Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Strutture dati ad albero

Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
IL PRESENTE MATERIALE È RISERVATO AL PERSONALE DELL'UNIVERSITÀ DI BOLOGNA E NON PUÒ ESSERE UTILIZZATO AI TERMINI DI LEGGE DA ALTRE PERSONE O PER FINI NON ISTITUZIONALI

STRUTTURE DATI NON SEQUENZIALI

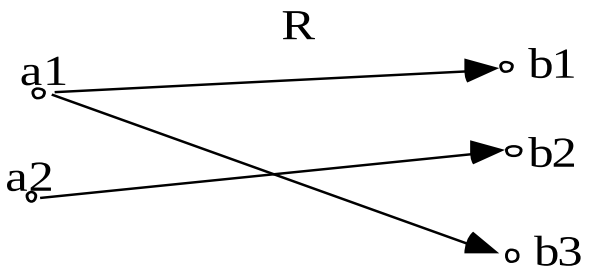
- Le strutture dati sequenziali (liste, array..) risolvono un ampio ventaglio di problemi, ma sono *poco adatte* se
  - le operazioni più frequenti implicano *accesso non sequenziale*
  - sono coinvolte *grandi quantità* di dati
- Quando sono coinvolte grandi quantità di dati o è importante mantenere traccia delle relazioni fra dati e/o si vuole rendere efficiente la ricerca dei dati, diviene cruciale disporre di *strutture dati bidimensionali*
  - le mappe, con la loro struttura tabellare, sono un tipico esempio
- Gli *alberi* costituiscono un ulteriore tipo di struttura dati non sequenziale, di larghissimo uso per molteplici impieghi

ALBERI

- Formalmente, un *albero* è un *grafo orientato aciclico* con alcune particolari proprietà.

- A sua volta, un *grafo* rappresenta *relazioni binarie su insiemi di elementi.*

Una *relazione* fra due insiemi A,B è un sottoinsieme R del prodotto cartesiano A × B.
Qui A = { a1, a2 } e B={ b1, b2, b3 }.



- Caso particolare: il *prodotto cartesiano A x A*, ossia una relazione di un insieme su se stesso.

GRAFI

- Un *grafo orientato* è una struttura $G = \langle A,R \rangle$ dove:
  - A è un insieme non vuoto di *nodi*
  - R è un insieme di *archi orientati* tali che se $\langle a_i,a_j \rangle \in R$, allora c'è un *arco orientato* da a_i verso a_j

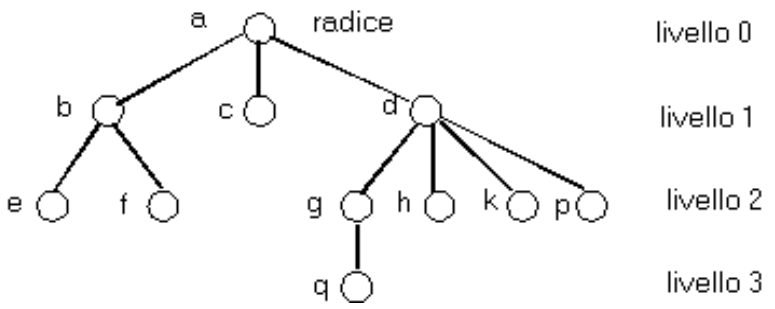
NOMENCLATURA

- Grado di ingresso (uscita) del nodo a_i: è il numero di archi che hanno a_i come nodo *finale (iniziale)*.
- Grafo completo: $\forall a_i,a_j \in A$ esiste un arco da a_i verso a_j (la relazione R coincide col prodotto cartesiano AxA, ossia è una *relazione totale*)
- Cammino da α a β: sequenza di nodi a_0, $a_1 \ldots a_N$ tale che $N \geq 0$, $a_0 = \alpha$, $a_N = \beta$, e $\langle a_i,a_j \rangle \in R$ per ogni $0 \leq i \leq N-1$.
  - cammino semplice: i nodi del cammino sono distinti
  - cammino ciclico (o *ciclo*): quando $\alpha = \beta$
  - cammino aciclico: non vi sono cicli.

DAI GRAFI AGLI ALBERI

- Un *albero* è un *grafo orientato aciclico* tale che
  - esiste un nodo (radice) con grado d'ingresso 0
  - ogni altro nodo ha grado d'ingresso 1

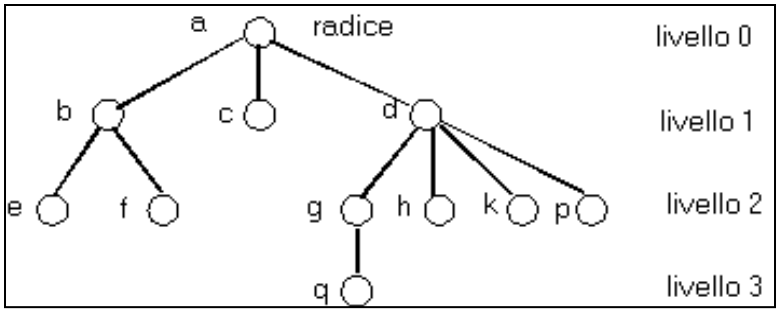
- I nodi con grado di uscita 0 si dicono *foglie*
- La lunghezza del cammino dalla radice a un nodo si dice *livello* di quel nodo.
- Il cammino più lungo dalla radice a una foglia si dice *altezza* dell'albero.
- Se un arco collega il nodo α al nodo β, il nodo α si dice *nodo padre* di β,
- In tal caso, il nodo β detto *nodo figlio* (o *discendente diretto*) di α.



Poiché *in un albero* il grado d'ingresso di ogni nodo è noto a priori, in luogo di "*grado di uscita*" si dice spesso semplicemente "*grado*".

GRAFI e ALBERI

- Conseguenze:
  - esiste *esattamente un cammino* (semplice) dalla radice a qualsiasi altro nodo
  - tranne la radice, tutti i nodi hanno *esattamente un padre*
  - un padre può avere zero o più figli
  - tra i figli di un nodo esiste una *relazione d'ordine* che distingue il 1° nodo, il 2° nodo, etc. (disegnati solitamente da sinistra a destra)



ALBERI

PROPRIETÀ SALIENTI

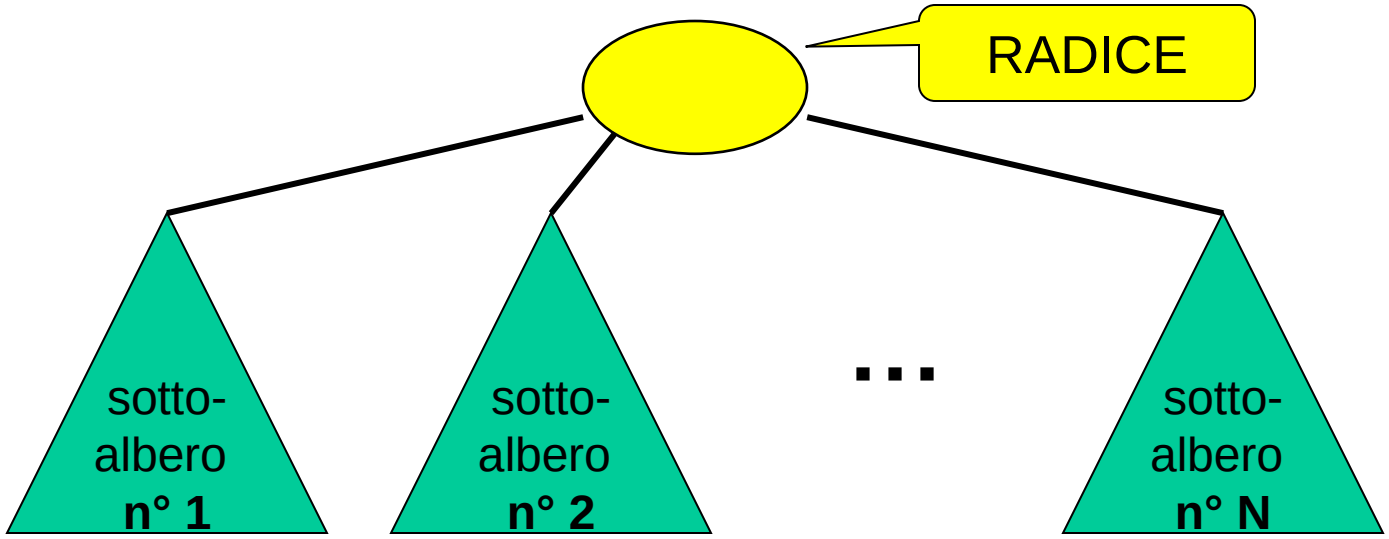
- Un albero rappresenta una *relazione d'ordine parziale* sull'insieme A
- Un albero può essere usato per rappresentare *relazioni gerarchiche* fra entità di A
- I *discendenti* di un dato nodo X si calcolano considerando dapprima i figli di X, poi i figli dei figli, indi i figli di questi, ecc., fino alle foglie.

CATEGORIE DI ALBERI

- Gli alberi si distinguono in base *al numero massimo di figli* che *ogni* loro nodo può avere.
- In un *albero binario* ogni nodo ha *al più due figli* (ma può averne uno solo o nessuno)
- In un *albero ternario,* ogni nodo ha *al più tre figli* (ma può averne meno, o nessuno)
- …
- In un *albero N-ario* ogni nodo ha *al più N figli*.

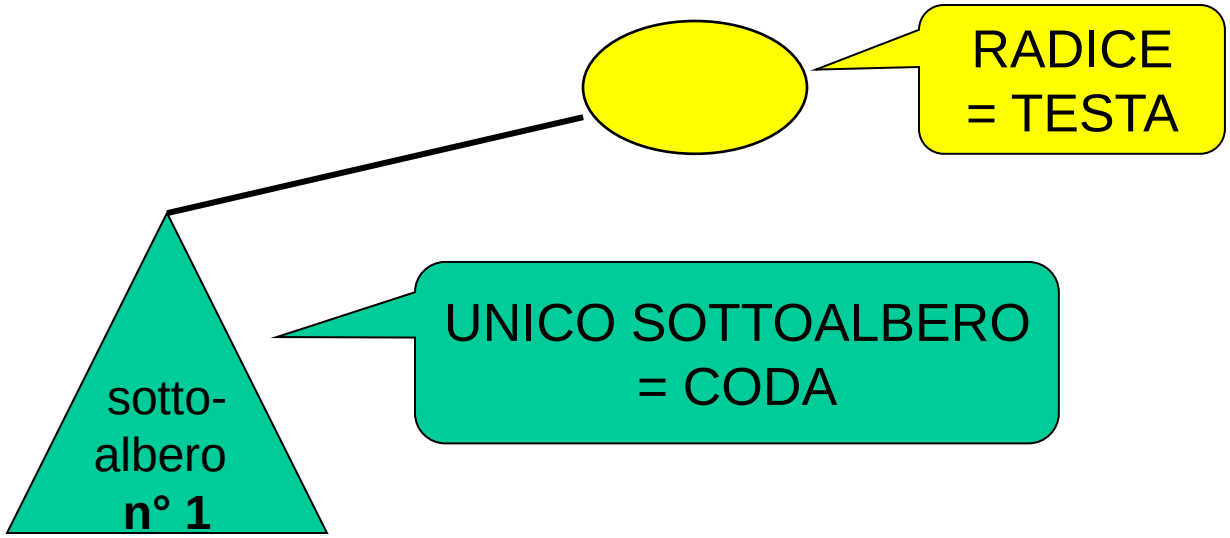
ALBERI COME STRUTTURE RICORSIVE

- Un albero costituisce una struttura *naturalmente ricorsiva*
  - a parte la radice, i nodi si possono ripartire in *N insiemi disgiunti*
  - ogni sottoinsieme comprende *un figlio della radice e i suoi discendenti*
  - dunque, ognuno di questi sottoinsiemi individua un *sottoalbero*



LISTE COME ALBERI UNARI

- Se N=1, l'albero degenera in una lista:
  - a parte la radice, detta in questo caso *testa*, i nodi appartengono a un solo sotto-insieme
  - c'è quindi un solo sottoalbero, detto *coda*



NAVIGARE NELLE STRUTTURE DATI

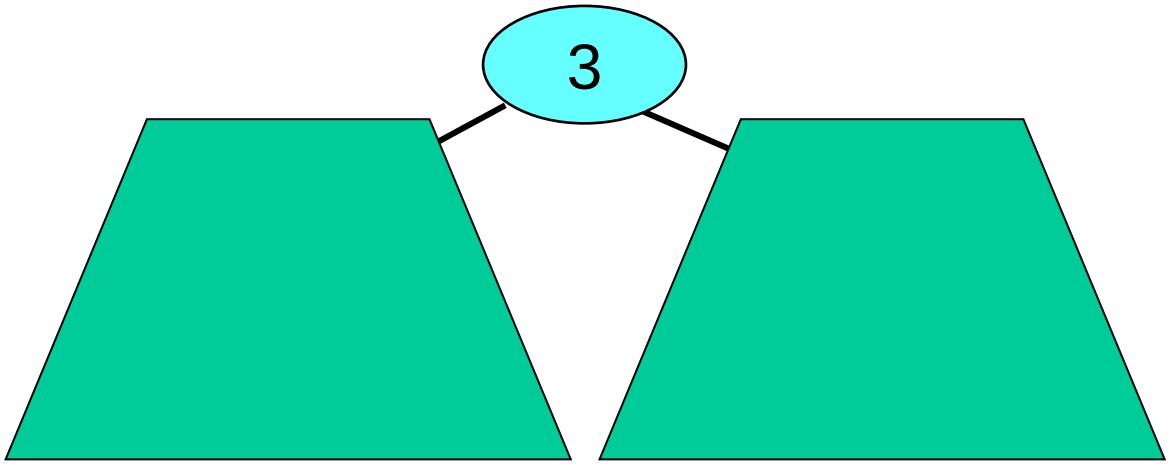
- Si può *navigare* in una struttura *sequenziale* in due modi:
  1. approccio iterativo
     - si parte dal primo elemento
     - si passa via via al successivo
  2. approccio ricorsivo
     - si opera sulla testa della lista (un singolo elemento)
     - si richiama ricorsivamente l'operazione sulla *sottolista-coda*
- Ma in una struttura dati *non sequenziale …?*
  - cade il concetto di "sequenza naturale" degli elementi, quindi *l'approccio iterativo perde di senso*
  - l'approccio ricorsivo invece resta valido, con opportuna estensione

