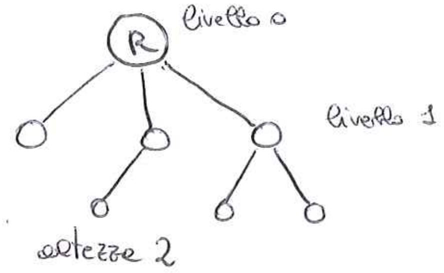
5. ALBERI

- **LIVELLO**: numero di nodi attraversati per raggiungere quel nodo a partire dalla radice,
- **ALTEZZA**: lunghezza del ramo più lungo (senza contare le foglie)
- **GRADO**: numero di figli.



ALBERO BINARIO PIENO: se ogni nodo è FOGLIA oppure ha 2 FIGLI (grado 2)

ALBERO BINARIO COMPLETO: se è riempito su TUTTI I LIVELLI, tranne l'ultimo

Operazioni Supportate

(1) Inserimento di un nodo;

(2) Ricerca di un nodo;

(3) parent(v): restituisce il NODO GENITORE di v;

(4) children(v): restituisce l'insieme dei figli di v;

(5) isLeaf(v): true se v è FOGLIA

(6) isRoot(v): true se v è RADICE

(7) root(): restituisce un riferimento al NODO RADICE

(8) level(v): livello del nodo v

(9) height(): ALTEZZA albero

(10) len(): NUMERO di nodi dell'albero

(11) leaves(): NUMERO di foglie

(12) arity(): MAX NUMERO di FIGLI DI 1 NODO.

ALBERI BINARI

Uso di Array:

Ogni nodo è memorizzato in posizione $p(v)$:

Se v è ROOT $\Rightarrow p(v) = 1$

Se v è figlio sinistro di $u \Rightarrow p(v) = 2 \cdot p(u)$
Se v è figlio destro di $u \Rightarrow p(v) = 2 \cdot p(u) + 1$

$\Big\}$ SX $\rightarrow$ PARI, DX $\rightarrow$ DISPARI

* PROBLEMI!

Nel caso di ALBERO INCOMPLETO: ci può essere uno spreco di risorse

$\longrightarrow$ vettore di $2^m - 1$ elementi

Visita DFS (Depth-First-Search):

In PREORDINE: visita v, poi sottoalbero sinistro di v, poi sottoalbero destro

In ORDINE SIMMETRICO: visita sottoalbero sinistro, poi v, poi il destro

In POSTORDINE: visita sottoalbero sinistro, poi sottoalbero destro, poi v.

Visita BFS (Breadth-First Search):

Implementata utilizzando strutture dati di appoggio (code, stack, ...)

I nodi vengono visitati PER LIVELLI.

```
BFS (root):
   q ← Queue()
   node ← root
   while node != ⊥ :
       < do something with node >
       for each child of node :        # inserire i figli nella CODA
           q.enqueue(child)            (per LIVELLI)
       q.dequeue()                     # estrarre un figlio alla volta
```

ALBERI BINARI DI RICERCA

PROPRIETÀ FONDAMENTALE:

Se v è un nodo di un albero binario di ricerca, sia w un suo nodo figlio.

Se w è nel sottoalbero sinistro di v $\Rightarrow$ $w.key \leq v.key$

Se w è nel sottoalbero destro di v $\Rightarrow$ $w.key > v.key$

Implementano ulteriori operazioni:

- search() : ricerca di un NODO ASSOCIATO ad una CHIAVE
- minimum() : elemento con CHIAVE MINORE
- maximum() : elemento con CHIAVE MASSIMA
- predecessor() : dato un nodo, restituisce quello con la chiave immediatamente precedente
- successor() : nodo con chiave immediatamente successiva
- insert() : inserisce un nuovo nodo con una determinata CHIAVE
- delete() : rimuove nodo

Per ACCEDERE ai NODI IN ORDINE DI CHIAVE $\longrightarrow$ $\boxed{\text{DFS in ORDINE SIMMETRICO}}$

- SEARCH() :

Si può utilizzare la PROPRIETÀ FONDAMENTALE:

```
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)
```

