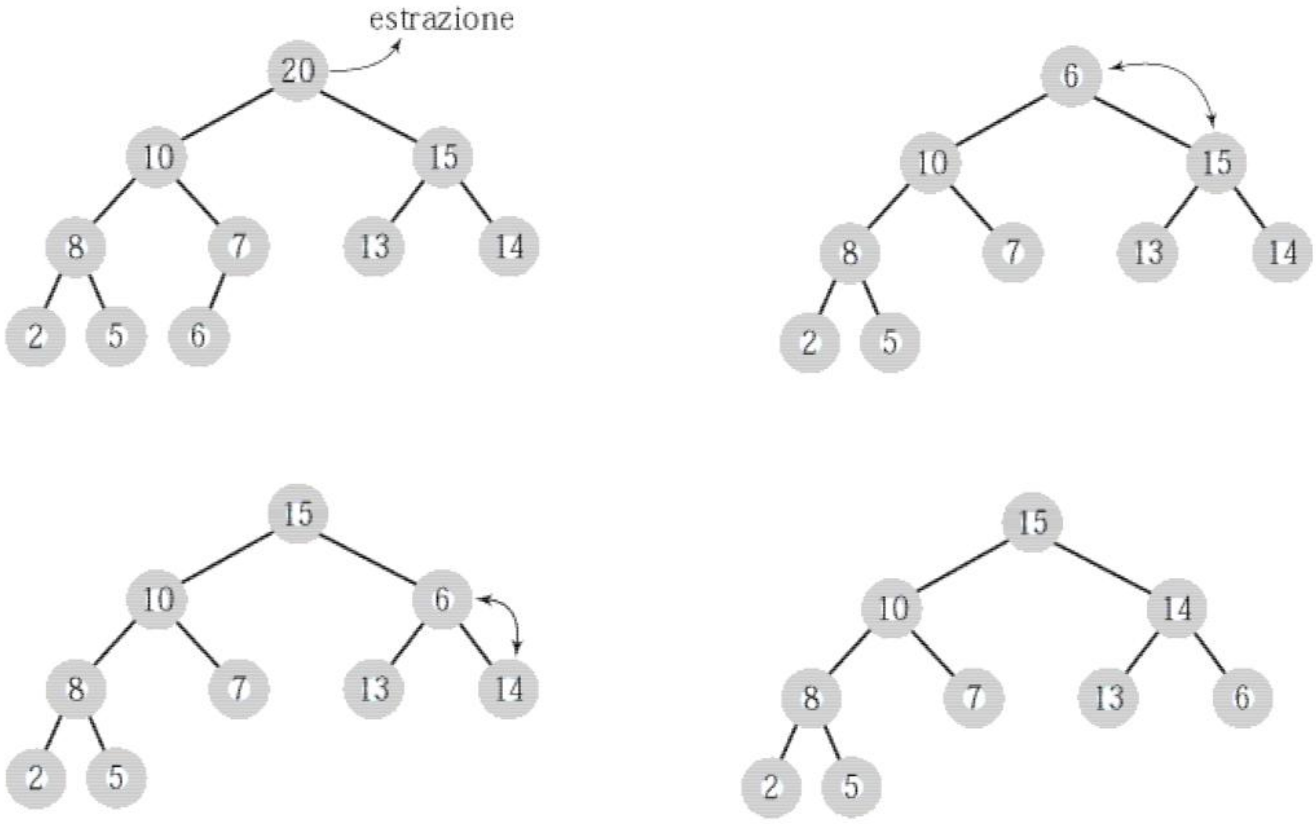
```
HEAPINSERT(H, key):
    node ← Node(key)
    LEAFINSERT(H, node)
    while node.parent ≠ ⊥ and node.parent.key < node.key:
        [scambia node e node.parent]
        node ← node.parent
```