VISITA DI UN ALBERO

- Con il termine *visita* si intende appunto *percorrere l'albero* secondo un qualche criterio che garantisca comunque di *transitare una e una sola volta in ogni nodo*
- Approccio ricorsivo: *ordine anticipato (pre-order)*
  - prima si opera sulla radice
  - poi si richiama ricorsivamente l'operazione sui *vari sottoalberi*
- Approccio ricorsivo: *ordine posticipato (post-order)*
  - prima si richiama ricorsivamente l'operazione sui *vari sottoalberi*
  - solo alla fine si opera sulla radice
- Questi approcci agiscono tutti *scendendo in profondità* nell'albero (*depth-first*)

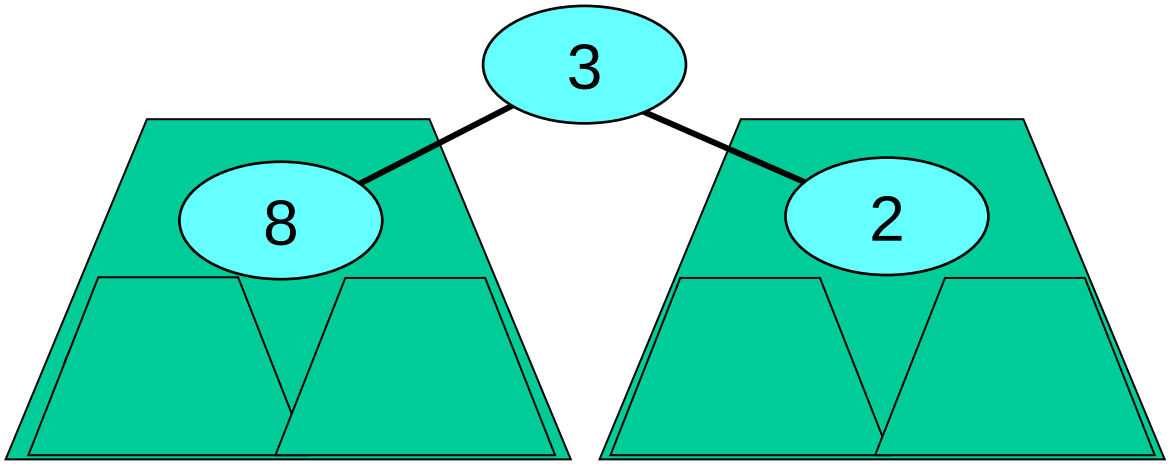
ESEMPIO (1/4)

- Nel caso dell'albero sotto illustrato:
  - visita in preorder (prima la radice, poi i sottoalberi): {3, *sinistro*, *destro*}
  - visita in postorder (prima i sottoalberi, poi la radice): {*sinistro*, *destro,* 3}



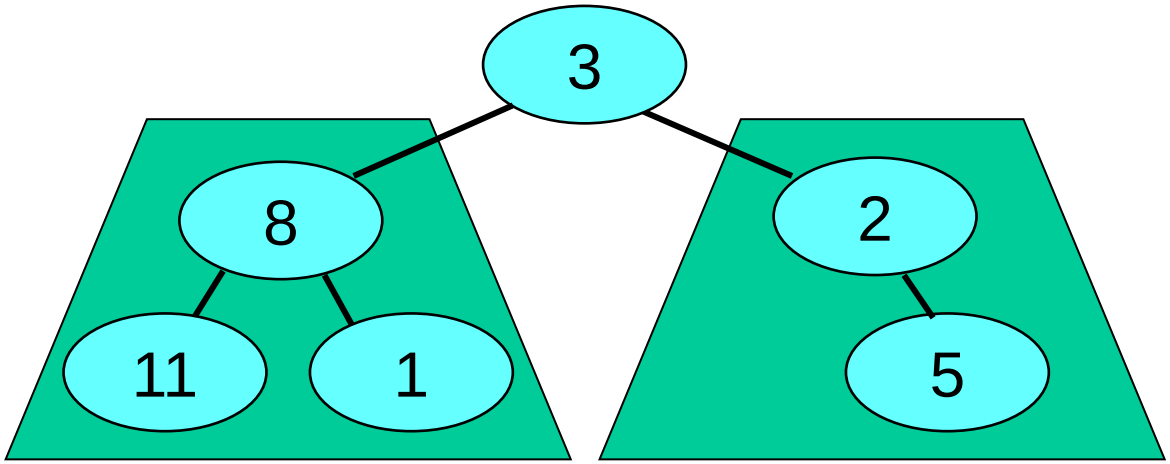
ESEMPIO (2/4)

- Ora sviluppiamo i sottoalberi (non importa l'ordine):
  - visita in preorder (prima la radice, poi i sottoalberi):
    {3, *sinistro*, *destro*} = {3, {8,{sx,dx}}, {2,sx,dx}} =
  - visita in postorder (prima i sottoalberi, poi la radice):
    {*sinistro*, *destro,* 3} = { {{sx,dx},8}, {{sx,dx},2}, 3}



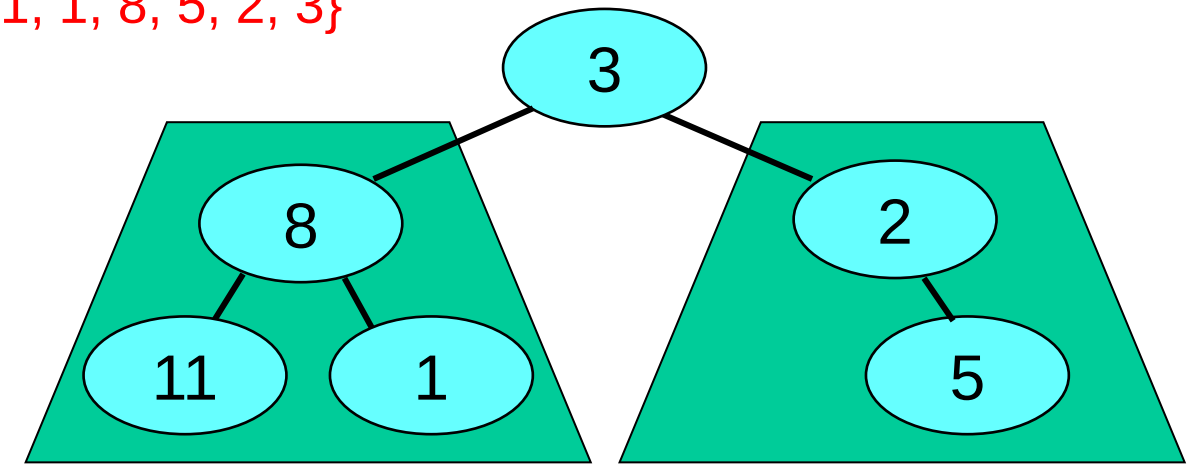
ESEMPIO (3/4)

- Procediamo quindi al terzo ed ultimo livello:
  - visita in preorder (prima la radice, poi i sottoalberi):
    {3, *sinistro*, *destro* } = {3, {8,{11, 1}}, {2, {5} }}
  - visita in postorder (prima i sottoalberi, poi la radice):
    {*sinistro*, *destro,* 3} = { {{11,1},8}, {{5},2}, 3}



ESEMPIO (4/4)

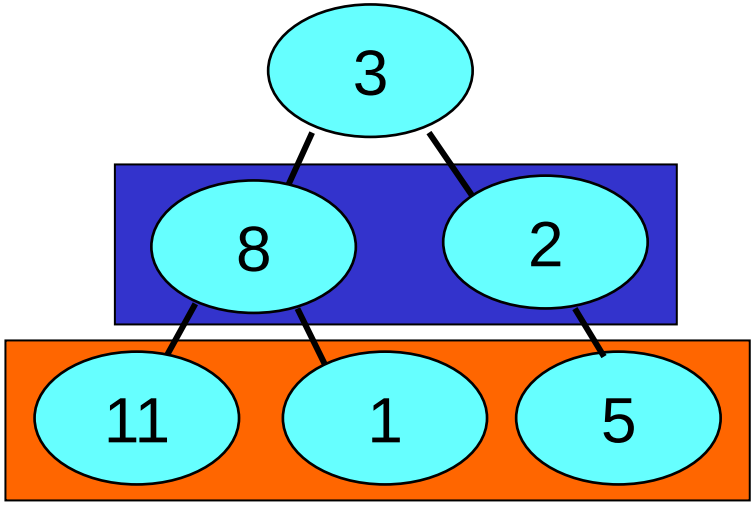
- Procediamo quindi al terzo ed ultimo livello:
  - visita in preorder (prima la radice, poi i sottoalberi):
    {3, *sinistro*, *destro* } = {3, {8,{11, 1}}, {2, {5} }} =
    {3, 8, 11, 1, 2, 5}
  - visita in postorder (prima i sottoalberi, poi la radice):
    {*sinistro*, *destro,* 3} = { {{11,1},8}, {{5},2}, 3} =
    {11, 1, 8, 5, 2, 3}



ALTERNATIVA: VISITA IN AMPIEZZA

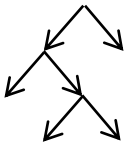
- Anziché scendere in profondità, si può pensare di visitare un albero *in ampiezza (breadth-first):*
  - prima la radice (livello 0)
  - poi tutti i nodi del 1° livello,
  - poi tutti quelli del 2° livello,
  - etc.

- Si ottiene allora:
  - {3, *livello1*, *livello2*} =
    = {3, {8, 2}, {11,1,5} } =
    = {3, 8, 2, 11, 1, 5}

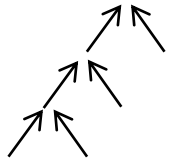


LE TRE VISITE A CONFRONTO

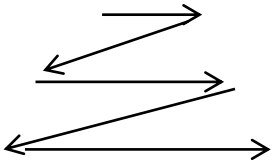
- depth-first, pre-order {3, 8, 11, 1, 2, 5}

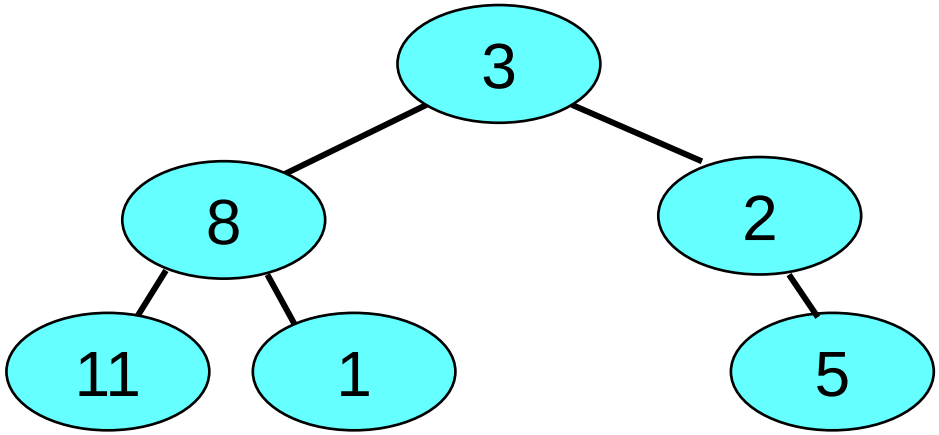


- depth-first, post-order {11, 1, 8, 5, 2, 3}



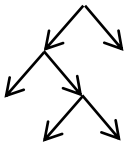
- breadth-first {3, 8, 2, 11, 1, 5}



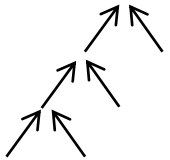


LE TRE VISITE A CONFRONTO

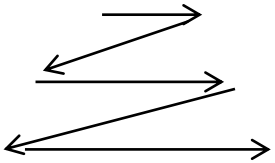
- depth-first, pre-order {3, 8, 11, 1, 2, 5}



- depth-first, post-order {11, 1, 8, 5, 2, 3}



- breadth-first {3, 8, 2, 11, 1, 5}

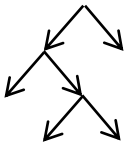


Se l'albero è realizzato con puntatori:

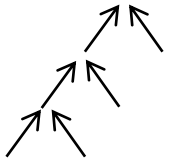
- le visite in *profondità* sono *facili* da realizzare
- la visita in *ampiezza* è *più difficile* da realizzare