Di fatto, partendo da v, si percorre il ramo che porta al nodo di chiave key, se presente. Il massimo cammino è il ramo più lungo dell'albero, cioè la sua altezza h $\Rightarrow$ COSTO : $\boxed{T(n) = O(h)}$

- **MINIMUM () e MAXIMUM ():**

Sempre sfruttando la proprietà FONDAMENTALE, il MIN è l'elemento più a sinistra, il MAX quello più a destra.

```
TreeMinimum (v):
    while v.left ≠ ⊥:
        v ← v.left
    return v
```

```
TreeMaximum (v):
    while v.right ≠ ⊥:
        v ← v.right
    return v
```

- **SUCCESSOR ():**

Se v ha un sottoalbero destro, banalmente il minimo di tale sottoalbero sarà il suo successore.
Altrimenti, si risale sempre al genitore di volta in volta, "controllando" che si stia andando verso sinistra; non appena si sale a destra, quello è il successore.
Praticamente, in questo secondo caso, il successore di v è il più vicino ANTENATO di v tale che v ∈ sottoalbero sinistro di tale antenato.

```
TreeSuccessor (v):
    if v.right ≠ ⊥ then:                     # CASO 1
        return TreeMinimum (v.right)
    y ← v.parent                             # CASO 2
    while y ≠ ⊥ and x = y.right :            # Fin quando si proviene da un sottoalbero destro
        x ← y
        y ← y.parent
    return y
```

- INSERT ():

```
TREE INSERT (T, z):
    y ← ⊥                          # x.parent
    x ← T.root
    while x ≠ ⊥:
        y ← x
        if z.key < x.key then:
            x ← x.left
        else:
            x ← x.right
    z.parent ← y                   # attacco figlio alla genitore
    if y == ⊥ then:                # Albero vuoto
        T.root ← z                 # Nodo inserito è radice
    else if z.key < y.key then:
        y.left ← z                 # Nodo inserito come figlio Sx
    else:
        y.right ← z                # Nodo inserito come figlio Dx
```

- DELETE ():

```
TREE DELETE (T, z):
    if z.left == ⊥ or z.right == ⊥ then:     # CASI 1 e 2
        y ← z
    else:
        y ← TREE SUCCESSOR (z)               # CASO 3 (ha 2 figli)

    if y.left ≠ ⊥ then:
        x ← y.left
    else:
        x ← y.right

    if x ≠ ⊥ then:                           # x è il nodo figlio da attaccare quando
        x.parent ← y.parent                  # viene eliminato y

SGANCIO y
    if y.parent == ⊥ then:                   # Caso di eliminazione della radice
        T.root ← x
```

SGANCIO y

```
else if y = y.parent.left then:        # Se y era figlio sinistro...
    y.parent.left ← x
else:                                  # ... o figlio destro
    y.parent.right ← x

if y ≠ z then:
    z.key ← y.key                      # Copia dati aggiuntivi

return y
```

RIASSUNTO:

Ci sono da considerare 3 CASI differenti:

(1) Il nodo da eliminare è FOGLIA (non ha figli) → lo si stacca e basta

(2) Il nodo ha 1 solo FIGLIO → Si attacca il figlio al genitore del nodo da eliminare e si restituisce il nodo (y).

(3) Il nodo ha 2 FIGLI →

- Si cerca il SUCCESSORE di z, cioè y.
- x diventa il figlio (di solito destro) di y e viene agganciato ad y.parent, cioè x.parent ← y.parent.
- Si sgancia y, vedendo se y era figlio destro o sinistro (o radice) e si imposta x come figlio destro o sinistro di y.parent, cioè:

```
if y == y.parent.left:
    y.parent.left ← x
...
```

- Si copia il successore di z al posto di z e si restituisce y:

```
if y ≠ z:
    z.key ← y.key
```

Alberi Binari di Ricerca Autobilanciati

I **SELF-BALANCING TREES** sono alberi binari di ricerca che tentano di ribilanciarsi ogni volta che individuano un fattore di sbilanciamento troppo grande dovuto ad un inserimento o ad una rimozione.

Il **BILANCIAMENTO** avviene modificando le relazioni (e le posizioni) dei nodi.
Alcuni esempi sono: AVL Tree, Red-Black Tree, ...

• Red-Black Tree:

Assicurano che un percorso radice-foglia non sia mai più lungo del doppio di ogni altro percorso radice-foglia.

