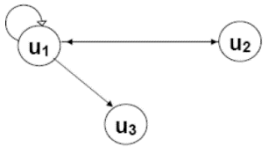
L08.1. ADT ALBERO

Un **grafo orientato G** è una coppia <N, A> dove:

- N è un insieme finito non vuoto di **nodi**;
- A⊆NxN è un insieme finito di coppie ordinate di nodi, detti **archi**.



Se la coppia ordinata di nodi $<u_i, u_j> \in A$, vuol dire che nel grafo vi è un **arco diretto** da nodo u_i al nodo u_j.

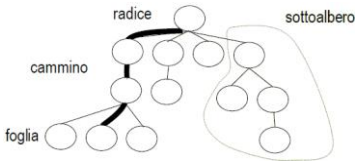
In figura abbiamo che l'insieme N = {u_1, u_2, u_3,}, l'insieme A = {(u_1,u_1),(u_1,u_2),(u_2,u_1),(u_1,u_3)}.

Un **albero** è un tipo particolare di grafo, ma anche una lista può essere considerata un tipo particolare di grafo.

L'albero è una struttura informativa che serve a rappresentare diverse cose, ad esempio organizzazioni gerarchiche di dati o partizioni successive di un insieme in sottoinsiemi disgiunti, un esempio di queste due prime categorie può essere il file system di un sistema operativo, ma può anche rappresentare procedimenti decisionali enumerativi, come l'albero decisionale per rappresentare la complessità degli algoritmi ricorsivi.

Questa struttura albero ha diverse proprietà, come:

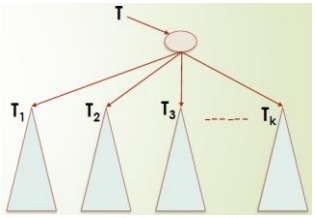
- Ogni nodo ha un unico arco entrante, tranne la **radice**, che non ha archi entranti;
- Ogni nodo può avere zero o più archi uscenti, ed i nodi senza archi uscenti sono detti **foglie**;
- Un arco nell'albero induce una **relazione padre-figlio**;
- A ciascun nodo è solitamente associato un valore, detto **etichetta** del nodo;
- Il **grado** di un nodo è il numero di figli del nodo;
- L'**ordine** dell'albero è il grado max tra tutti i nodi;
- Un **cammino** è una sequenza di nodi $<n_0, n_1, \ldots, n_k>$ dove il nodo n_i è padre del nodo n_{i+1}, per $0 \le i < k$;
- Il **livello** di un nodo è la lunghezza del cammino dalla radice al nodo stesso;
  Definizione ricorsiva: il livello della radice è 0, il livello di un nodo non radice è 1+il livello del padre.
- L'**altezza** dell'albero è la lunghezza del più lungo cammino, parte dalla radice e termina in una foglia.



Un albero è un grafo diretto aciclico, in cui per ogni nodo esiste un solo arco entrante (tranne la radice che non ne ha nessuno). Non tutti i nodi sono collegati tra loro da un cammino, ma se esiste un cammino che va da un nodo u ad un altro nodo v, tale cammino e sicuramente unico, la radice invece è collegata a qualunque altro nodo, quindi esiste sempre un cammino che va dalla radice ad un qualunque altro nodo.

Dato un nodo u, i suoi discendenti costituiscono un albero, detto **sottoalbero** di radice u.

Anche gli alberi hanno una natura ricorsiva, se consideriamo tutte le parti dell'albero collegate alla radice vediamo che ciascuna di esse costituisce un sottoalbero della radice e questo è a sua volta un albero.



ALBERI BINARI:

Sono dei particolari alberi e n-ari in cui ogni nodo può avere 0 o al più 2 figli, distinguiamo due sottoalberi, ovvero il sottoalbero SX e il sottoalbero DX.

Anche gli alberi binari possono essere definiti ricorsivamente:

- **Base**: albero binario vuoto;
- **Passo**: è una terna (s, r, d), dove r è un nodo (la radice), s e d sono alberi binari.