LE TRE VISITE A CONFRONTO

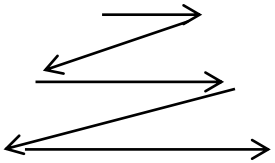
- depth-first, pre-order {3, 8, 11, 1, 2, 5}



- depth-first, post-order {11, 1, 8, 5, 2, 3}



- breadth-first {3, 8, 2, 11, 1, 5}



Però, se l'albero è infinito *(es. albero di soluzioni):*

- le visite in *profondità* sono *incomplete*
- la visita in *ampiezza* è comunque *completa*

ESEMPI IN JAVA CON `TreeItem`

- **`TreeItem`** è una classe di JavaFX che modella alberi Java
  - pur essendo pensata per lavorare in tandem con il componente grafico **`TreeView`**, è in realtà indipendente dalla grafica
  - può essere utile per fare qualche rapido esperimento
- **`TreeItem<T>`** è un *nodo* che contiene un *valore* di tipo **`T`**
  - l'albero si costruisce *nodo per nodo*
  - poiché non è prestabilito il numero di figli possibili, ogni nodo mantiene al suo interno la *lista* dei suoi figli
  - il metodo **`getChildren`** consente di recuperare tale lista
  - è quindi alla lista che vanno aggiunti, col metodo **`add`**, gli altri nodi

UN ALBERO DI ESSERI VIVENTI

1. costruzione nodo radice

Java

```
TreeItem<String> root = new TreeItem<>("Esseri viventi");
```

2. costruzione e aggancio nodi di primo livello

```
TreeItem<String> animali = new TreeItem<>("Animali");
TreeItem<String> vegetali = new TreeItem<>("Vegetali");
root.getChildren().add(animali);
root.getChildren().add(vegetali);
```

3. aggancio nodi foglie (di secondo livello)

```
animali.getChildren().add(new TreeItem<>("Pesci"));
animali.getChildren().add(new TreeItem<>("Mammiferi"));
vegetali.getChildren().add(new TreeItem<>("Graminacee"));
vegetali.getChildren().add(new TreeItem<>("Betullacee"));
```

UN ALBERO DI ESSERI VIVENTI

Java

1. costruzione nodo radice

```
TreeItem<String>                                    Esseri viventi");
```

2. costruzione e ag

```
TreeItem<String                                    >("
TreeItem<String> vegetali = new TreeItem<>(
root.getChildre
root.getChildre
```

3. aggancio nodi fo

```
animali.getChildren().add(new TreeItem<>("Pesci"));
animali.getChildren().add(new TreeItem<>("Mammiferi"));
vegetali.getChildren().add(new TreeItem<>("Graminacee"));
vegetali.getChildren().add(new TreeItem<>("Betullacee"));
```

Un albero di **`TreeItem`** può essere mostrato in una **`TreeView`**

`TreeView` è molto simile a **`ListView`**

Siete liberi di esplorarlo ☺

VISITA IN PREORDER

Accumuliamo il risultato in uno **StringBuilder**:

Java

```
private static <T> void preorder(
   TreeItem<T> root, StringBuilder sb, String separator) {
    if (root != null) {
       sb.append(root.getValue());
       List<TreeItem<T>> children = root.getChildren();
       if (children != null) {
         for (TreeItem<T> child : children) {
              sb.append(separator);
              preorder(child, sb, separator);
         }
       }
   }
}
```

Invocazione lato cliente:

```
StringBuilder sb = new StringBuilder();
preorder(root, sb, ", ");
System.out.println("preorder: " + sb);
```

VISITA IN PREORDER

Accumuliamo il risultato in uno **StringBuilder**

Java

```
private static <T> void preorder(
   TreeItem<T> root, StringBuilder sb, String separator) {
    if (root != null) {
       sb.append(root.getValue());
       List<TreeItem<T>> children = root.getChildren();
       if (children != null) {
         for (TreeItem<T> child : children) {
              sb.append(separator);
              preorder(child, sb, separator);
         }
       }
    }
}
```

```
preorder: Esseri viventi, Animali, Pesci, Mammiferi, Vegetali, Graminacee, Betul
lacee
```

VISITA IN POSTORDER

Accumuliamo il risultato in uno **StringBuilder**

Java

```
private static <T> void postorder(
   TreeItem<T> root, StringBuilder sb, String separator) {
    if (root != null) {
       List<TreeItem<T>> children = root.getChildren();
       if (children != null) {
         for (TreeItem<T> child : children) {
              postorder(child, sb, separator);
              sb.append(separator);
         }
       }
       sb.append(root.getValue());
  }
}
```

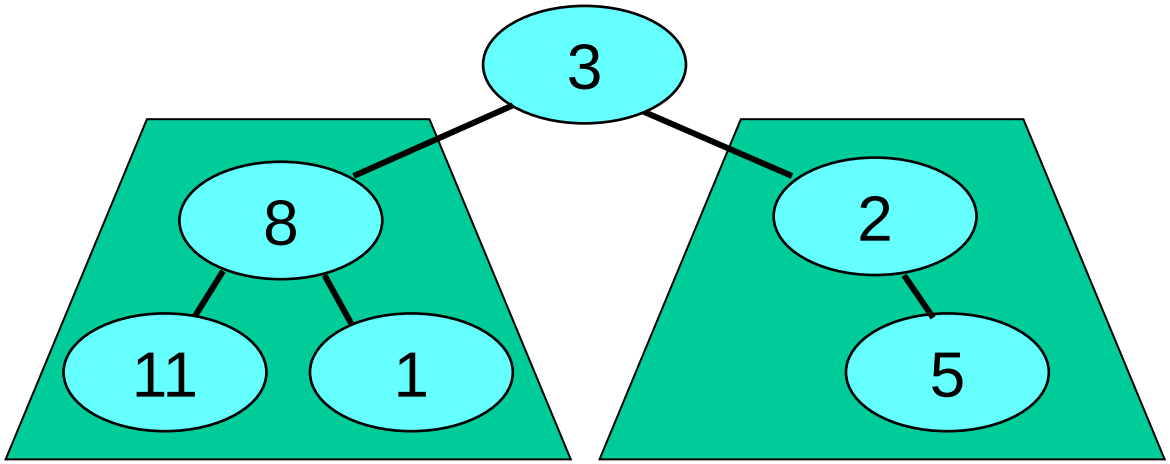
```
postorder: Pesci, Mammiferi, Animali, Graminacee, Betullacee, Vegetali, Esseri v
iventi
```

ALBERI BINARI

- Il caso più tipico di albero è l'albero *binario*
  - VINCOLO: al più due sottoalberi figli, chiamati *sinistro* e *destro*
- Si apre una nuova possibilità di visita
  - oltre all'ordine anticipato (*radice*, figlio1, figlio2, …, figlioN)
  - e all'ordine posticipato (figlio1, figlio2, …, figlioN, *radice*)
  - essendo i figli esattamente due, ha senso anche il caso in cui la radice sia visitata *in mezzo*
- **Approccio ricorsivo: *in ordine*** *(in-order)*
  - prima si richiama ricorsivamente l'operazione sul *figlio sinistro*
  - poi si opera sulla radice
  - infine si richiama ricorsivamente l'operazione sul *figlio destro*

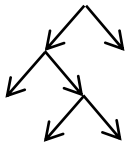
ESEMPIO

- Nel caso dell'albero sotto illustrato:
  - visita in order (sottoalbero sinistro, radice, sottoalbero destro):
    {*sinistro*, **3**, *destro* } = {{sx,8,dx}, **3**, {sx,2,dx}} =
    = {{{11},8,{1}}, **3**, {2,{5}}} = {11, 8, 1, 3, 2, 5}

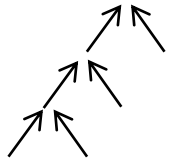


VISITE DI ALBERI BINARI

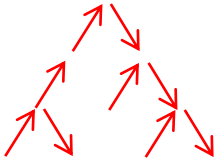
- depth-first, pre-order {3, 8, 11, 1, 2, 5}



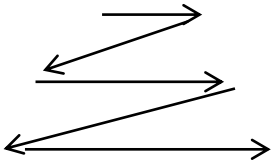
- depth-first, post-order {11, 1, 8, 5, 2, 3}

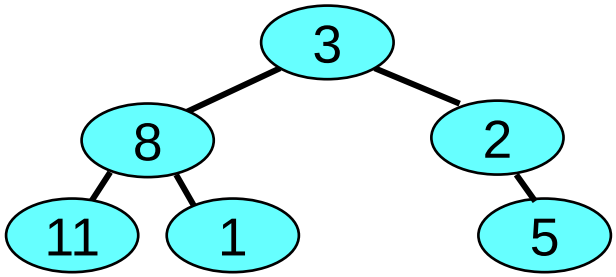


- depth-first, in-order {11, 8, 1, 3, 2, 5}



- breadth-first {3, 8, 2, 11, 1, 5}





VISITA IN ORDER

Specializziamo la visita per il caso di alberi binari:

Java

```
private static <T> void inorder(
   TreeItem<T> root, StringBuilder sb, String separator) {
  if (root != null) {
   List<TreeItem<T>> children = root.getChildren();
   if (children != null && children.size()>2)
       throw new IllegalArgumentException("Not binary!");
   if (children != null && children.size()>0) {
       inorder(children.get(0), sb, separator);
   }
   sb.append(root.getValue()+ separator);
   if (children != null && children.size()>1) {
       inorder(children.get(1), sb, separator);
   }
  }
}
```

```
inorder: Pesci, Animali, Mammiferi, Esseri viventi, Graminacee, Vegetali, Betull
acee,
```

ALBERI & ALGORITMI

La visita è *la base di qualunque algoritmo* su alberi

- qualunque cosa si voglia fare sull'albero, occorre infatti
  - saperla fare su un singolo elemento
  - ripeterla (in un qualche ordine) su tutti gli elementi
- la visita garantisce infatti proprio la *navigabilità*
  - non importa che criterio di visita si scelga (se non per efficienza)

Ergo, per definire un algoritmo occorre

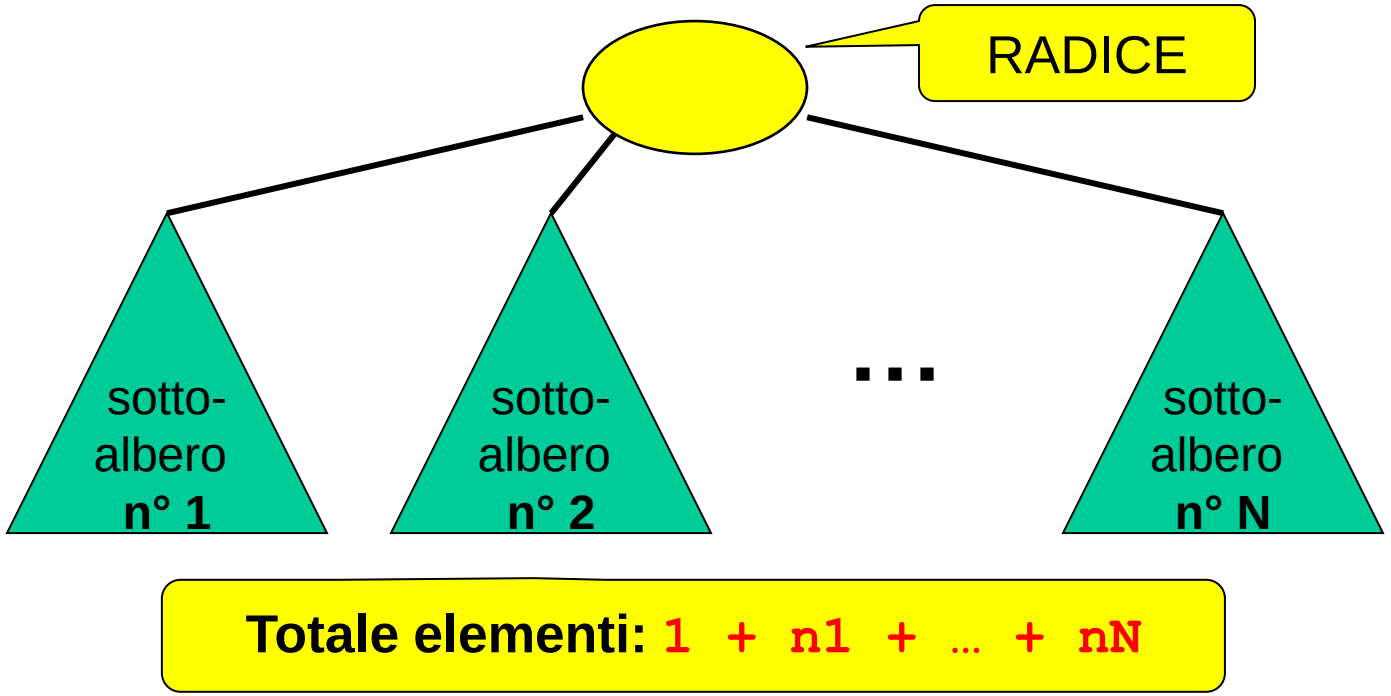
- definire la mossa-base da fare sul singolo elemento
- inserirla in un algoritmo di visita come *operazione sulla radice*

Non è una questione di linguaggio: la ricorsione farà la «magia», propagando la mossa-base su tutti i nodi dell'albero.