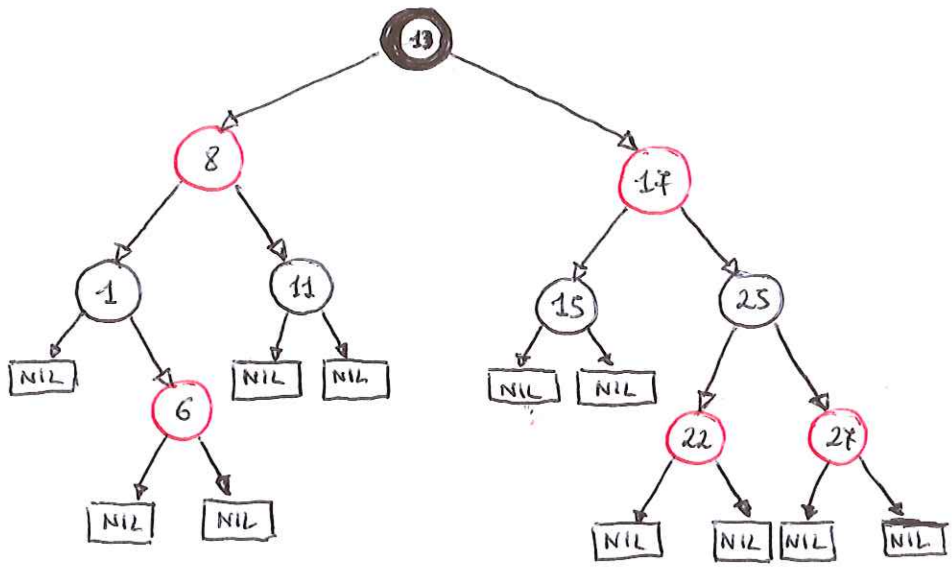
I nodi possono essere ROSSI o NERI, secondo le seguenti regole:

(1) La RADICE è sempre NERA;

(2) Ogni foglia (NIL) è sempre NERA;

(3) Se un nodo è ROSSO $\Rightarrow$ entrambi i figli sono NERI;

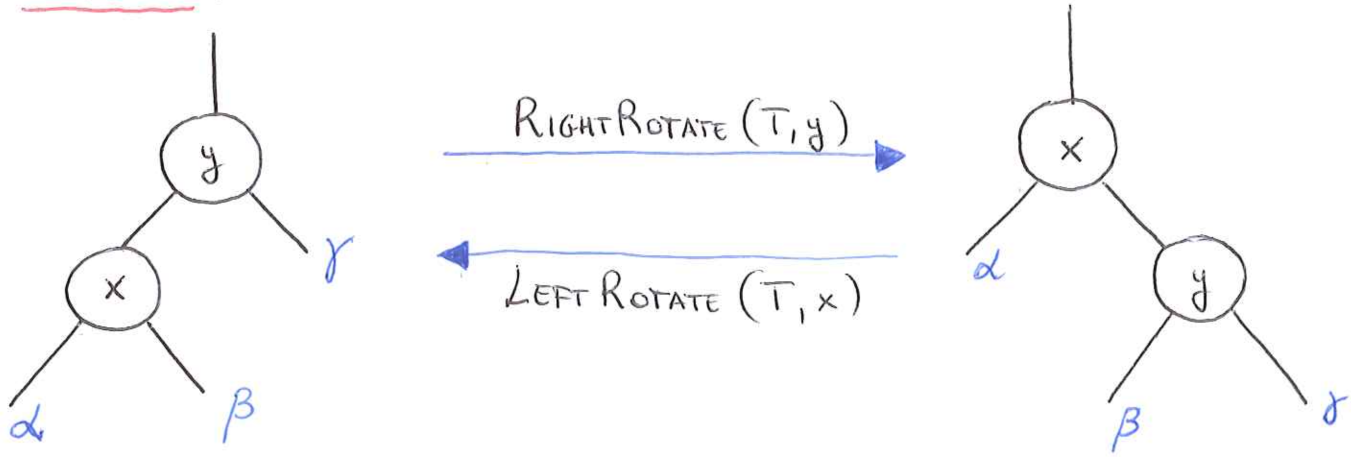
(4) Dato qualsiasi nodo, tutti i percorsi verso le foglie contengono lo stesso numero di nodi neri.