Rimozione dell'elemento radice

- L'elemento radice è l'elemento a priorità massima, sia per i MinHeap che per i MaxHeap
- Rimuovere la radice di un heap corrisponde ad implementare l'operazione getMin() di una coda di priorità

```
DELETEFIRST(H):
    lastLeaf ← get the last leaf in the heap
    ret ← H.root
    rimpiazza H.root con lastLeaf
    HEAPIFY(H.root)
    return ret
```

Rimozione dell'elemento radice

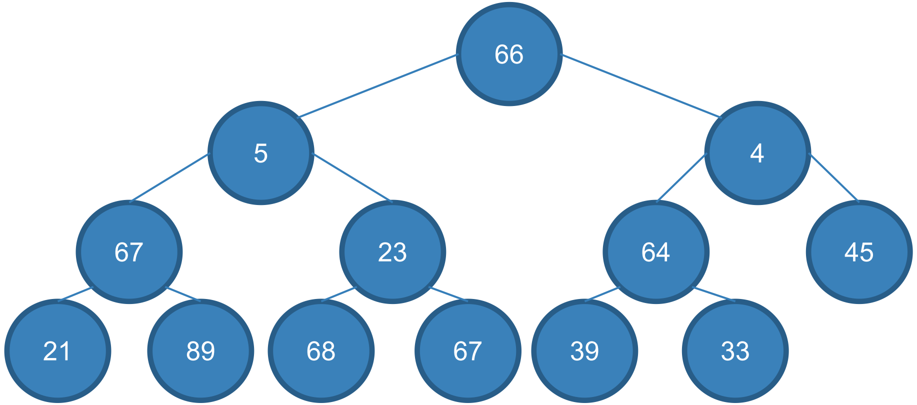


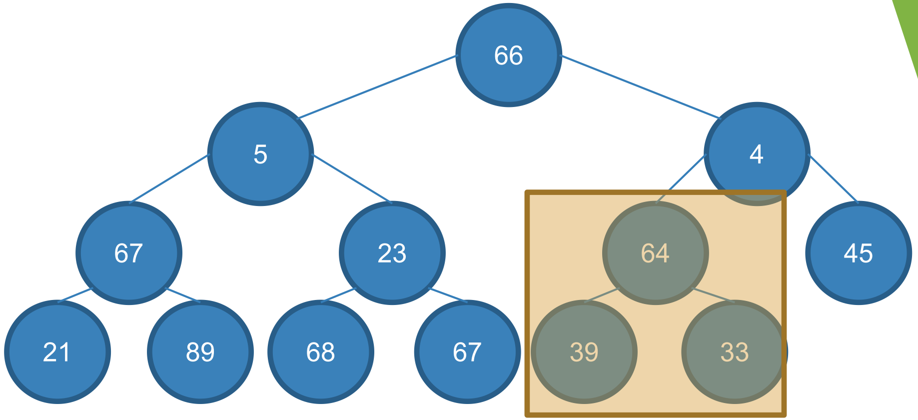
Algoritmo di Floyd

- Dato un vettore di numeri, l'algoritmo di Floyd costruisce un heap con gli elementi del vettore
- Inizialmente, il vettore viene convertito in un albero binario (sfruttando la rappresentazione tramite array dell'albero)
- In seguito, si parte dal penultimo livello
  - le foglie sono già heap
- Per ciascun nodo, si applica l'algoritmo HEAPIFY()
- Si risale di un livello, fino a giungere alla radice, applicando ripetutamente HEAPIFY()