ESEMPIO: CONTARE GLI ELEMENTI

Algoritmo: conteggio

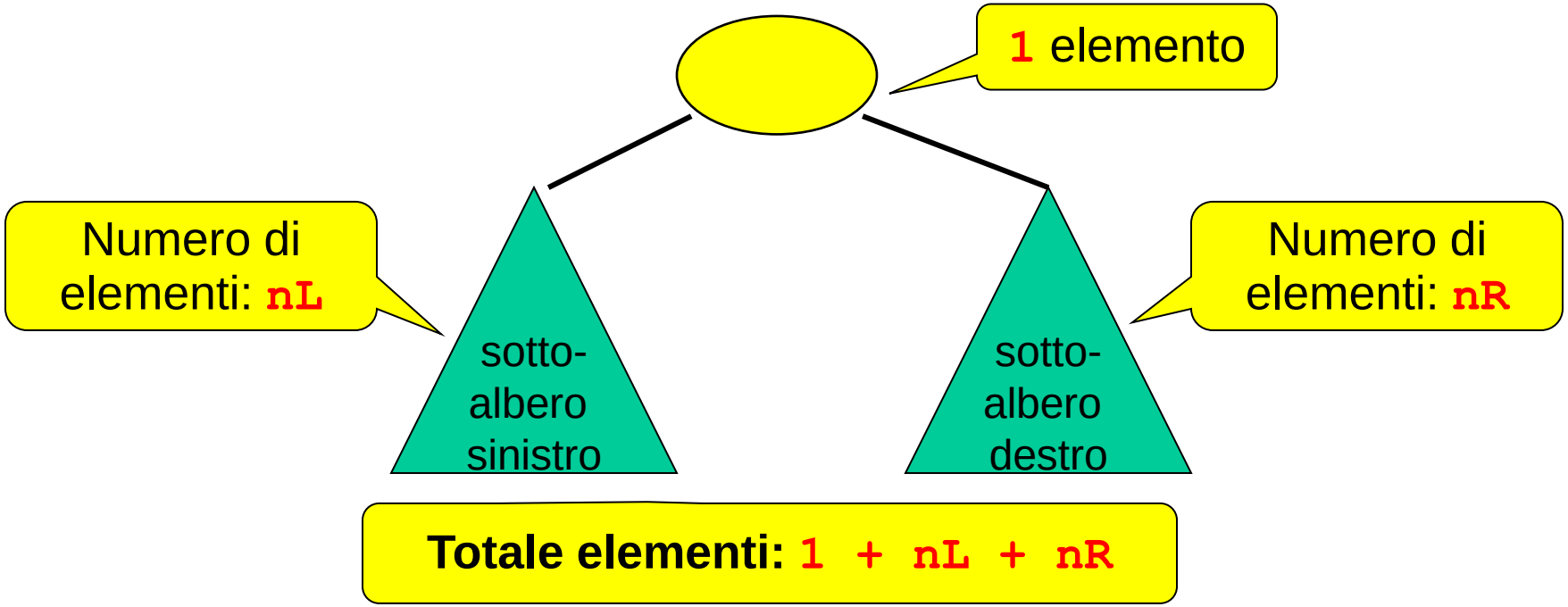
- se l'albero è *vuoto*, gli elementi sono 0
- altrimenti, gli elementi sono 1 (la radice) + quelli dei figli



VERSIONE PER ALBERO BINARIO

Algoritmo: conteggio

- se l'albero è *vuoto*, gli elementi sono 0
- altrimenti, gli elementi sono
  1 (la radice) + quelli del figlio sinistro + quelli del figlio destro



L'ALGORITMO DI CONTEGGIO

Conteggio nodi (versione per albero N-ario)

Java

```
private static <T> long count(TreeItem<T> root) {
  if (root==null) return 0;
  int sum = 1;
  List<TreeItem<T>> children = root.getChildren();
  if (children != null) {
    for (TreeItem<T> child : children) {
          sum += count(child); // chiamata ricorsiva
    }
  }
  return sum;
}
```

Invocazione lato cliente:

```
System.out.println("nodes: " + count(root));
```

```
nodes: 7
```

VARIANTE
CONTARE SOLO "ALCUNI" ELEMENTI

Algoritmo: conteggio dei soli elementi *tali che….*

- se l'albero è *vuoto*, gli elementi sono 0
- altrimenti, gli elementi *interessanti* sono
  1 o 0 (secondo se la radice soddisfa o no il criterio)
  + quelli *interessanti* del figlio sinistro e del figlio destro

Implementazione

- Introdurre una funzione **`filter`** che *valuti la radice* in base al criterio desiderato, restituendo 1 o 0 come necessario
- Possibili criteri:
  - solo gli elementi positivi,
  - solo le stringhe che iniziano per "A",
  - etc.

CONTEGGIO FILTRATO

Conteggio nodi (versione per albero N-ario)

Java

```
private static <T> long countIf(TreeItem<T> root) {
  if (root==null) return 0;
  int sum = filter(root);
  List<TreeItem<T>> children = r
  if (children != null) {
    for (TreeItem<T> child : children) {
          sum += countIf(child); // chiamata ricorsiva
    }
  }
  return sum;
}
```

Incapsula il criterio
"solo i nodi che.."

..ma così il criterio
non si vede!
E' cablato nel codice!

Invocazione lato cliente:

```
System.out.println("nodes: " + countIf(root));
```

CONTEGGIO FILTRATO

IDEA MIGLIORE: passare `filter` come argomento

Java

- una lambda expression ☺
- è una funzione che, dato un **`TreeItem<T>`**, restituisce un **`int`**
- interfacce funzionali utili: **`Function<U,R>`** o **`ToIntFunction<U>`**
- con la prima dovremo usare un **`Integer`** per incapsulare l' **`int`**

```
Function<TreeItem<T>,Integer> filter
```

Attenzione all'uso:

Java

- RICORDA: in Java le lambda non sono "vere funzioni", quindi non si possono invocare direttamente con l'operatore **`()`**
- occorre attivarle invocando il metodo interno **`apply`**

```
int sum = filter.apply(root);
```

(usando **`ToIntFunction`**, il metodo sarà **`applyAsInt`**)

CONTEGGIO FILTRATO

Quale criterio di filtro?

Java

- come esempio, filtriamo *solo i nodi che contengono una 'A'*
  - dato il nodo, estraiamo il valore (una stringa)…
  - ..lo convertiamo in maiuscolo e lo confrontiamo con 'A'..
  - ..e in base a questo decidiamo se restituire 1 o 0
- nel nostro albero di esseri viventi, ciò esclude 2 nodi su 7
  → ci aspettiamo come risultato 5

Invocazione lato cliente:

Java

```
System.out.println("nodes with 'A': " +
  countIf(root,
    (TreeItem<String> n) ->
     n.getValue().toUpperCase().indexOf('A')>=0 ? 1 : 0 ));
);
```

```
nodes with 'A': 5
```

UN ALTRO ESEMPIO: RICERCA ESAUSTIVA DI UN ELEMENTO

Algoritmo: ricerca esaustiva (inefficiente!!)

- se l'albero è *vuoto*, l'elemento sicuramente non c'è
- altrimenti, se coincide con la radice, lo si è trovato
- altrimenti, va cercato ricorsivamente nei sottoalberi figli

Algoritmo: pseudo-codice (per albero binario)

Java

```
public static boolean search(Tree t, Element e){
  if (albero vuoto) return false;
  if (radice.equals(e)) return true;
  return search(figlioSX,e) ? true : search(figlioDX,e);
}
```

Esercizio: adattarlo al nostro albero di **`TreeItem<T>`**

ALTRI ALGORITMI…?

In definitiva:

- tutti gli algoritmi sono una *variazione sul tema* di uno degli algoritmi di visita (in linea di principio, non importa quale)
- ciò che differenzia un algoritmo da un altro è *solo l'operazione da fare sulla radice*

Perché?

- propagando la mossa-base su tutti i sottoalberi, la ricorsione fa via via diventare «radice» (di un qualche sottoalbero) ogni nodo, in un qualche momento
- ergo, la mossa specificata per la radice finisce per essere svolta prima o poi su tutti i nodi dell'albero ☺

VERSO UNA RICERCA EFFICIENTE

- La ricerca esaustiva è drammaticamente inefficiente
- D'altronde, *se non si ha idea di dove sia l'elemento,* l'unica possibilità è *cercarlo ovunque*
  - ma… in effetti, perché costruire l'albero a caso?
- IDEA: strutturare l'albero in modo da *sapere a priori* dove potrebbe essere un certo elemento
  - se la struttura seguisse un qualche criterio furbo, potremmo *evitare di cercare un elemento dove sicuramente non può essere*
- Nozione di *Binary Search Tree (BST)* altrimenti detto *albero binario di ricerca*

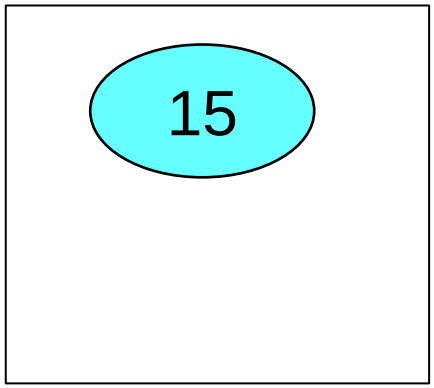
ALBERI BINARI DI RICERCA

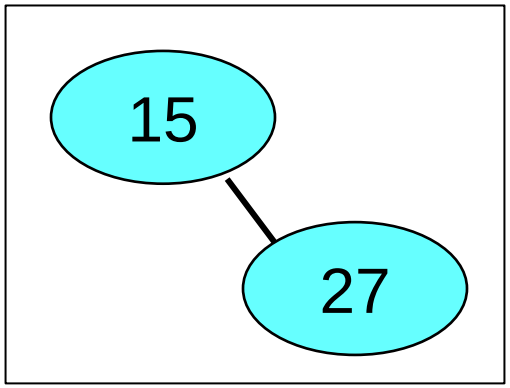
- In un *albero binario di ricerca (Binary Search Tree, BST)*
  - gli elementi rispettano tutti una *relazione d'ordine*
    - deve esistere una relazione d'ordine totale sul dominio
  - il figlio sinistro contiene solo *elementi minori o uguali alla radice*
  - il figlio destro contiene solo *elementi maggiori della radice*
- Così, quando si cerca un elemento *si sa dove cercare*
  - se è minore della radice *si cerca solo a sinistra*
  - se è maggiore della radice *si cerca solo a destra*
  - semplice, ma molto efficace!

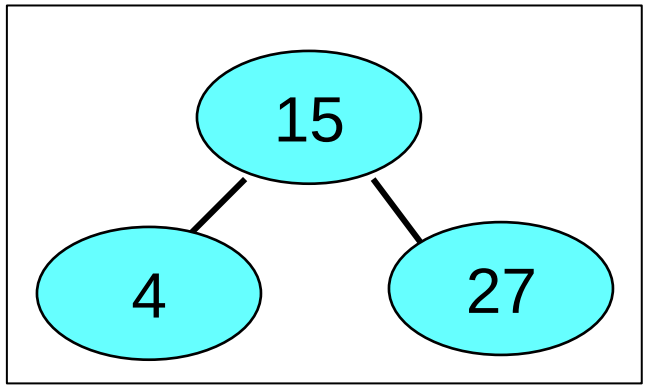
ALBERI BINARI DI RICERCA

- Assunto fondamentale:
  la parte di albero già costruita non si cambia

- Conseguenza:
  gli inserimenti si fanno solo appendendo nuove foglie
  - le ex foglie diventano nodi intermedi
  - per decidere dove appendere la nuova foglia, si parte dalla radice e si devia a sinistra/destra secondo i valori che si incontrano





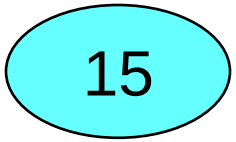


ESEMPIO (1/8)

- L'ordine con cui si inseriscono gli elementi è rilevante
  - inserendo gli stessi elementi in un ordine diverso si ottiene, in generale, un albero diverso.

Esempio A:

1) appare il 15 → radice

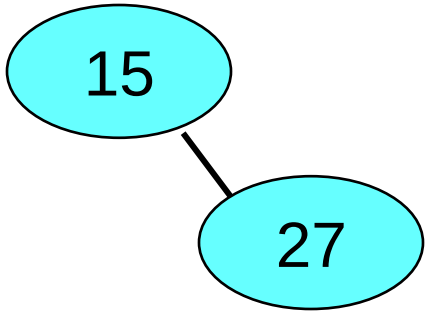


ESEMPIO (2/8)

- L'ordine con cui si inseriscono gli elementi è rilevante
  - inserendo gli stessi elementi in un ordine diverso si ottiene, in generale, un albero diverso.

Esempio A:

1) appare il 15 → radice
2) appare 27 → va a destra

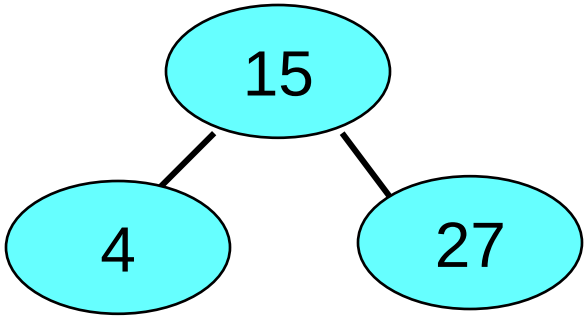


ESEMPIO (3/8)

- L'ordine con cui si inseriscono gli elementi è rilevante
  - inserendo gli stessi elementi in un ordine diverso si ottiene, in generale, un albero diverso.