Per la progettazione e l'implementazione degli alberi binari, avremmo come operatori un costruttore bottom-up, che ci consente di costruire un albero a partire da singoli nodi (partendo dalle foglie), degli operatori di selezione, in cui potremo ottenere delle parti dell'albero, e degli operatori di visita, mentre le etichette associate agli alberi potranno essere numeriche, stringhe o qualunque altro tipo di ADT a nostra scelta.

Per l'implementazione degli alberi binari possiamo utilizzare, come abbiamo fatto per le liste, una struttura **auto-referenziale**, questa struttura avrà due puntatori, uno alla radice del sottoalbero SX ed uno al DX, in più avremo l'item che è l'etichetta del nodo. Per rappresentare l'intero albero binario possiamo utilizzare un puntatore che punterà ad una di queste istanze che è la root dell'albero, se l'albero binario è vuoto questo puntatore sarà NULL.

La dichiarazione del tipo nodo sarà la seguente:

```
struct node{
   Item value;              /*etichetta del nodo*/
   struct node *left;       /*puntatore al sottoalbero sinistro*/
   struct node* right;      /*puntatore al sottoalbero destro*/
};
```

All'interno dell'interfaccia, cioè del file .h in cui andremo a realizzare la nostra libreria che realizza l'ADT albero binario, metteremo un puntatore ad una struttura struct node che chiameremo tramite un'istruzione di typedef struct node *Btree, questa variabile punterà al nodo radice dell'albero. Quando andremo ad inizializzare questa struttura all'interno dell'operatore che ci costruisce un nuovo Btree, porremo Btree T = NULL, indicando che l'albero è inizialmente vuoto.

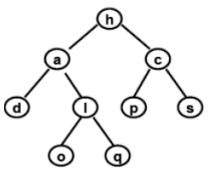
Per aggiungere nodi all'albero binario procederemo in modo bottom-up, cioè aggiungere un nodo alla volta dalle foglie, il motivo è che per ciascun nodo che andremo a creare dovremmo indicare i suoi due sottoalberi e l'etichetta, quindi partendo dalle foglie potremmo indicare che i due sottoalberi sono vuoti e dovremmo soltanto aggiungere l'item all'interno del nodo. Salendo verso la radice invece avremmo già creato i nodi foglia e quindi potremmo agganciarli ai nodi del livello immediatamente superiore e così via fino ad arrivare alla radice dell'albero.

Per **creare un nodo**, dobbiamo rispettare questi passi:

1. **Allocare** la memoria necessaria;
2. **Memorizzare** i dati nel nodo;
3. **Collegare** il sottoalbero SX e DX, già costruiti in precedenza.

Per la stampa delle informazioni contenute in una struttura albero, l'ordine con cui possiamo stampare questi nodi può cambiare, infatti sono noti tre algoritmi di visita che propongono tre modi differenti di visitare i nodi di un albero:

- **Visita in pre-ordine**: si applica ad un albero non vuoto e richiede dapprima l'analisi della radice dell'albero e, poi, la visita, effettuata con lo stesso metodo, dei due sottoalberi, prima il sinistro, poi il destro;
- **Visita in post-ordine**: si applica ad un albero non vuoto e richiede dapprima la visita, effettuata con lo stesso metodo, dei sottoalberi, prima il sinistro e poi il destro, e, in seguito, l'analisi della radice dell'albero;
- **Visita simmetrica**: richiede prima la visita del sottoalbero sinistro (effettuata sempre con lo stesso metodo), poi l'analisi della radice, e poi la visita del sottoalbero destro.