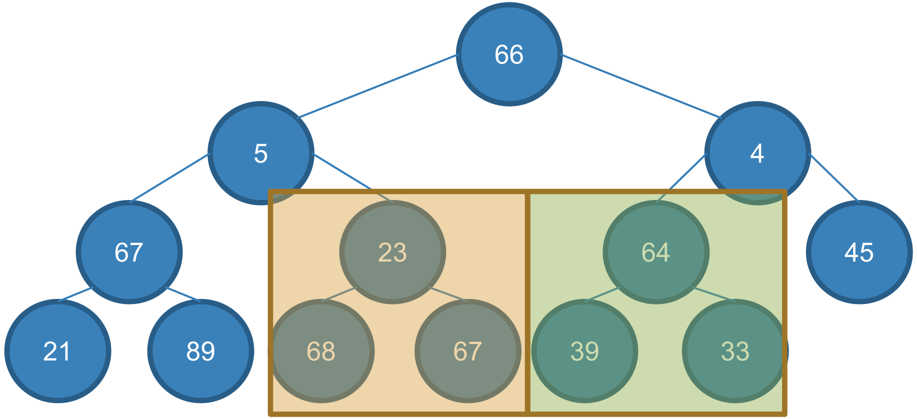
Algoritmo di Floyd

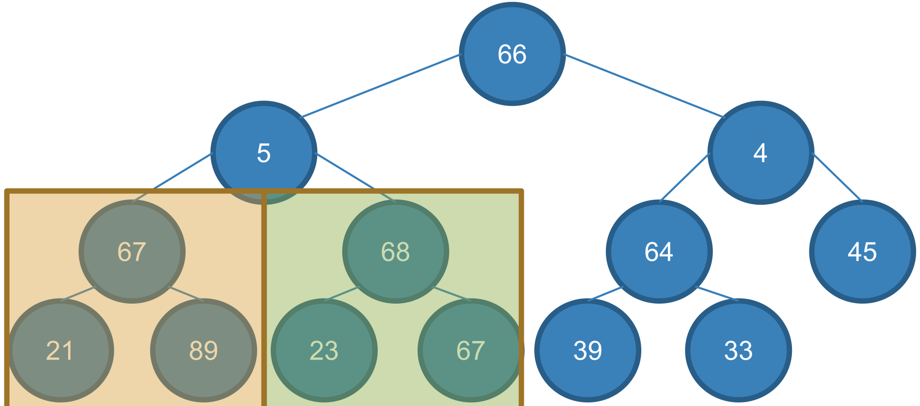
0	1	2	3	4	5	6	7	8	9	10	11	12
66	**5**	**4**	**67**	**23**	**64**	**45**	**21**	**89**	**68**	**67**	**39**	**33**