Esempio A:

1) appare il 15 → radice
2) appare 27 → va a destra
3) appare 4 → va a sinistra

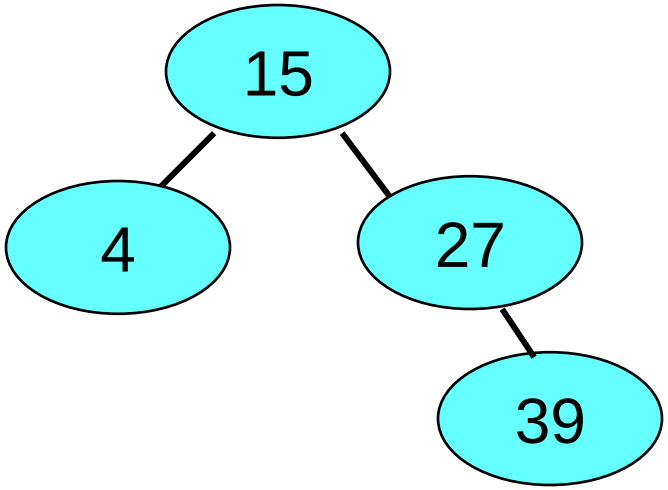


ESEMPIO (4/8)

- L'ordine con cui si inseriscono gli elementi è rilevante
  - inserendo gli stessi elementi in un ordine diverso si ottiene, in generale, un albero diverso.

Esempio A:

1) appare il 15 → radice
2) appare 27 → va a destra
3) appare 4 → va a sinistra
4) appare 39 → in fondo a destra

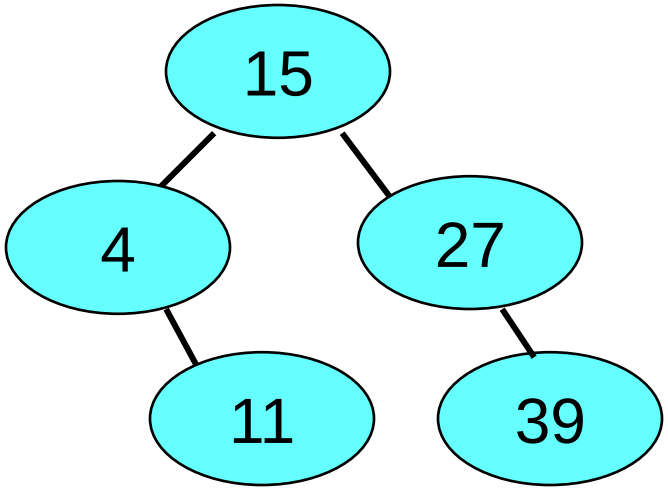


ESEMPIO (5/8)

- L'ordine con cui si inseriscono gli elementi è rilevante
  - inserendo gli stessi elementi in un ordine diverso si ottiene, in generale, un albero diverso.

Esempio A:

1) appare il 15 → radice
2) appare 27 → va a destra
3) appare 4 → va a sinistra
4) appare 39 → in fondo a destra
5) appare 11 → sinistra, poi destra

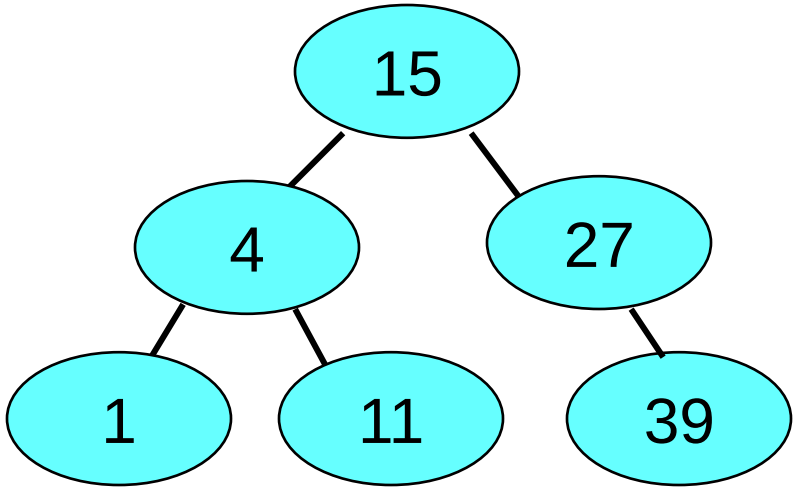


ESEMPIO (6/8)

- L'ordine con cui si inseriscono gli elementi è rilevante
  - inserendo gli stessi elementi in un ordine diverso si ottiene, in generale, un albero diverso.

Esempio A:

1) appare il 15 → radice
2) appare 27 → va a destra
3) appare 4 → va a sinistra
4) appare 39 → in fondo a destra
5) appare 11 → sinistra, poi destra
6) appare 1 → in fondo a sinistra

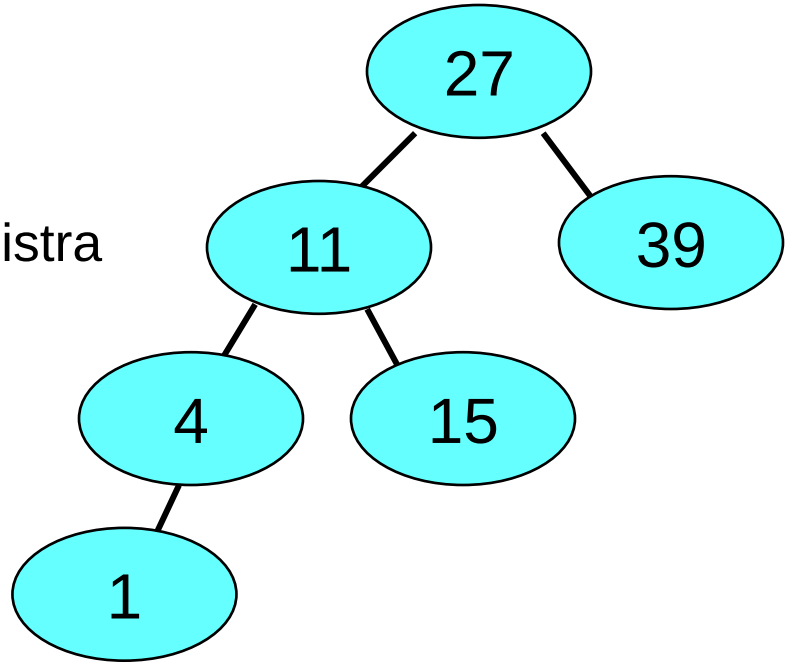


ESEMPIO (7/8)

- L'ordine con cui si inseriscono gli elementi è rilevante
  - inserendo gli stessi elementi in un ordine diverso si ottiene, in generale, un albero diverso.

Esempio B:

1) appare 27 → radice
2) appare 11 → va a sinistra
3) appare 4 → va a sinistra, poi sinistra
4) appare 39 → va a destra
5) appare 1 → in fondo a sinistra
6) appare 15 → sinistra, poi destra

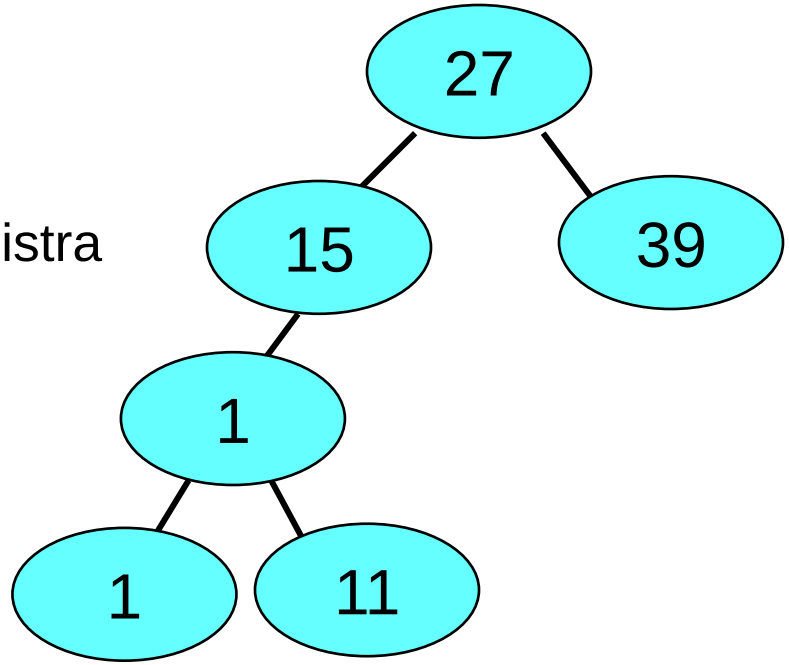


ESEMPIO (8/8)

- L'ordine con cui si inseriscono gli elementi è rilevante
  - inserendo gli stessi elementi in un ordine diverso si ottiene, in generale, un albero diverso.

Esempio C:

1) appare 27 → radice
2) appare 15 → va a sinistra
3) appare 4 → va a sinistra, poi sinistra
4) appare 39 → va a destra
5) appare 1 → in fondo a sinistra
6) appare 11 → sinistra, poi destra



EFFICIENZA NELLA RICERCA

- Un BST velocizza notevolmente la ricerca di un elemento, perché *esclude un pezzo di albero a ogni confronto*
- A ogni confronto, l'esito dice da che parte sta l'elemento:
  - a sinistra, se l'elemento cercato è *minore* della radice
  - a destra, se l'elemento cercato è *maggiore* della radice
- Optimum: albero *bilanciato* → ogni volta dimezza
  - in tal caso, dopo K confronti ha operato K dimezzamenti, esplorando quindi virtualmente uno spazio di $N=2^K$ elementi
- RISULTATO: *ricerca binaria*
  - esplora uno spazio di N elementi con al più $K = \log_2 N$ confronti.

ALBERI BILANCIATI

- Un albero i cui elementi sono distribuiti uniformemente fra i sottoalberi si dice *bilanciato*
- Esistono algoritmi per bilanciare alberi non bilanciati.

Alberi come valori
vs
Alberi come contenitori

ALBERI MODIFICABILI o IMMODIFICABILI?

- Come tutte le strutture dati, un albero può essere inteso (e progettato) come *modificabile* o *immodificabile*
  - nel secondo caso, «modificarlo» significa generare un nuovo albero uguale a quello originale ± la modifica
- Alberi *modificabili* si usano come *contenitori*
  - se ordinati, supportano ricerche efficienti
- Alberi *immodificabili* si usano per *rappresentare relazioni* o più in generale *valori*
  - esempio tipico: espressioni aritmetiche (come 3+4x5)
  - *non ha proprio senso modificarli*, perché facendolo cambierebbe l'espressione rappresentata: l'espressione «è» quell'albero!

ALBERI: VALORI O CONTENITORI?

ALBERI COME CONTENITORI

- Un albero è un particolare tipo di *collezione* di elementi

CONSEGUENZE

- Un albero è un contenitore
- Si può modificare un albero già esistente
- È un approccio efficiente (non ricostruisce ciò che già esiste)
- Ma può essere pericoloso in presenza di *structure sharing*

CASI D'USO

- Frequenti inserimenti di oggetti
- È l'approccio di **`java.util`**

ALBERI COME VALORI

- Un albero è una entità fatta da *un elemento* e da *N (sotto)alberi*

CONSEGUENZE

- Un albero *non è* un contenitore
- *Non* si può modificare un albero già esistente
- È un approccio sicuro (non si modifica mai niente)
- Ma non è sempre efficiente poi-ché opera ricostruendo strutture

CASI D'USO

- Espressioni, interpreti, compilatori
- Strutture (poco/im-)modificabili

ALBERI NELLA JCF

- La JCF adotta l'approccio «*alberi come contenitori*» Java
- Gli alberi JCF sono sempre *ordinati*
  - o gli elementi implementano **`Comparable`**, o si fornisce un **`Comparator`** ad hoc (magari sotto forma di lambda)
  - interfacce **`SortedSet`** e **`SortedMap`**
  - **classi `TreeSet` e `TreeMap`**
  - ricerca binaria garantita con efficienza dell'ordine di log(n)

		Implementations				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

USO: ESEMPIO 1

Java

```
import java.util.TreeSet;

public class Test1 {
  public static void main(String args[]){
      TreeSet<Integer> t1 = new TreeSet<>();
      t1.add(8);
      t1.add(-1);
      t1.add(121);
      t1.add(4);
      System.out.println("Albero: " + t1);
  }
}
```

```
Albero: [-1, 4, 8, 121]
```

ITERATORI SU ALBERI

- A differenza di ciò che ci si potrebbe aspettare…
  - per navigare in un albero, serve una visita ricorsiva
- …i collection framework offrono anche *iteratori* su alberi
  - in tal modo, l'albero diventa navigabile tramite normali cicli
- Come fanno ?
  - idealmente, la visita ricorsiva la fa l'iteratore, che si costruisce poi una lista interna col risultato della visita: **`next()`** usa tale lista
  - in realtà, l'implementazione Java di **`TreeSet`** è più furba perché sfrutta internamente una **`TreeMap`**
  - in tal modo garantisce prestazioni dell'ordine di O(n) per tutte le operazioni fondamentali (la **`TreeMap`** sottostante O(1))

Java

CURIOSI..?
DIAMO UNA SBIRCIATA…

- Sbirciando nel codice della JCF:

Java

```
public class TreeSet<E> extends AbstractSet<E>
                                implements …{
   private transient NavigableMap<E,Object> map;
   private static final Object PRESENT = new Object();
   public TreeSet() {
        this(new TreeMap<E,Object>());
   }
   public boolean add(E e) {
        return map.put(e,PRESENT)==null;
   }
   ...
}
```

Oggetto *dummy* messo nella mappa solo per metterci qualcosa

Il vero scopo è solo *avere una mappa* perché è una struttura efficiente ☺

USO: ESEMPIO 2

Java

```
public class Test2 {
 public static void main(String args[]){
  TreeSet<Integer> t1 = new TreeSet<>();
  t1.add(8);   t1.add(-1);
  t1.add(121); t1.add(4);
  Iterator<Integer> iterator = t1.iterator();
  int cont = 0;
  while (iterator.hasNext()) {
      Integer n = iterator.next();
      cont++;
      System.out.println(cont + ") " + n);
  }
 }
}
```

```
1) -1
2) 4
3) 8
4) 121
```

USO: ESEMPIO 2 – VARIANTE

Java

```
public class Test2v {
 public static void main(String args[]){
  TreeSet<Integer> t1 = new TreeSet<>();
  t1.add(8);   t1.add(-1);
  t1.add(121); t1.add(4);

  int cont = 0;
  for (Integer n : t1) {
      cont++;
      System.out.println(cont + ") " + n);
  }
 }
}
```

L'iteratore è implicito nel costrutto *for each*

```
1) -1
2) 4
3) 8
4) 121
```

USO: ESEMPIO 3

Java

```
public class Test3 {
  public static void main(String args[]){
    TreeSet<Integer> t1 = new TreeSet<>();
    t1.add(8);
    t1.add(-1);
    t1.add(121);
    t1.add(4);

    SortedSet<Integer> t2 = t1.headSet(8);

    for (Integer n: t2) {
        System.out.println(n);
    }
  }
}
```

Il metodo **`headSet`** restituisce il sottoinsieme degli elementi *minori* dell'elemento indicato (senza duplicarli) e *garantisce il mantenimento di tale proprietà nel tempo (v. Esempio 5)*

```
-1
4
```

USO: ESEMPIO 4

Java

```
public class Test4 {
  public static void main(String args[]){
    TreeSet<Integer> t1 = new TreeSet<>();
    t1.add(8);
    t1.add(-1);
    t1.add(121);
    t1.add(4);
    SortedSet<Integer> t2 = t1.headSet(8);
    t2.remove(4);
    System.out.println(
        "Albero: " + t1);
  }
}
```

Attenzione: **`headSet`** *non duplica* gli elementi. Quindi, ogni modifica su t2 si ripercuote su t1.

`remove` restituisce **`true`** se ha trovato e rimosso l'elemento, **`false`** altrimenti.

```
Albero: [-1, 8, 121]
```

E infatti… questo è ora t1!

USO: ESEMPIO 5

Java

```
public class Test5 {
  public static void main(String args[]){
    TreeSet<Integer> t1 = n
    ...
    SortedSet<Integer> t2 = t1.headSet(8);
    t2.add(10); // lancia eccezione!
    // t2 doveva contenere solo elementi < 8
  }
}
```

Il metodo **`headSet`** garantisce il mantenimento della proprietà nel tempo

Violazione di consistenza tentando di aggiungere un elemento maggiore di 8

```
java.lang.IllegalArgumentException: key out of range
```

OLTRE GLI ALBERI-CONTENITORI

- Sebbene possa sembrare contro-intuitivo, possono risultare molto utili anche alberi che *non siano* contenitori
  - per garantire *sicurezza* nella condivisione
  - per rappresentare strutture *immutabili*
  - per rappresentare strutture che *siano intrinsecamente così*
    - caso tipico: espressioni aritmetiche
    - caso tipico: alberi sintattici in interpreti e compilatori
- Come progettarli?
  - «circa» come le liste in C..?
  - ..o magari in un modo più object-oriented..?
  - ..o magari ad hoc per il caso specifico..?

ALBERI COME VALORI: APPROCCIO LIST-LIKE

- Nelle liste
  - un *costruttore* (*cons*) costruisce una nuova lista a partire da
    - un nuovo elemento (la *testa*)
    - una lista già esistente (la *coda*)
  - due *selettori (head* e *tail)* recuperano tali elementi
  - un predicato (*isEmpty*) permette di verificare se la lista è vuota

- Analogamente, negli alberi
  - un *costruttore* (*cons*) costruisce un nuovo albero a partire da
    - un nuovo elemento (la *radice*)
    - due o più alberi già esistenti (i *figli*)
  - tre o più *selettori (root, children)* recuperano tali elementi
  - un predicato (*isEmpty*) permette di verificare se l'albero è vuoto

ALBERI COME VALORI: APPROCCIO LIST-LIKE

- Nel caso degli alberi binari
  - un *costruttore* (*cons*) costruisce un nuovo albero a partire da
    - un nuovo elemento (la *radice*)
    - due sotto-alberi già esistenti (i *figli sinistro e destroi*)
  - tre *selettori (root, left, right)* recuperano tali elementi
  - un predicato (*isEmpty*) permette di verificare se l'albero è vuoto

In C, solitamente:

- Si definisce la **`struct Node`**
- Si definisce **`Tree`** come **`Node*`**
- Il costruttore incapsula **`malloc`**
- I selettori restituiscono *puntatori*
- Il predicato è spesso sostituito dal test diretto **`t==NULL`**

In Java si potrebbe fare come in C:

- Si definisce la classe **`Tree`** con il suo costruttore, i selettori come metodi, il test **`t==null`**
- .. ma il risultato non è un design veramente *"object-oriented"*

Oppure, si può rivedere il design

IL TIPICO APPROCCIO IN C (1)

```
typedef struct node {
  Element root;
  struct node *left, *right;
} *Tree;

Tree consTree(Element e, Tree l, Tree r) {
  Tree t = (Tree) malloc(sizeof(struct node));
  t -> root = e;  t -> left = l;  t -> right = r;
  return t;
}

Element root(Tree t) { return t -> root; }
Tree    left(Tree t) { return t -> left; }
Tree    right(Tree t){ return t -> right; }

/* predicato: per verificare se l'albero t è vuoto */
boolean isEmpty(Tree t){ return t==NULL; }
```

Element è il tipo dell'elemento dell'albero

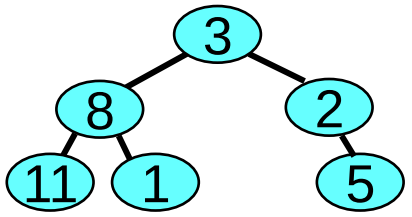
IL TIPICO APPROCCIO IN C (2)

```
void preorder(Tree t, char separator[]) {
  if (!isEmpty(t)) {
    printf("%d%s", root(t), separator);
    if (!isEmpty(left(t))) {
      preorder(left(t), separator);
    }
    if (!isEmpty(right(t))) {
      preorder(right(t), separator);
    }
  }
}
```

Le altre visite sono analoghe

IL TIPICO APPROCCIO IN C (3)

```
int main(){
  Tree foglia1 = consTree(11, NULL, NULL);
  Tree foglia2 = consTree(1, NULL, NULL);
  Tree st1 = consTree(8, foglia1, foglia2);
  Tree foglia3 = consTree(5, NULL, NULL);
  Tree st2 = consTree(2, NULL, foglia3);
  Tree t = consTree(3, st1, st2);
  printf("preorder:  "); preorder(t, ","); printf("\n");
  printf("postorder: "); postorder(t, ","); printf("\n");
  printf("inorder:   "); inorder(t, ","); printf("\n");
}
```




```
preorder:  3,8,11,1,2,5,
postorder: 11,1,8,5,2,3,
inorder:   11,8,1,3,2,5,
```

LO STESSO APPROCCIO IN JAVA

Java

```
public class Tree<T> {
   T root;
   Tree<T> left, right;

   public Tree(T e, Tree<T> left, Tree<T> right) {
      root = e; this.left=left; this.right=right;
   }
   public Tree(T e) {
      root = e;  this.left=null;  this.right=null;
   }
   T         root() { return root; }
   Tree<T>   left() { return left; }
   Tree<T>   right(){ return right; }
```

NB: non si può esprimere il predicato **`isEmpty`** come metodo perché, se l'oggetto corrente esiste, l'albero di cui fa parte *non può mai* essere vuoto. Occorre per forza usare il classico test **`t==null`**

```
}
```

LO STESSO APPROCCIO IN JAVA

Java

```
void preorder(String separator) {
   // dev'essere non-null altrimenti esplode con NullPointerException
   // prima ancora di cominciare!
   System.out.print(this.root() + separator);
   if (this.left()!=null) {
         this.left().preorder(separator);
   }
   if (this.right()!=null) {
         this.right().preorder(separator);
   }
}
```

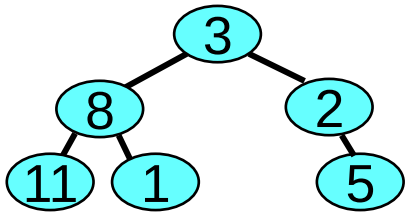
Le altre visite sono analoghe

LO STESSO APPROCCIO IN JAVA

Java

```
public static void main(String[] args){

  Tree<Integer> t =
    new Tree<>(
        3,
        new Tree<>(8, new Tree<>(11), new Tree<>(1)),
        new Tree<>(2, null,           new Tree<>(5)) );
  t.preorder(","); System.out.println();
  t.postorder(","); System.out.println();
  t.inorder(","); System.out.println();
}
```

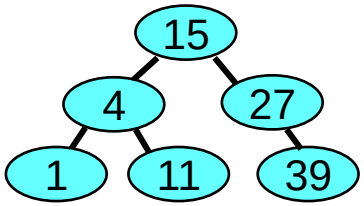



```
3,8,11,1,2,5,
11,1,8,5,2,3,
11,8,1,3,2,5,
```

LO STESSO APPROCCIO IN JAVA: BST

Java

```
public boolean binSearch(Comparable<T> e) {
  if (e.compareTo(root())==0) return true;
  if (left()!=null && e.compareTo(root())<0){
          return left().binSearch(e);
  }
  if (right()!=null){
          return right().binSearch(e);
  }
  return false;
}
```




```
System.out.println(bst.binSearch(4)); // vero!
System.out.println(bst.binSearch(2)); // falso: non c'è..
```

USO & PROBLEMI

- Sembra tutto a posto, ma:
  - il cliente deve sempre controllare che un dato albero non sia vuoto prima di poter agire

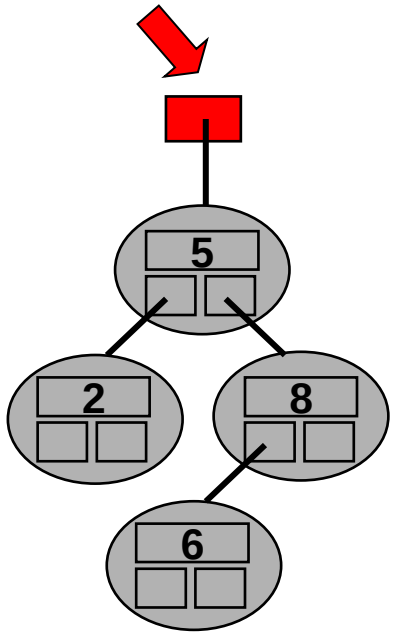
```
if (t.left()!=null) {
  // lavora sul figlio sinistro
}
```

  - non esiste il metodo **`isEmpty`** per impossibilità di definirlo: se l'albero è vuoto, non si possono invocare metodi su di esso!

È la spia si qualcosa di più profondo che non torna…

IL PROBLEMA SOSTANZIALE

- Il punto è che in questo modello *l'albero non esiste!*
  - esiste solo un *riferimento alla radice*
  - si confonde la parte con il tutto: vi par poco?
  - per questo, *albero vuoto* implica *riferimento nullo*
- È *implementato* in un linguaggio a oggetti, ma *non adotta realmente la filosofia OOP*
  - l'albero vuoto non è rappresentabile *perché non esiste alcun oggetto che lo esprima*
  - quindi, non può essere «interrogato» in alcun modo, sotto pena di *NullPointerException*



- È un progetto *"clonato e ispirato dal C", non OOP!*
  - *complica l'uso* dell'astrazione e porta a *codice meno robusto*

UN NUOVO APPROCCIO

- Un buon progetto deve partire *analizzando e modellando bene, in modo chiaro e pulito,* il dominio del problema
  - DOMANDA DI FONDO
    - Cos'è un albero ?
      È un mero «puntatore al (primo) nodo» o… altro?
  - RESPONSABILITÀ:
    - Chi deve verificare le precondizioni delle operazioni?
      Il cliente o l'albero su cui l'operazione è invocata?
    - Cosa succede se un'operazione non può essere fatta?
      Eccezione, restituzione di null, o.. altro?
  - ARCHITETTURA:
    - L'albero è davvero solo una mera aggregazione di nodi?