VISITA IN PREORDINE: h a d l o q c p s
VISITA IN POSTORDINE: d o q l a p s c h
VISITA SIMMETRICA: d a o l q h p c s

Sintattica	Semantica
Nome del tipo: BTree Tipi usati: Item, boolean	Dominio: T= nil | T = <N, T1, T2> N ϵ NODO, T1 e T2 sono BTree
newBTree() → BTree	newBTree() → T • Post: T = nil
isEmpty(BTree) → boolean	isEmpty(T) → b • Post: se T=nil allora b = true altrimenti b = false
buildBTree(Btree, Btree, Item) → BTree	buildBTree(T1,T2, e) → T • Pre: e != nil • Post: T = <N, T1, T2>; N ha etichetta e
getBTreeRoot(BTree) → Item	getBTreeRoot(T) → e • Pre: T = <N, T_{left}, T_{right}> non è vuoto • Post: N ha etichetta e
getLeft(BTree) → Btree getRight(BTree) → BTree	getLeft(T) → T' • Pre: T = <N, T_{left}, T_{right}> non è vuoto • Post: T' = T_{left}

btree.c

```
#include "btree.h"
#include "item.h"

struct node{
  Item value;
  struct node *left;
  struct node *right;
};

BTree newTree(){
  return NULL;
}

int isEmptyTree(BTree t){
  if(t==NULL)
    return 1;
  return 0;
}

BTree buildTree(BTree l, BTree r, Item value){
  BTree t=malloc(sizeof(struct node));
  t->left=l;
  t->right=r;
  t->value=value;
  return t;
}

Item getBTreeRoot(BTree t){
  if(!isEmptyTree(t))
    return t->value;
  return NULL;
}

BTree getLeft(BTree t){
  if(!isEmptyTree(t))
    return t->left;
  return NULL;
}

BTree getRight(BTree t){
  if(!isEmptyTree(t))
    return t->right;
  return NULL;
}
```

```
void preOrder(BTree t){
  if(!isEmptyTree(t)){
    outputItem(t->value);
    preOrder(t->left);
    preOrder(t->right);
  }
}

void postOrder(BTree t){
  if(!isEmptyTree(t)){
    postOrder(t->left);
    postOrder(t->right);
    outputItem(t->value);
  }
}

void inOrder(BTree t){
  if(!isEmptyTree(t)){
    inOrder(t->left);
    outputItem(t->value);
    inOrder(t->right);
  }
}
```

btree.h

```
#include "item.h"

typedef struct node *BTree;

BTree newTree();
int isEmptyTree(BTree);
BTree buildTree(BTree, BTree, Item);
Item getBTreeRoot(BTree);
BTree getLeft(BTree);
BTree getRight(BTree);
void preOrder(BTree);
void postOrder(BTree);
void inOrder(BTree);
```

main.c

```
#include <stdio.h>
#include "item.h"
#include "btree.h"

int main(){
  Item h="h",a="a",d="d",l="l",o="o",q="q",c="c",p="p",s="s";
  BTree th,ta,td,tl,to,tq,tc,tp,ts;
  td=buildTree(NULL,NULL,d);  //FOGLIE
  to=buildTree(NULL,NULL,o);
  tq=buildTree(NULL,NULL,q);
  tp=buildTree(NULL,NULL,p);
  ts=buildTree(NULL,NULL,s);
  tl=buildTree(to,tq,l);     //NODI INTERNI
  ta=buildTree(td,tl,a);
  tc=buildTree(tp,ts,c);
  th=buildTree(ta,tc,h);
  printf("preorder: ");
  preOrder(th);
  printf("\npostorder: ");
  postOrder(th);
  printf("\ninOrder: ");
  inOrder(th);
  printf("\n");
}
```

L08.2. ALBERO BINARIO DI RICERCA (BINARY SEARCH TREE)

Questo tipo di albero può essere utilizzato per la realizzazione di insiemi ordinati e supporta operazioni particolarmente efficienti di ricerca, inserimento e cancellazione.