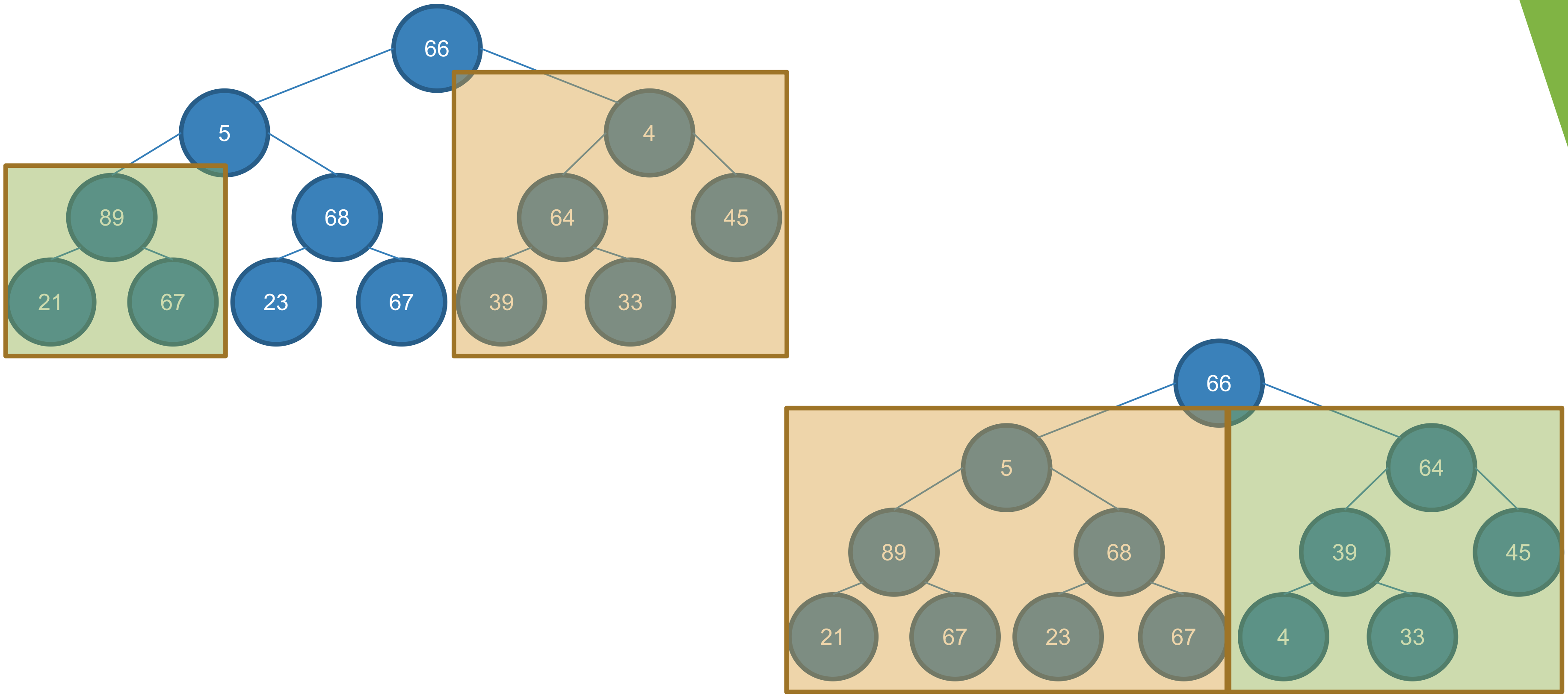


Algoritmo di Floyd

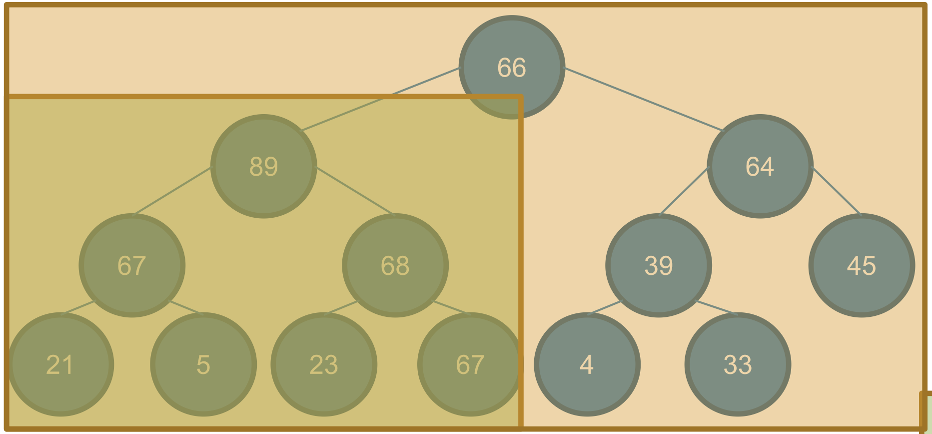


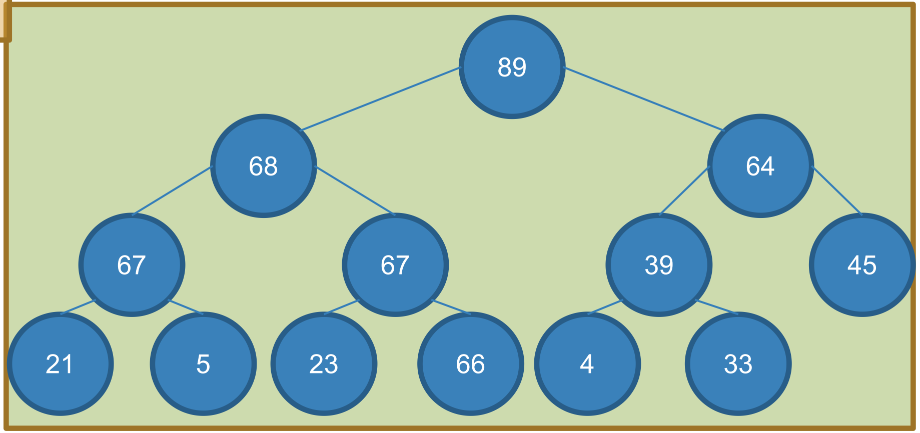


Algoritmo di Floyd



Algoritmo di Floyd





Algoritmo di Floyd

```
ARRAYTOHEAP(A):
    for i ← len(A) - 1 downto 0:
        HEAPIFYINPLACE(A, i)

HEAPIFYINPLACE(A, i):
    if isLeaf(A, i) then:
        return
    j ← GETMAXCHILDINDEX(A, i)
    if A[i] < A[j] then:
        EXCHANGE(A, i, j)
        HEAPIFYINPLACE(A, j)
```

Algoritmo di Floyd: analisi della complessità

- Nel caso peggiore, ogni chiamata ad HEAPIFYINPLACE() effettua il numero massimo di scambi
- Supponiamo di avere un albero completo (con $n = 2^k - 1$ nodi)
- Quanti nodi troviamo ad ogni livello?
  - Nell'ultimo livello: $\frac{n+1}{2}$ foglie
  - Nel penultimo livello: $\frac{n+1}{4}$ nodi interni
  - Nel terzultimo livello: $\frac{n+1}{8}$ nodi interni
  - …

Algoritmo di Floyd: analisi della complessità

- HEAPIFYINPLACE() invocato su un nodo di livello i effettua al più $k - i$ scambi (operazione dominante)