Per autobilanciarsi, fanno uso delle **ROTAZIONI** a destra e a sinistra. Sono operazioni locali che conservano la proprietà fondamentale degli alberi binari di ricerca.

Coinvolgono 2 NODI e i loro SOTTOALBERI DESTRO e SINISTRO.

• Rotazioni:



Si nota che sono una l'inversa dell'altra.

* Nella LeftRotate, α continua ad essere figlio sinistro di x, così come γ continua ad essere figlio destro di y.

L'unico <u>sottoalbero</u> a cambiare è β : $\boxed{\text{da figlio Sx di } y \longrightarrow \text{figlio DX di } x}$

LeftRotate (T, x):

```
y ← x.right
x.right ← y.left                    # β diventa figlio DESTRO di x
~~y.parent~~
y.left.parent ← x
y.parent ← x.parent                 # porta y sopra (diventa figlio del genitore di x)

if x.parent == ⊥ :                  # così per vedere se y sarà figlio dx o sx
    T.root ← y
else if x == x.parent.left :
    x.parent.left ← y
else :
    x.parent.right ← y

y.left ← x                          # x diventa figlio sinistro di y
x.parent ← y                        # collega x al padre y.
```

INSERIMENTO R-B TREE:

RB-INSERT (T, z):

```
y ← ⊥                                # Scorro fino a trovare il posto
x ← T.root                           giusto dove inserire z come foglia
while x ≠ ⊥ :
    y ← x
    if z.key ≤ x.key :
        x ← x.left
    else:
        x ← x.right
z.parent ← y                         # attacco z all'albero
if y == ⊥ :                          # casi per capire se z è figlio dx o sx
    T.root ← z
else if y == y.parent.left :
    y.parent.left ← z
else:
    y.parent.right ← z
z.left ← ⊥
z.right ← ⊥
z.color ← RED                        # ogni nuovo nodo inserito viene impostato a ROSSO
RB-INSERTFIXUP (T, z)                # BILANCIAMENTO (se necessario)
```

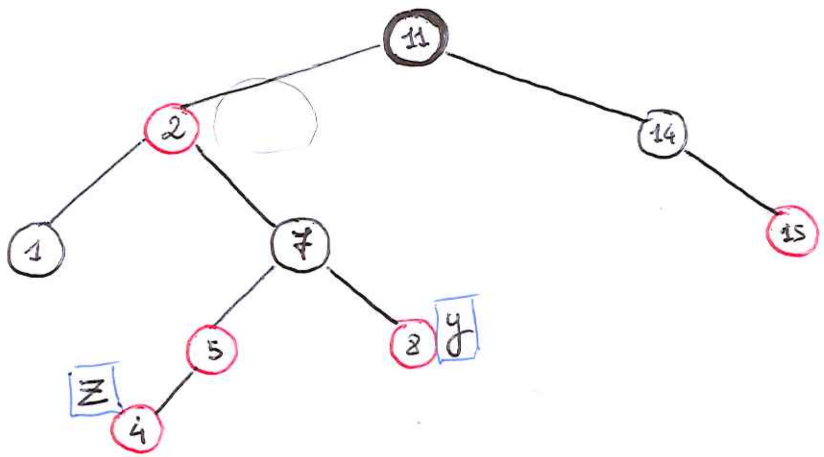
Fixup per l'Inserimento:

Cerca di spostarsi tra 3 CASI verso il bilanciamento. In realtà i casi da coprire sono 6, ma sono SIMMETRICI a seconda se ci troviamo in un sottoalbero destro o sinistro.

```
RB-InsertFixup(T, z):
  while z.parent.color == RED:                      # ci fa muovere tra i 3 casi
     if z.parent == z.parent.parent.left :          # se sottoalbero SINISTRO
        y ← z.parent.parent.right

        if y.color == RED :                         # CASO 1
           z.parent.color ← BLACK
           y.color ← BLACK
           z.parent.parent.color ← RED
           z ← z.parent.parent

        else if z == z.parent.right :               # CASO 2
           z ← z.parent
           LEFTROTATE(T, z)                         # rotazione a sinistra del genitore di z
                                                    (con z)
        else :                                      # CASO 3
           z.parent.color ← BLACK
           z.parent.parent.color ← RED
           RIGHTROTATE(T, z.parent.parent)          # rotazione a destra del nonno di z
                                                    (con il genitore di z)
     else:

           - - -                                    # Caso del sottoalbero destro con
                                                    right e left invertiti
  T.root.color ← BLACK
```