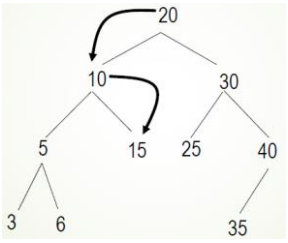
Definiamo questo nuovo ADT, se l'albero non è vuoto ogni elemento del sottoalbero di sinistra precede la radice:

- se prendiamo uno qualunque degli elementi del sottoalbero sinistro vediamo che la sua chiave è minore rispetto a quella della radice;
- se prendiamo un elemento del sottoalbero destro vedremo che questo avrà chiave maggiore di quella della radice;
- Inoltre, sia il sottoalbero sinistro che il sottoalbero destro sono a loro volta degli alberi binari di ricerca quindi godono della stessa proprietà della radice.

L'operatore **Search** ci consente di effettuare una ricerca di un elemento all'interno dell'albero, deve essere efficiente in quanto il BStree è pensato proprio per rendere efficiente la ricerca all'interno di un insieme. Possiamo progettare questo algoritmo di ricerca ricorsivamente:

- **Base**: Se l'albero è vuoto allora restituisce null.
- **Passo**:
  1. Se l'elemento cercato coincide con la radice dell'albero restituisce l'item della radice;
  2. Se l'elemento cercato è minore della radice restituisce il risultato della ricerca dell'elemento nel sottoalbero sinistro;
  3. Se l'elemento cercato è maggiore della radice restituisce il risultato della ricerca dell'elemento nel sottoalbero destro.

Ricerca di 15:



L'operatore **min** ci consente di ottenere l'elemento minimo all'interno di un BStree, lo stesso ragionamento che facciamo adesso per il minimo può essere fatto per il **max**. Possiamo implementare sia una versione ricorsiva che una iterativa di questo algoritmo. Per la prima versione ricorsiva:

- **Base**:
  1. Se l'albero è vuoto allora restituisci null;
  2. Se non esiste un sottoalbero sinistro (destro), ritorna l'item associato alla radice.
- **Passo**: Se esiste un sottoalbero sinistro (destro) effettua la ricerca del minimo (massimo) nel sottoalbero sinistro (destro).

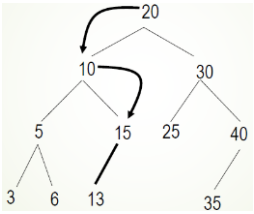
Pseudocodice versione iterativa:

```
Tree_minimum(x)
  while(x.left != NULL)
    x = x.left;
  return x;
```

L'operatore **insert** ci consente di inserire un nodo all'interno della struttura, anche in questo caso lo definiamo ricorsivamente:

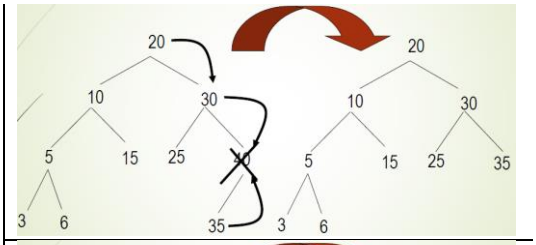
- **Base**: Se l'albero è vuoto allora crea un nuovo albero con un solo elemento.
- **Passo**:
  1. Se l'elemento coincide con la radice non si fa niente (elemento già presente);
  2. Se l'elemento è minore della radice allora lo inserisce nel sottoalbero sinistro;
  3. Se l'elemento è maggiore della radice allora lo inserisce nel sottoalbero destro.

Inserimento di 13:

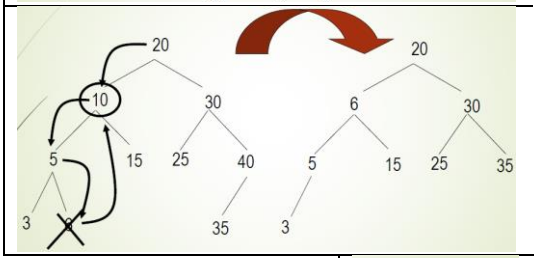


L'operatore **delete** ci consente di eliminare un nodo all'interno della struttura, la prima cosa che facciamo è di ricercare ricorsivamente il nodo da rimuovere, trovato il nodo da rimuovere si presentano due casi:

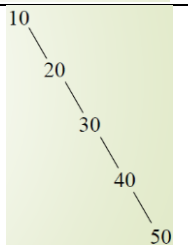
1. Se il nodo ha al più un solo sottoalbero di radice r, si bypassa il nodo da rimuovere e agganciando direttamente il suo unico sottoalbero al padre per poi rimuovere il nodo.