POSSIBILI SPUNTI

- Punti fermi
  - l'albero è un oggetto, non un «puntatore al primo nodo di qualcosa»
    - anche se non avesse nodi.. dovrebbe esistere comunque
    - se esiste, è sempre interrogabile *senza rischiare NullPointerException*
  - l'albero non è solo un mero «mucchio» di nodi
    - è *composto* di nodi
    - una *collezione* che *gestisce e manipola* elementi e altri sottoalberi
- Possibili approcci
  - l'insieme degli alberi si distingue in *alberi vuoti* e *non vuoti*
    - alberi "qualsiasi" come classe base → `AnyTree`
    - alberi vuoti e non vuoti come sottoclassi concrete → `EmptyTree`, `Tree`
    - gli alberi vuoti hanno risposte "facili" a tutte le domande
  - quale architettura software ..? interfacce? classi astratte? factory..?

Un caso di studio interessante: espressioni aritmetiche

ESPRESSIONI ARITMETICHE

- Tutti noi conosciamo le espressioni aritmetiche con le quattro operazioni standard, e da sempre le scriviamo con una *notazione infissa* basata su operatori: **+**, **-**, *****, **/**

ESEMPI:

```
3+4-5      3+4*5          9-4-1      9-4/2
```

- Essa però richiede le nozioni di *priorità* fra operatori e *associatività* fra operatori equiprioritari:
  - di solito, gli operatori moltiplicativi sono prioritari su quelli additivi e l'associatività fra operatori equiproritari è a sinistra
- Si usano *parentesi* per esprimere priorità e associatività diverse da quelle standard

```
3+4*5      (3+4)*5      9-4-1      9-(4-1)
```

NOTAZIONI PREFISSE E POSTFISSE

- In realtà, la *notazione infissa* è solo *uno* dei modi per rappresentare espressioni, *e neanche il più felice!*
  - È lei che rende necessarie le parentesi!
- Due alternative sono la *notazione prefissa* o *postfissa*
  - NOTAZIONE PREFISSA: prima l'operatore, poi gli operandi
    - tipica notazione funzionale dell'analisi matematica: **f(x,y)**
  - NOTAZIONE POSTFISSA: prima gli operandi, poi l'operatore
    - meno comune, poco usata in matematica: sarebbe **(x,y)f**
- Entrambe comunque non richiedono mai parentesi perché non c'è ambiguità sull'ordine di esecuzione delle operazioni
  - ogni operatore si applica sempre ai due operandi "collegati" a esso

NOTAZIONE PREFISSA: ESEMPI

- Per chiarire, consideriamo qualche espressione:

```
ESEMPIO 1:   3 + 4 * 5
ESEMPIO 2:   (3 + 4)* 5
ESEMPIO 3:   9 – 4 – 1
ESEMPIO 4:   9 – (4 – 1)
```

ventitre

trentacinque

quattro

sei

```
c
```

- In matematica, con una notazione funzionale, scriveremmo:

```
ESEMPIO 1:   add(3,mul(4,5))
ESEMPIO 2:   mul(add(3,4),5)
ESEMPIO 3:   sub(sub(9,4),1)
ESEMPIO 4:   sub(9,sub(4,1))
```

  - OSSERVA: grazie alla composizione di funzioni, che rende evidente «cosa fa fatto prima e cosa dopo», NON sono necessarie regole o parentesi per esprimere priorità e associatività: *è tutto esplicito*

NOTAZIONE PREFISSA: ESEMPI

- Per chiarire, consideriamo qualche espressione:

ESEMPIO 1:	`3 + 4 * 5`
ESEMPIO 2:	`(3 + 4)* 5`
ESEMPIO 3:	`9 – 4 – 1`
ESEMPIO 4:	`9 – (4 – 1)`

Notazione infissa

- Sostituendo semplicemente i nomi con gli usuali simboli:

ESEMPIO 1:	`add(3, mul(4, 5))`	`+ 3 * 4 5`
ESEMPIO 2:	`mul(add(3, 4), 5)`	`* + 3 4 5`
ESEMPIO 3:	`sub(sub(9, 4), 1)`	`– – 9 4 1`
ESEMPIO 4:	`sub(9, sub(4, 1))`	`– 9 – 4 1`

Notazione prefissa

NOTAZIONE POSTFISSA: ESEMPI

- Per chiarire, consideriamo qualche espressione:

ESEMPIO 1:	`3 + 4 * 5`
ESEMPIO 2:	`(3 + 4)* 5`
ESEMPIO 3:	`9 – 4 – 1`
ESEMPIO 4:	`9 – (4 – 1)`

Notazione infissa

- Analogamente si ragiona se l'operatore è *postfisso*:

ESEMPIO 1:	`(3, (4, 5)mul) add`	`3 4 5 * +`
ESEMPIO 2:	`((3, 4)add, 5) mul`	`3 4 + 5 *`
ESEMPIO 3:	`((9, 4)sub, 1) sub`	`9 4 – 1 –`
ESEMPIO 4:	`(9, (4 ,1)sub) sub`	`9 4 1 – –`

  - Questa notazione non è di norma usata in matematica, ma, come vedremo, è invece molto comoda e usata nei sistemi informatici

Notazione postfissa

NOTAZIONI A CONFRONTO

- In definitiva:
  - NOTAZIONE INFISSA: operatore in mezzo ai due operandi (è quella usuale in matematica: x **f** y)
  - NOTAZIONE PREFISSA: prima l'operatore, poi gli operandi (tipica notazione funzionale dell'analisi matematica: **f** x y)
  - NOTAZIONE POSTFISSA: prima gli operandi, poi l'operatore (solitamente non usata in matematica: x y **f**)

INFISSA:	`3 + 4 * 5`	`(3 + 4) * 5`	`9 – 4 – 1`	`9 – (4 – 1)`
PREFISSA:	`+ 3 * 4 5`	`* + 3 4 5`	`– – 9 4 1`	`– 9 – 4 1`
POSTFISSA:	`3 4 5 * +`	`3 4 + 5 *`	`9 4 – 1 –`	`9 4 1 – –`

NOTAZIONI A CONFRONTO

- Priorità? Associatività?
  - La notazione infissa richiede regole di priorità e associatività per stabilire *l'ordine con cui effettuare le operazioni* e quindi parentesi per esprimere eventuali *deroghe* da tale ordine prestabilito.
  - Le notazioni prefissa e postfissa invece NON richiedono tutto ciò, perché l'ordine di esecuzione *è implicito nella struttura*
- Eh..? Ma allora…?
  - La notazione infissa è diffusa solo per ragioni storico-culturali, ma è *molto più complicata* da imparare e usare
  - In una macchina potrebbe essere più utile *cambiare approccio..*

INFISSA:	`3 + 4 * 5`	`(3 + 4) * 5`	`9 – 4 – 1`	`9 – (4 – 1)`
PREFISSA:	`+ 3 * 4 5`	`* + 3 4 5`	`– – 9 4 1`	`– 9 – 4 1`
POSTFISSA:	`3 4 5 * +`	`3 4 + 5 *`	`9 4 – 1 –`	`9 4 1 – –`

PRIORITÀ DEGLI OPERATORI (1)

INFISSA:	`3 + 4 * 5`	`(3 + 4) * 5`	`9 - 4 - 1`	`9 - (4 - 1)`
PREFISSA:	`+ 3 * 4 5`	`* + 3 4 5`	`- - 9 4 1`	`- 9 - 4 1`

L'operatore `+` si applica ai due operandi `3` e `* 4 5`.

La sottoespressione `* 4 5` evidenzia l'operatore `*` applicato agli operandi `4` e `5`.

Secondo le usuali interpretazioni dei simboli, la sottoespressione `* 4 5` denota il valore *venti.*
L'operatore `+` applicato ai valori *tre* e *venti* denota quindi ***ventitre.***

Qui, invece, è l'operatore `*` che si applica agli operandi `+ 3 4` e `5`.

La sottoespressione `+ 3 4` indica ovviamente l'operatore `+` applicato a `3` e `4`.

La sottoespressione `+ 3 4` denota *sette,* ergo l'operatore `*` applicato ai due valori *sette* e *cinque* denota il valore finale ***trentacinque.***

ASSOCIATIVITÀ DEGLI OPERATORI (1)

INFISSA:	`3 + 4 * 5`	`(3 + 4) * 5`	`9 - 4 - 1`	`9 - (4 - 1)`
PREFISSA:	`+ 3 * 4 5`	`* + 3 4 5`	`- - 9 4 1`	`- 9 - 4 1`

Il primo operatore `-` si applica agli operandi `- 9 4` e `1`.

La sottoespressione `- 9 4` evidenzia l'operatore `-` applicato agli operandi `9` e `4`.

Qui, invece, il primo operatore `-` si applica ai due operandi `9` e `- 4 1`.

Quest'ultima sottoespressione indica l'operatore `-` applicato a `4` e `1`.

Secondo le usuali interpretazioni dei simboli, la sottoespressione `- 9 4` denota il valore *cinque.* Perciò, il primo operatore `-`, applicato ai valori *cinque* e *uno,* denota ***quattro***.

Analogamente, la sottoespressione `- 4 1` denota *tre,* perciò il primo operatore `-` applicato ai due valori *nove* e *tre* denota come risultato il valore ***sei***.

PRIORITÀ DEGLI OPERATORI (2)

INFISSA:	`3 + 4 * 5`	`(3 + 4) * 5`	`9 - 4 - 1`	`9 - (4 - 1)`
POSTFISSA:	`3 4 5 * +`	`3 4 + 5 *`	`9 4 - 1 -`	`9 4 1 - -`

Analogamente, ma dualmente rispetto a prima, l'operatore `*` si applica agli operandi `4` e `5`, mentre il successivo operatore `+` si applica agli operandi `3` e `4 5 *`.

Qui il primo operatore `+` si applica agli operandi `3` e `4`, mentre l'operatore `*` si applica ai due operandi `3 4 +` e `5`.

La sottoespressione `4 5 *` denota il valore *venti,* perciò, il successivo operatore `+` applicato ai due valori *tre* e *venti* dà come risultato ***ventitre***.

Poiché la sottoespressione `3 4 +` denota il valore *sette,* il successivo operatore `*` applicato ai due valori *sette* e *cinque* denota ***trentacinque***.

ASSOCIATIVITÀ DEGLI OPERATORI (2)

INFISSA:	`3 + 4 * 5`	`(3 + 4) * 5`	`9 – 4 – 1`	`9 – (4 – 1)`
POSTFISSA:	`3 4 5 * +`	`3 4 + 5 *`	`9 4 – 1 –`	`9 4 1 – –`

Il primo operatore **–** si applica agli operandi **– 9 4**, mentre il secondo operatore **–** si applica ai due operandi **– 9 4** e **1**.

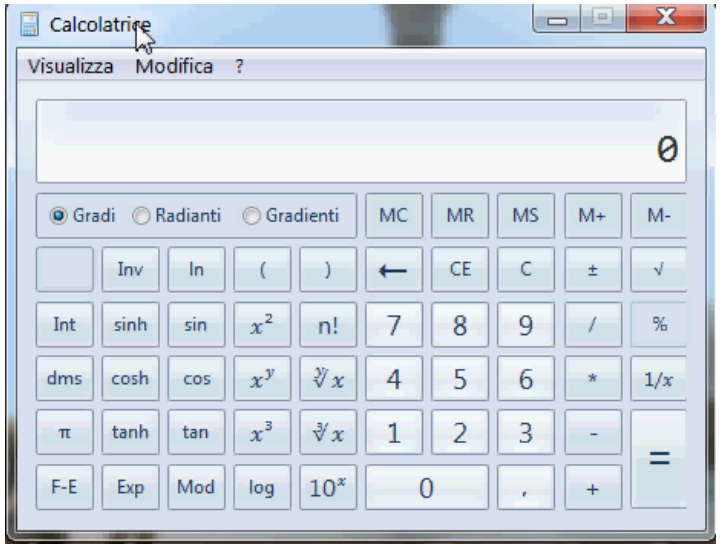
Perciò la sottoespressione **– 9 4**, secondo le usuali interpretazioni dei simboli, denota *cinque.* Il successivo operatore **–** applicato ai due valori *cinque* e *uno* denota perciò ***quattro.***

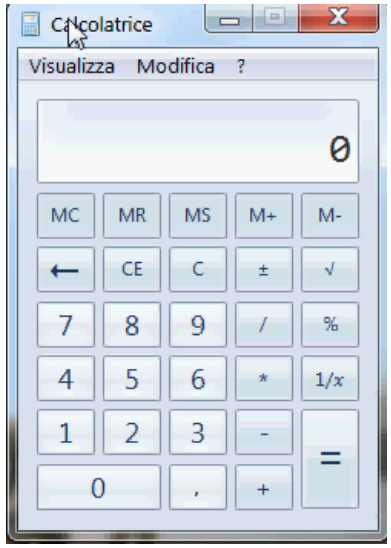
Qui, invece, il primo operatore **–** si applica agli operandi **4** e **1**, mentre il secondo operatore **–** si applica agli operandi **9** e **4 1 –**.

La sottoespressione **4 1 –** denota *tre,* ergo il successivo operatore **–** applicato ai valori *nove* e *tre* denota ***sei.***

ESPERIMENTO

"`3+4*5` denota *ventitré*, *non trentacinque*"





Utilizzare la modalità Scientifica

1. Scegliere **Scientifica** dal menu **Visualizza**.
2. Fare clic sui tasti della calcolatrice per eseguire i calcoli necessari.

   Per accedere alle funzioni inverse, fare clic su **Inv**.