INSERIMENTO RB - TREE

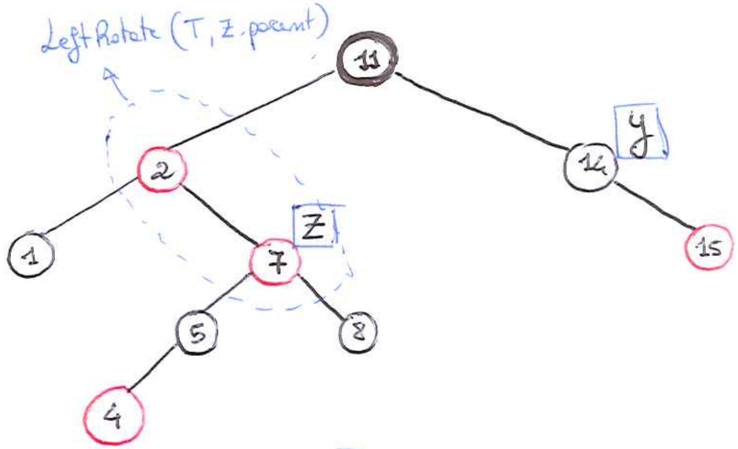
CASO 1



z ← z.parent.parent

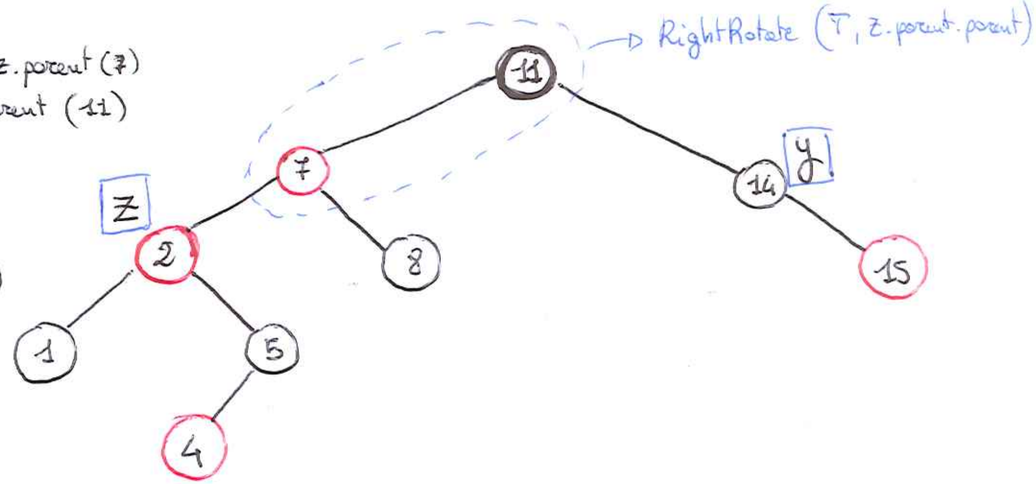
Cambio colori di z.parent, del fratello e del genitore (nonno di z)

CASO 2



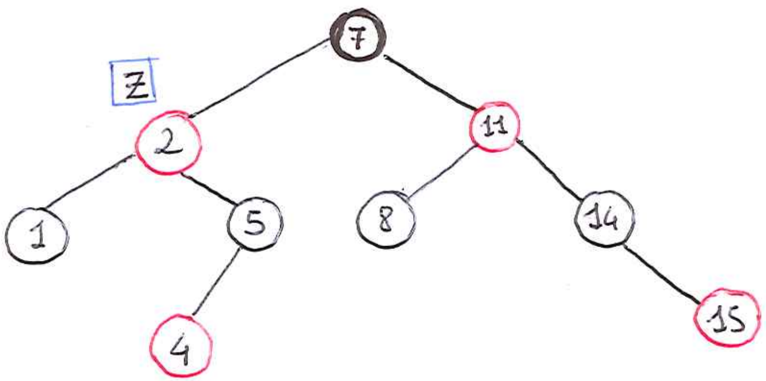
z ← z.parent (cioè 2)
+
LEFT ROTATE (T, 2)

CASO 3



Cambiare colori di z.parent (7) e di z.parent.parent (11)
+
RightRotate (11)

BILANCIATO !