2. Se il nodo ha entrambi i sottoalberi, si sostituisce l'elemento da eliminare con il max nel sottoalbero sinistro (da notare che tale elemento non ha sottoalbero destro, la cui radice altrimenti sarebbe maggiore), alternativamente si cerca e si sostituisce con l'elemento minimo nel sottoalbero destro, infine si chiama ricorsivamente la delete sul sottoalbero sinistro del nodo contenente l'elemento max.



Per quanto riguarda la **complessità** delle operazioni di ricerca, inserimento e cancellazione, hanno tutte e tre la stessa complessità che è funzione dell'altezza dell'albero, quindi questa complessità sarà O(h), poiché nel caso peggiore dovremo attraversare un percorso che parte dalla radice e arriva alla foglia più lontana, quindi attraverseremo l'albero per tutta la sua altezza. Se l'albero binario di ricerca fosse bilanciato, l'altezza dell'albero sarà $\log_2 n$, sarebbe un caso fortunato però non è detto che si verifica.
Ad esempio, creare l'albero e inserire i nodi le cui etichette sono ordinate in modo crescente (10, 20, 30, 40, 50), avremo quindi un albero che somiglia molto ad una lista, quindi l'altezza sarà lineare rispetto al numero di nodi dell'albero ed una qualunque operazione, come l'eliminazione o la ricerca del 50 ci costerà sempre un numero lineare di operazioni, quindi O(n).



Sintattica	Semantica
Nome del tipo: BST Tipi usati: Item, boolean	Dominio: T= nil | T = <N, T1, T2> N ϵ NODO, T1 e T2 sono BST
newBST() → BST	newBST() → T • Post: T = nil
isEmpty(BST) → boolean	isEmpty(T) → b • Post: se T=nil allora b = true altrimenti b = false
getLeft(BST) → BST getRight(BST) → BST	
search(BST, Item) → Item	search(T, e) → e' • Pre: e != nil • Post: e'= e se e ϵ T; e' = nil altrimenti
min(BST) → Item max(BST) → Item	
insert(BST, Item) → BST	insert(T, e) → T' • Post: T' contiene i nodi di T con l'aggiunta di e
delete(BST, Item) → BST	delete(T, e) → T' • Pre: T non è vuoto • Post: T' = T – {e}