- Il numero di livelli è $\log n + 1$

- Numero massimo di scambi totale:

$$\sum_{i=2}^{\log(n+1)} \frac{n+1}{2^i}(i-1) = (n+1) \sum_{i=2}^{\log(n+1)} \frac{i-1}{2^i}$$

$$= (n+1)\left(\sum_{i=2}^{\log(n+1)} \frac{i}{2^i} - \sum_{i=2}^{\log(n+1)} \frac{1}{2^i} \right)$$

Algoritmo di Floyd: analisi della complessità

- Considerato che $\sum_{i=2}^{\infty} \frac{i}{2^i} = \frac{3}{2}$, si ottiene che $\sum_{i=2}^{\log(n+1)} \frac{i}{2^i} > 0$
- Pertanto:

$$(n+1) \sum_{i=2}^{\log(n+1)} \frac{i-1}{2^i} < (n+1)\left(\frac{3}{2} - \sum_{i=2}^{\log(n+1)} \frac{1}{2^i}\right) < \frac{3}{2}(n+1)$$

- Ne segue che la complessità computazionale dell'algoritmo di Floyd è $O(n)$

Heap Sort

- Si tratta di un algoritmo di ordinamento basato sulla costruzione di un MaxHeap

```
HeapSort(A):
    heap ← ARRAYTOHEAP(A)
    for i ← len(A) - 1 downto 0:
        max ← getMax(heap)
        deleteFirst(heap)
        A[i] ← max
```

- Costo: $O(n) + O(\log n!) = O(n \log n)$, sfruttando l'approssimazione di Stirling