- RIMOZIONE RB-TREE :

```
RB-DELETE(T, z):
  if z.left == ⊥ or z.right == ⊥ :
      y ← z
  else :
      y ← TREESUCCESSOR(z)

  if y.left ≠ ⊥ :
      x ← y.left
  else :
      x ← y.right

  x.parent ← y.parent                  # collegare figlio di y al genitore di y

  if y.parent == ⊥ :                   # scollegare y
      T.root ← x
  else if y == y.parent.left :
      y.parent.left ← x
  else :
      y.parent.right ← x

  if y ≠ z :                           # copia dati aggiuntivi
      z.key ← y.key
  if y.color == BLACK :                # c'è FIXUP SOLO SE il "SOSTITUTO" di
      RB-DELETE FIXUP(T, x)            # z, cioè y è NERO

  return y
```

HEAP

E' una struttura dati formata da un ALBERO BINARIO COMPLETO (tutti i livelli riempiti, tranne l'ultimo), non necessariamente pieno.
Soddisfano la HEAP PROPERTY :

> Per ogni nodo v che non sia la radice, v.parent.Key ≥ v.Key.
> Quindi, la chiave di un nodo è al massimo pari a quella del genitore.

HEAPIFY ()

Dato un nodo v di un heap, HEAPIFY (v) assume che i sottoalberi destro e sinistro di v siano degli heap. Tuttavia, v potrebbe avere una chiave minore di almeno 1 dei figli, violando la HEAP PROPERTY.
HEAPIFY () ribilancia l'heap, ripristinando la proprietà.

```
HEAPIFY (v):
  l ← v.left
  r ← v.right
  if l ≠ ⊥ and l.key > v.key:        # prendo il più grande tra v, l, r
     largest ← l
  else
     largest ← v
  if r ≠ ⊥ and r.key > largest.key :
     largest ← r
  if largest ≠ v :
     < scambia i nodi v e largest >   # scambio v con largest
     HEAPIFY (v)                      # chiamo RICORSIVAMENTE fin quando
                                      # v non coincide con largest
```

* Gli Heap si usano per implementare CODE DI PRIORITA'. Finora abbiamo visto un MaxHeap. In un MinHEAP vale la proprietà opposta : v.parent.key ≤ v.key.
Bisogna implementare l'inserimento di un nuovo elemento e l'estrazione della RADICE, che equivale al getMin () della coda di priorità.

- **HEAPINSERT ( ):**

Si inserisca il nodo come foglia e lo si fa risalire tramite un confronto tra chiavi in posizione giusta.

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

- **DELETEFIRST ( ):**

Rimuovere la RADICE equivale a rimuovere l'elemento con priorità massima. Si trova l'ultima foglia dell'heap e la si mette al posto della radice, poi si chiama HEAPIFY ( ) per rimettere a posto tutto.

```
DELETEFIRST (H):
    lastLeaf ← get the last leaf in the heap      # la più profonda
    ret ← H.root
    < rimpiazzare H.root con lastLeaf >
    HEAPIFY (H.root)                              # chiamo Hepify () sulla radice
    return ret
```

Algoritmo di Floyd: da Array ad Heap

Dato un ARRAY di numeri, viene convertito in un Heap nel seguente modo:

(1) Il vettore viene convertito in un ALBERO BINARIO, sfruttando la rappresentazione tramite array;

(2) Le FOGLIE si considerano già heap $\Rightarrow$ Si parte dal penultimo livello e si applica HEAPIFY() ad OGNI NODO del LIVELLO;

(3) Si risale fino alla RADICE, applicando ripetutamente HEAPIFY().

```
ArrayToHeap(A):
    for i ← len(A) - 1 downto 0:
        HeapifyInPlace(A, i)
```

```
HeapifyInPlace(A, i):
    if isLeaf(A, i):
        return
    J ← getMaxChildIndex(A, i)
    if A[i] < A[J]:
        EXCHANGE(A, i, J)
        HeapifyInPlace(A, J)
```

Analisi della Complessità:

Nel caso peggiore, HeapifyInPlace() fa il massimo numero di scambi. Se abbiamo un ALBERO COMPLETO con $n = 2^k - 1$ nodi, per ogni livello abbiamo:

ULTIMO $\rightarrow \frac{n+1}{2}$ foglie

PENULTIMO $\rightarrow \frac{n+1}{4}$ nodi ...