bst.c

```
#include "item.h"
#include "bst.h"
#include "queue.h"

struct node {
  Item value;
  struct node *left;
  struct node *right;
};

BST newBST(){  return NULL; }

int isEmptyBST(BST t){
  if(t==NULL)    return 1;
  else
    return 0;
}

BST getLeft(BST t){
  if(isEmptyBST(t))    return NULL;
  else    return t->left;
}
BST getRight(BST t){
  if(isEmptyBST(t))    return NULL;
  else    return t->right;
}

Item getItem(BST t){
  if(isEmptyBST(t))    return NULL;
  else    return t->value;
}

Item search(BST t, Item elem){
  if(isEmptyBST(t))    return NULL;
  else{
    int c;
    c=cmpItem(elem,t->value);
    if(c<0)       return search(t->left,elem);
    else if(c>0)   return search(t->right,elem);
    else    return t->value;
  }
}

Item min(BST t){
  if(isEmptyBST(t))    return NULL;
  else if(t->left==NULL)    return t->value;
  else return min(t->left);
}
Item max(BST t){
  if(isEmptyBST(t))    return NULL;
  else if(t->right==NULL) return t->value;
  else return max(t->right);
}
```

```
void insertBST(BST *t, Item elem){
  if(isEmptyBST(*t)){
    *t=malloc(sizeof(struct node));
    (*t)->value=elem;
    (*t)->right=NULL;
    (*t)->left=NULL;
  }
  else if(cmpItem(elem,(*t)->value)<0)
    insertBST(&((*t)->left),elem);
  else if(cmpItem(elem,(*t)->value)>0)
    insertBST(&((*t)->right),elem);
}

Item deleteBST(BST *tree, Item elem){
  if(isEmptyBST((*tree))) return NULL;
  else{
    int c;
    c=cmpItem(elem,(*tree)->value);
    if(c<0) return deleteBST(&((*tree)->left), elem);
    else if(c>0)  return deleteBST(&((*tree)->right), elem);
    else{
      BST temp = NULL;
      if(isEmptyBST((*tree)->right)){
        temp = *tree;
        *tree = temp->left;
        Item it = temp->value;
        free(temp);
        return it;
      }
      else if(isEmptyBST((*tree)->left)){
        temp = *tree;
        *tree = temp->right;
        Item it = temp->value;
        free(temp);
        return it;
      }
      else{
        Item it = max((*tree)->left);
        Item it2 = (*tree)->value;
        (*tree)->value = it;
        deleteBST(&((*tree)->left), it);
        return it2;
      }
    }
  }
}

void visitLayers(BST t) {
   if(isEmptyBST(t)) return;
   Queue q = newQueue();
   enqueue(q,t);
   while(!isEmptyQueue(q)){
      BST node = dequeue(q);
      outputItem(node->value);
      if(node->left != NULL) enqueue(q,node->left);
      if(node->right != NULL) enqueue(q,node->right);
   }
}
```

bst.h

```
#include "item.h"

typedef struct node *BST;

BST newBST();
int isEmptyBST(BST);
BST getLeft(BST);
BST getRight(BST);
Item getItem(BST);
Item search(BST, Item);
Item min(BST);
Item max(BST);
void insertBST(BST *, Item);
Item deleteBST(BST *, Item);
void visitLayers(BST);
```

L08.3. AVL

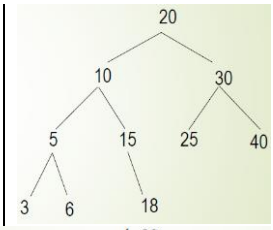
Per rendere efficienti gli alberi binari di ricerca possiamo utilizzare una particolare implementazione, che si chiama alberi AVL.

ALBERO BILANCIATO E ALBERO Δ-BILANCIATO:

Un albero binario di ricerca si dice Delta bilanciato se per ogni nodo la differenza in valore assoluto tra le altezze dei suoi due sottoalberi è minore uguale a Delta, per Δ=1 si parla di alberi bilanciati.

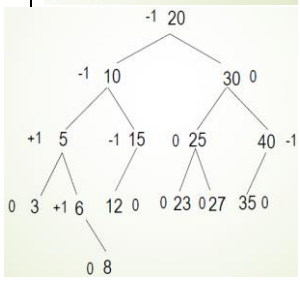
L'albero a destra è bilanciato poiché le altezze dei suoi due sottoalberi differiscono di un solo livello.

Un albero Delta bilanciato è particolarmente favorevole dal punto di vista dell'efficienza poiché si può dimostrare che la sua altezza è $\Delta+\log_2 n$, essendo Delta una costante le operazioni su questo albero binario di ricerca hanno complessità logaritmica.