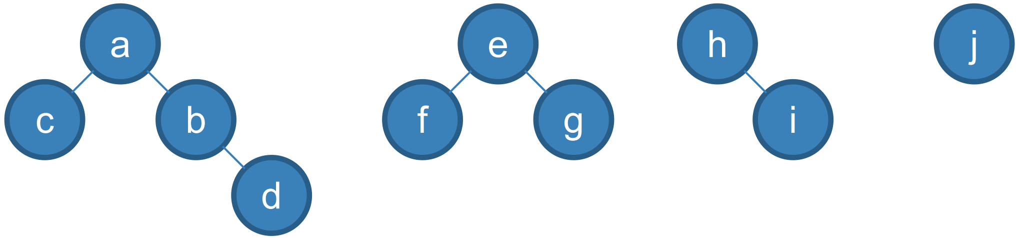
Insiemi Disgiunti

Abstract Data Type: Insiemi disgiunti

- Alcune applicazioni hanno la necessità di raggruppare elementi in una collezione di insiemi disgiunti, ciascuno identificato da un *elemento rappresentativo*
- Una struttura dati per insiemi disgiunti mantiene una collezione di n insiemi disgiunti *dinamici*
- Tipicamente, questi insiemi raggruppano oggetti contenitori come quelli che abbiamo visto per le altre strutture dati
- Operazioni fondamentali sono:
  - MAKESET(x): crea un nuovo insieme il cui elemento rappresentativo è x
  - UNION(x, y): unisce i due insiemi contenenti rispettivamente x ed y
  - FINDSET(x): restituisce un puntatore all'elemento rappresentativo dell'insieme che contiene x

Un esempio di applicazione

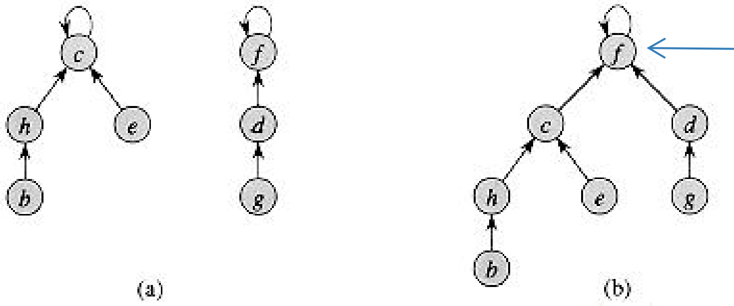
- Trovare le componenti connesse di una foresta



arco analizzato	insiemi
-	{a} {b} {c} {d} {e} {f} {g} {h} {i} {j}
(a,c)	{a,c} {b} {d} {e} {f} {g} {h} {i} {j}
(a,b)	{a,c,b} {d} {e} {f} {g} {h} {i} {j}
(b,d)	{a,c,b,d} {e} {f} {g} {h} {i} {j}
(e,f)	{a,c,b,d} {e,f} {g} {h} {i} {j}
(e,g)	{a,c,b,d} {e,f,g} {h} {i} {j}
(h,i)	{a,c,b,d} {e,f,g} {h,i} {j}

Rappresentazione di insiemi tramite alberi

- Rappresentiamo gli insiemi mediante alberi, il cui nodo radice è l'elemento rappresentativo
- I collegamenti sono “al contrario”: i figli puntano al padre
- MAKESET(x) restituisce un albero formato da un solo nodo
- FINDSET(x) naviga verso il nodo radice
- UNION(x, y) fa sì che la radice di un insieme punti alla radice dell'altro insieme