HeapifyInPlace() su un NODO di LIVELLO i effettua massimo $\boxed{k-i \text{ scambi}}$ (operazione dominante). Il numero di livelli è $\log(n+1) = k$

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

Essendo che: $\sum_{i=2}^{\infty} \frac{i}{2^i} = \frac{3}{2} \Rightarrow \sum_{i=2}^{\log(n+1)} \frac{i}{2^i} < \frac{3}{2}$; Inoltre l'altra sommatoria è negativa e si può togliere (tende $\xrightarrow[n \to +\infty]{}$ per)

$$\Rightarrow T(n) = (n+1) \cdot \sum_{i=2}^{\log(n+1)} \frac{i-1}{2^i} < (n+1) \cdot \frac{3}{2} = \frac{3}{2}n + \frac{3}{2}$$

$$\Rightarrow \boxed{T(n) = O'(n)} \longrightarrow$$

L'ALGORITMO DI FLOYD HA COSTO LINEARE!

Heap Sort

Ordina un array creandone un Heap con l'Algoritmo di Floyd. In particolare è un MaxHeap ed estrae ogni volta la radice (cioè il massimo) e la mette in fondo all'array.

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

Costo: $O(n) + O(\log n!) = \boxed{O(n \log n)}$, con l'approssimazione di Stirling

Insiemi Disgiunti

Sono COLLEZIONI disgiunte di elementi ed ogni collezione ha un ELEMENTO RAPPRESENTATIVO. Una struttura dati per insiemi disgiunti, mantiene una collezione di n insiemi disgiunti DINAMICI.

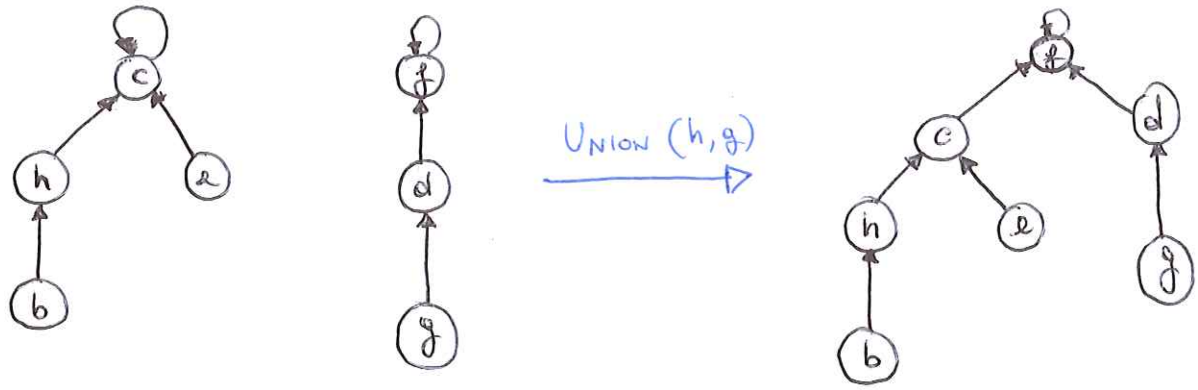
Ci sono le seguenti operazioni fondamentali:

- MakeSet(x): crea un nuovo insieme, con ELEMENTO RAPPRESENTATIVO x
- Union(x, y): unisce i due insiemi contenenti x ed y
- FindSet(x): vede in quale insieme è contenuto x e RESTITUISCE un RIFERIMENTO all'ELEMENTO RAPPRESENTATIVO di tale insieme.

* ESEMPIO DI APPLICAZIONE → Per trovare le componenti connesse di una foresta.