Passiamo a parlare degli alberi **AVL**, che sono un esempio di alberi bilanciati, per evitare che l'albero non sia bilanciato dobbiamo aggiungere un ulteriore dato ad ogni nodo, in particolare questo dato sarà un marcatore che può assumere i seguenti valori:

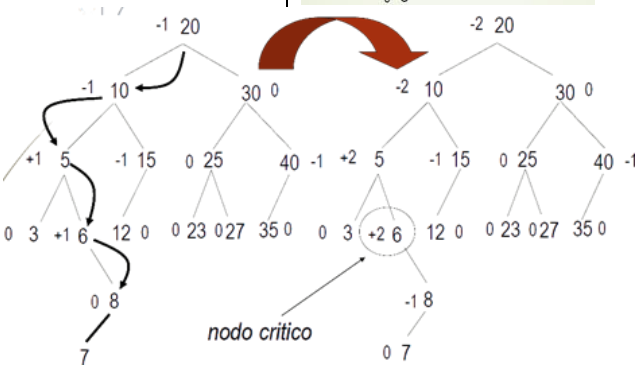
- -1, se l'altezza del sottoalbero sinistro è maggiore (di 1) dell'altezza del sottoalbero destro;
- 0, se l'altezza del sottoalbero sinistro è uguale all'altezza del sottoalbero destro;
- +1, se l'altezza del sottoalbero sinistro è minore (di 1) dell'altezza del sottoalbero destro.



Una operazione di inserimento o di cancellazione può provocare uno sbilanciamento dell'albero, quindi aggiornando sempre questi marcatori troveremo che un marcatore si troverà ad avere il valore +2 o -2, indicando che l'albero non è più bilanciato.

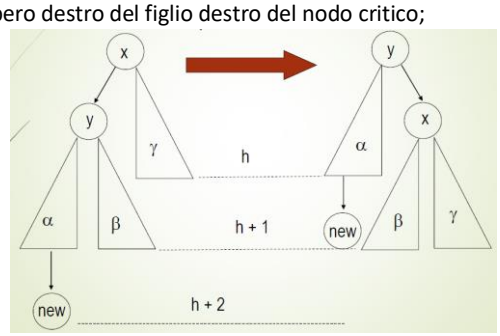
Se questo avviene bisogna ribilanciare l'albero, esistono delle **operazioni di rotazione** che possono essere **semplice** oppure **doppia**.

Si applicano queste operazioni sul nodo x a profondità massima che presenta un non bilanciamento, tale nodo viene detto **nodo critico** e si trova sul percorso che va dalla radice al nodo inserito oppure al nodo cancellato.

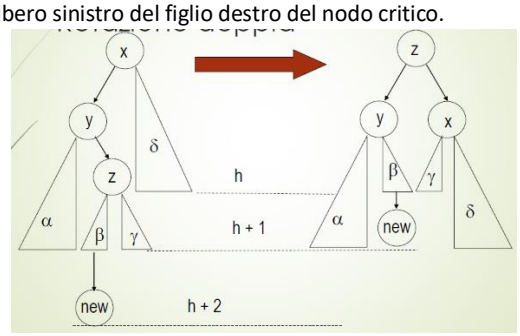


Dopo lo sbilanciamento dovremmo quindi eseguire un'operazione di rotazione sul nodo critico, abbiamo due tipi di rotazione:

- **Rotazione semplice**, quando avviene un inserimento nel sottoalbero sinistro del figlio sinistro del nodo critico, oppure inserimento nel sottoalbero destro del figlio destro del nodo critico;



- **Rotazione doppia**, quando avviene un inserimento nel sottoalbero destro del figlio sinistro del nodo critico, oppure inserimento nel sottoalbero sinistro del figlio destro del nodo critico.