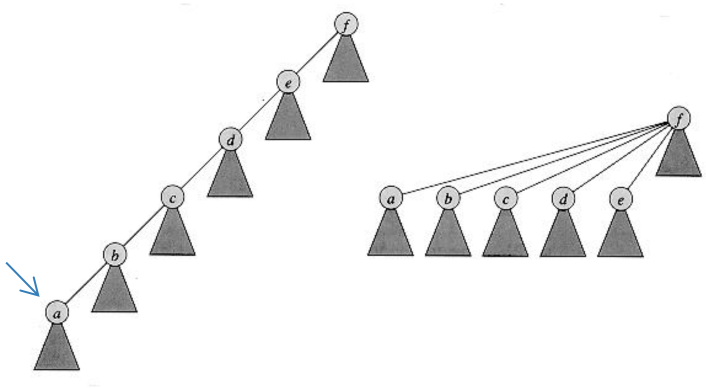
la radice ha come genitore sé stessa

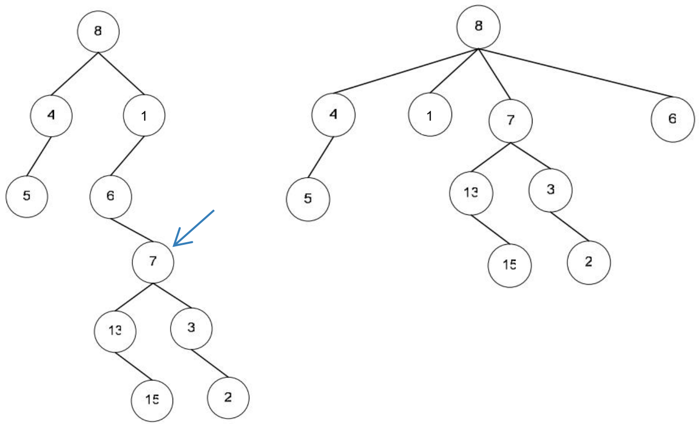
Problema di sbilanciamento

- La creazione di un insieme più grande aggiungendo progressivamente insiemi generati da varie chiamate a MAKESET(x) può creare un albero degenere
- Si può adottare un approccio *euristico*
  - Un'euristica, dal greco εὑρίσκω (scopro, trovo), è una modalità di soluzione di problemi che non segue un percorso chiaramente dimostrato, ma che si affida all'*intuito* e *allo stato temporaneo delle circostanze*

Problema di sbilanciamento

- Euristiche per risolvere il problema dello sbilanciamento:
  - *union by rank*: far sì che la radice dell'albero con meno livelli punti alla radice dell'albero con più nodi—si utilizza il concetto di rank: approssimazione del logaritmo della dimensione dei sottoalberi
  - *path compression*: si utilizza FINDSET() per far risalire i nodi verso la radice





Algoritmi

```
MAKESET(x):
    x.parent ← x
    x.rank ← 0

UNION(x, y):
    LINK(FINDSET(x), FINDSET(y))

FINDSET(x):
    if x ≠ x.parent then:
        x.parent ← FINDSET(x.parent)
    return x.parent
```

```
LINK(x,y):
if x.rank > y.rank then:
    y.parent ← x
else:
    x.parent ← y
    if x.rank = y.rank then:
        y.rank ← y.rank + 1
```

Analisi

- Consideriamo la sequenza MAKESET(), FINDSET(), LINK().
- Il costo quando si usa path compression può essere calcolato in maniera ammortizzata con il metodo degli accantonamenti:
  - MAKESET(): ha costo ammortizzato 2 (+1 di credito)
  - LINK(): ha costo ammortizzato 1
  - FINDSET(): ha costo ammortizzato 1
    - Questo costo paga la visita della radice e di un figlio
    - Tutte le operazioni di compressione successiva sono pagate dal credito di MAKESET().
  - Al termine dell'operazione, tutti i nodi saranno figli dello stesso nodo (la radice) e non dovranno più essere compressi
  - Il costo per m operazioni ha costo $O(m)$

Analisi

- Nel caso di utilizzo di union by rank, il costo di ciascuna operazione MAKESET() e LINK() costa $O(1)$
- Poiché il rank di un nodo è un limite superiore all'altezza del sottoalbero, ciascun percorso necessario a trovare l'elemento rappresentativo è $O(\log n)$
- Una sequenza di MAKESET(), LINK() e FINDSET() ha un costo pari a $O(\log n)$.
- Una sequenza di m MAKESET(), LINK() e FINDSET() ha un costo pari a $O(\mathrm{m}\log n)$.

- L'utilizzo congiunto di union by rank e path compression, nel caso peggiore si ottiene un costo di $O(m \cdot \alpha(n)) \approx O(m)$