Rappresentazione di Insiemi tramite Alberi:

- La RADICE è l'elemento rappresentativo; ha come genitore sé stesso;
- I collegamenti sono verso l'alto: i figli puntano al padre;
- MAKESET(x) restituisce un albero formato da 1 solo nodo;
- FINDSET(x) naviga verso la RADICE
- UNION(x,y) fa sì che la radice di un insieme punti alla RADICE dell'altro insieme.



La creazione di insiemi più grandi tramite MAKESET() e UNION() può portare a problemi di SBILANCIAMENTO. Si può adottare un approccio EURISTICO.

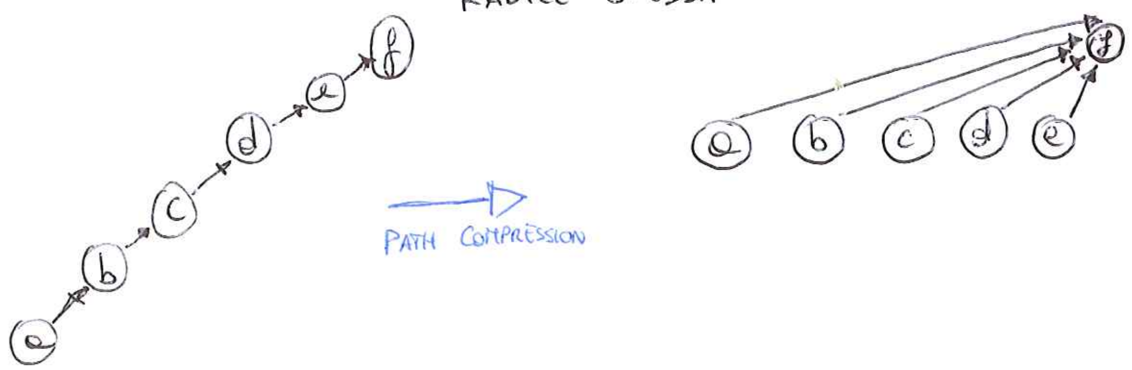
- ### Euristiche: sbilanciamento

Un'EURISTICA è una soluzione che si affida all'INTUITO e all'efficienza pratica, senza seguire un percorso chiaramente dimostrato.

(1) UNION BY RANK: fa sì che la radice dell'albero con meno livelli punti alla radice dell'albero con più nodi.

CONCETTO DI RANK: livello, approssimazione del logaritmo della dimensione dei sottoalberi

(2) PATH COMPRESSION: si utilizza FINDSET() per far risalire i nodi verso la radice, "comprimendo" così il loro percorso fino alla RADICE STESSA



- ALGORITMI:

MAKESET (x):

```
x.parent ← x        # la RADICE ha sé stessa come genitore
x.rank ← 0
```

UNION (x, y):

```
LINK(FINDSET(x), FINDSET(y))        # la RADICE di x punta alla RADICE di y
```

FINDSET (x):

```
if x ≠ x.parent:
    x.parent ← FINDSET(x.parent)        # attacca x al suo elemento rappresentativo
return x.parent
```

LINK (x, y):

```
if x.rank > y.rank:
    y.parent ← x        # attacca y ad x
else:
    x.parent ← y        # attacca x ad y
    if x.rank == y.rank:
        y.rank ← y.rank + 1
```

- ANALISI:

Con PATH COMPRESSION: costo ammortizzato (ACCANTONAMENTI) → MAKESET() : 2 (+1 di credito)
LINK() : 1
FINDSET() : 1

FINDSET() → 1 copre la visita della RADICE e di 1 figlio, tutte le altre compressioni sono pagate dal credito di MAKESET().

Costo di m operazioni → $O(m)$

Con UNION BY RANK: MAKESET() : $O(1)$
LINK() : $O(1)$

Poiché il RANK è un limite superiore all'altezza del sottoalbero, ciascun percorso necessario a trovare l'elemento rappresentativo è $O(\log n)$ → ALTEZZA MASSIMA

⇒ 1 sequenza di MAKESET(), LINK(), FINDSET() → $O(\log n)$

⇒ m sequenze → $O(m \log n)$

* Se si usano insieme UNION BY RANK e PATH COMPRESSION → $O(m \cdot \alpha(n)) \approx O(m)$