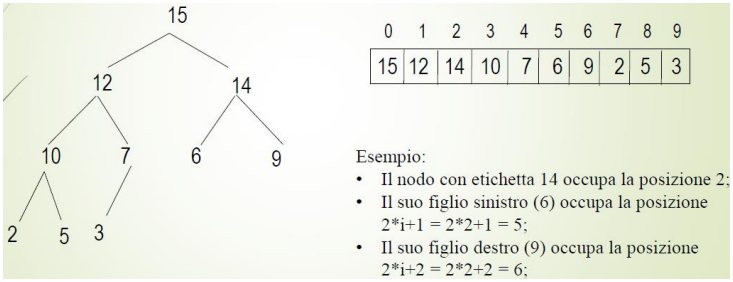
L08.4. CENNI SUGLI HEAP

Un **heap** è un albero binario bilanciato con le seguenti proprietà:

- Le foglie (nodi a livello h) sono tutte addossate a sinistra;
- Ogni nodo v ha la caratteristica che l'informazione ad esso associata è la più grande tra tutte le informazioni presenti nel sottoalbero che ha v come radice. Quindi la radice è più grande di tutti gli altri nodi contenuti nei suoi sottoalberi e questo vale per tutti i suoi sottoalberi.

Uno heap ha come principale applicazione la realizzazione delle **code a priorità** che sono degli ADT in cui possiamo inserire degli elementi oppure rimuovere soltanto l'elemento massimo, cioè quello che si trova nella radice.

L'heap si presta particolarmente bene ad essere rappresentato realizzato tramite un Array, i nodi sono disposti all'interno dell'array in ordine di livello e la radice occupa la posizione 0 dell'array, poi se un nodo occupa la posizione i il suo figlio sinistro occupa la posizione 2*i+1 e il suo figlio destro occupa la posizione 2*i+2. Possiamo passare molto rapidamente, con un semplice calcolo sugli indici, da un nodo ai suoi due sottoalberi.

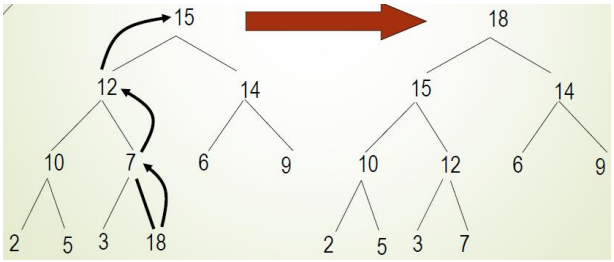


L'operazione di **inserimento** in un Heap, viene fatto in ordine di livello, i passi sono:

- Si inserisce il nuovo nodo come ultima foglia;
- Il nodo inserito risale lungo il percorso che porta alla radice per individuare la posizione giusta.

Ad esempio, inseriamo il numero 18 →

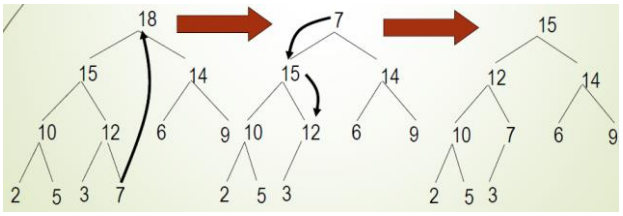
Il 18 è maggiore del suo nodo genitore, cioè il 7, e abbiamo violato la proprietà, facciamo risalire il 18 finché non sarà di nuovo bilanciato.



L'operazione di **rimozione** in un Heap si elimina sempre la radice, poiché nella realizzazione che viene utilizzata per realizzare le code a priorità, la cancellazione è sempre relativa nodo con valore massimo.

Quello che si fa è porre l'ultima foglia al posto della radice, quindi si scambia l'informazione contenuta nella radice con la maggiore dei suoi due sottoalberi e si ripete il procedimento fino ad arrivare ad una foglia.

Ad esempio, cancelliamo il 18, prendiamo l'ultima foglia e sostituiamola al 18, quello che otterremo è uno heap con un nodo in meno, violando il bilanciamento.



Per ribilanciare scambieremo il nodo 7 con il maggiore dei suoi due figli, ovvero il 15, e metteremo il 7 al posto del 15, ancora una volta scambieremo il 7 con il maggiore dei suoi due figli che in questo caso è il 12. A questo punto poiché il 7 è maggiore del 3 avremmo che l'heap è sistemato.