Note

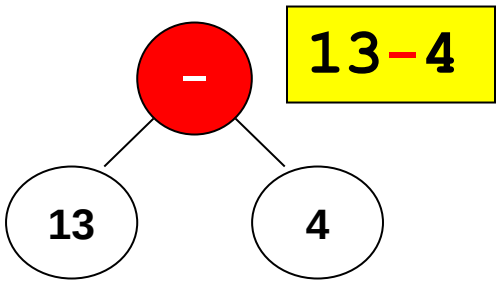
- In modalità Scientifica la Calcolatrice è precisa fino a 32 cifre significative.
- Durante i calcoli in modalità scientifica la Calcolatrice rispetta la precedenza degli operatori.

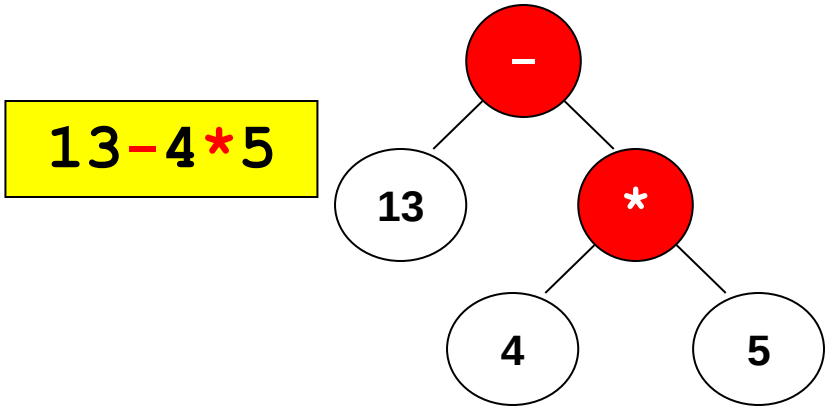
WHICH FOR WHAT?

- Le notazioni *prefissa* e *postfissa* sono importanti appunto perché *non c'è mai ambiguità sull'ordine di esecuzione* delle operazioni e quindi *non richiedono parentesi*
- In particolare, la notazione *postfissa* è adattissima a un elaboratore, che ha necessità di disporre dei dati *prima* di svolgere le operazioni nell'ALU
  - guarda caso, la notazione *postfissa* specifica proprio *prima* gli operandi e *solo dopo* "comanda" l'esecuzione dell'operazione
  - la notazione *infissa* è adatta agli umani che operano con «carta e penna», ma non a una macchina

ESPRESSIONI E ALBERI

- Gli alberi sono *il modo più efficace e universalmente usato di rappresentare espressioni (e programmi..)* in un elaboratore
- Ogni operatore è un nodo con due figli:
  - il figlio *sinistro* è il *primo operando*
  - il figlio *destro* è il *secondo operando*
- I valori numerici sono le foglie.

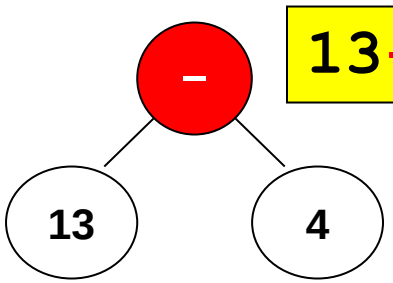


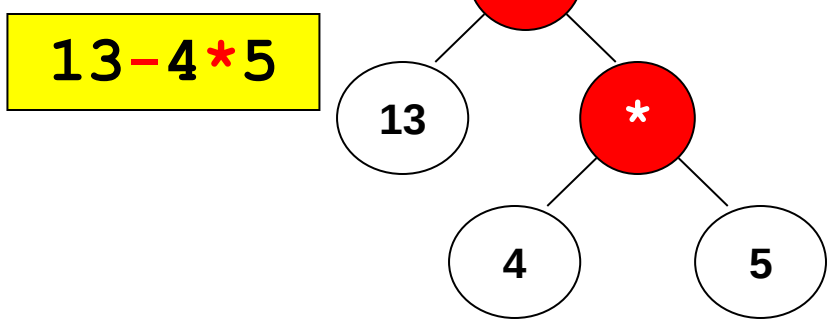


ESPRESSIONI E ALBERI

- Gli alberi sono *il modo più efficace e universalmente usato di rap* ratore
- Og
  - –
  - –
- I v

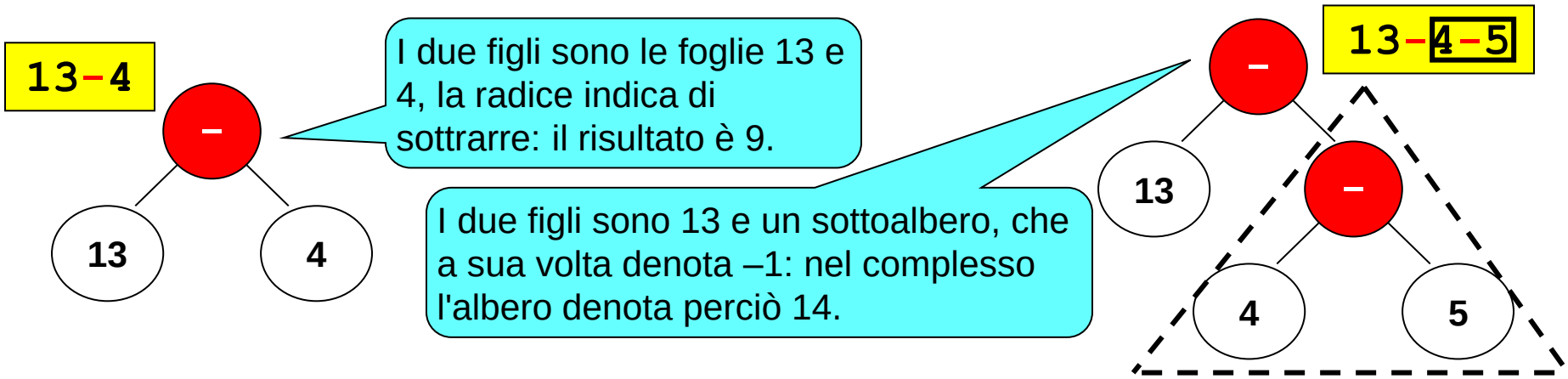
- Sono alberi come valori, perché non ha senso cambiare un nodo: cambierebbe l'espressione rappresentata!
- Inoltre, poiché la struttura dell'albero rappresenta l'espressione, *non ha senso parlare di "alberi ordinati":* l'albero è semplicemente *come deve essere* per rappresentare correttamente quell'espressione.





ESPRESSIONI E ALBERI

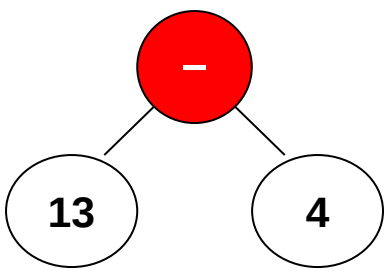
- La rappresentazione ottenuta è univoca: ogni espressione è rappresentata in *un solo modo, senza ambiguità*
- Inoltre, la *struttura* dell'albero fornisce intrinsecamente *l'ordine di valutazione* delle operazioni
  - è impossibile valutare un nodo senza disporre prima dei due figli!
  - quindi, occorre *per forza* valutare prima la parte *«in basso»*
  - alla fine, la valutazione del nodo radice fornisce il risultato



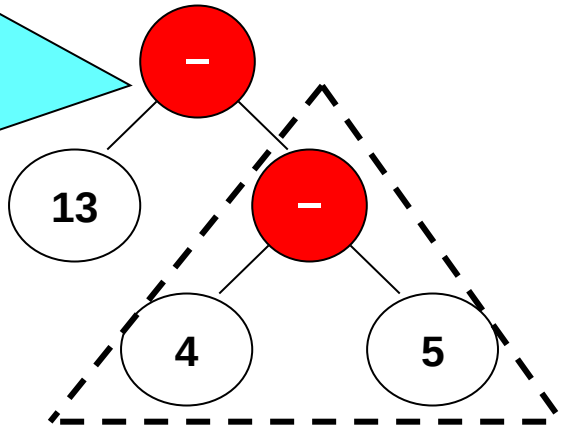
VISITE DI ALBERI-ESPRESSIONE

- Visitando l'albero-espressione nei diversi modi, si ottengono:
  - la notazione *prefissa* con la visita *pre-order*
  - la notazione *infissa* con la visita *in-order*
  - la notazione *postfissa* con la visita *post-order*

PREFISSA: **- 13 4**
INFISSA: **13 - 4**
POSTFISSA: **13 4 -**



PREFISSA: **- 13 R**
INFISSA: **13 - R**
POSTFISSA: **13 R -**
dove R è il sottoalbero**:**
PREFISSA(R): **- 4 5**
INFISSA(R): **4 - 5**
POSTFISSA(R): **4 5 -**



VISITE DI ALBERI-ESPRESSIONE

- Visitando l'albero-espressione nei diversi modi, si ottengono:
  - la notazione *prefissa* con la visita *pre-order*
  - la notazione *infissa*
  - la notazione *postfissa*

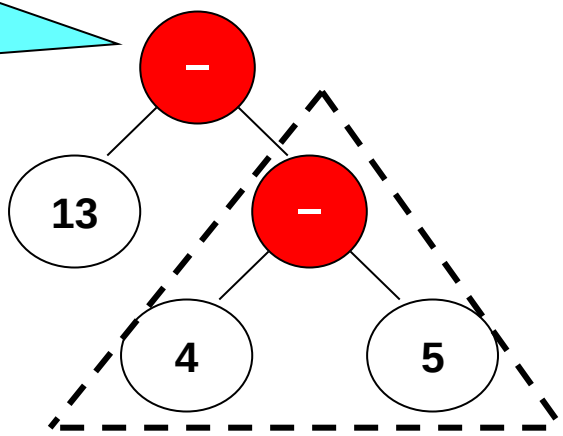
ATTENZIONE ALLA NOTAZIONE INFISSA!

Questa espressione dovrebbe essere scritta come `13-(4-5)`, perché secondo le nostre usuali convenzioni `13-4-5` è un'altra cosa!!

PREFISSA: `- 13 - 4 5`

INFISSA: `13 - 4 - 5`

POSTFISSA: `13 4 5 - -`



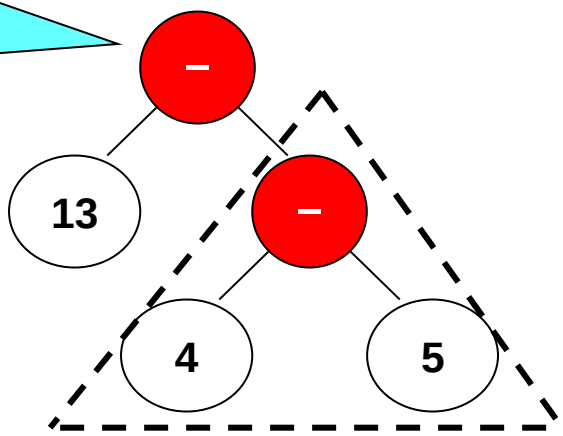
ATTENZIONE ALLA NOTAZIONE INFISSA!

Non evidenziando il "livello" (tramite parentesi, colori, indentazioni, font, o altri artifici), l'algoritmo può dar luogo a frasi che, *secondo le usuali convenzioni, indicano un'espressione diversa!*

VISITE DI ALBERI-ESPRESSIONE

- Visitando l'albero-espressione nei diversi modi, si ottengono:
  - la notazione *prefissa* con la visita *pre-order*
  - la notazione *infissa* con la visita *in-order*
  - la notazione *postfissa* con la visita *post-order*

PREFISSA: **- 13 - 4 5**
INFISSA: **13 - 4 - 5**
POSTFISSA: **13 4 5 - -**



Occorre modificare l'algoritmo di visita in-order in modo che *introduca un'indicazione del livello visitato*, evitando così di "appiattire" l'albero e perdere informazione → *parentesi* o altri artifici (font, colori..)
[es. font `13-4-5` , `13-4-5`, `13-4-5`,...]

NOTAZIONE POSTFISSA.. e oltre

- La notazione *postfissa è adattissima a un elaboratore* perché fornisce *prima* gli operandi
  - che possono così essere caricati nei registri del processore o in altre zone opportune di memoria, pronti per l'ALU

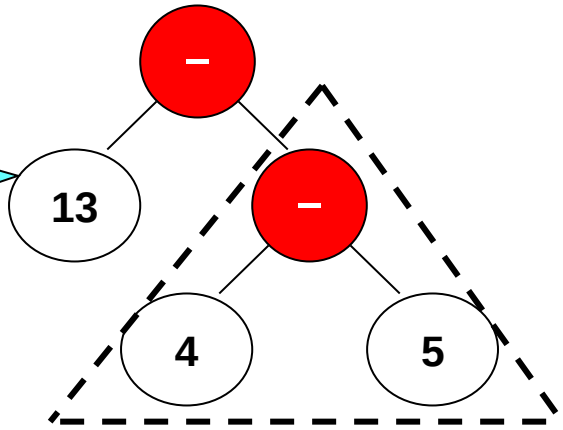
  e *solo dopo* "comanda" l'esecuzione dell'operazione.
- Lo stesso principio è utilizzato anche nei compilatori:
  - ogni nodo-operatore viene tradotto nell'operazione assembler
  - ogni nodo-valore viene tradotto nel caricamento di tale valore in un registro macchina

POSTFISSA:

```
13 4 5 - -
```

CODICE MACCHINA:

```
load 13, r1
load  4, r2
load  5, r3
sub  r2, r3
sub  r1, r2
```



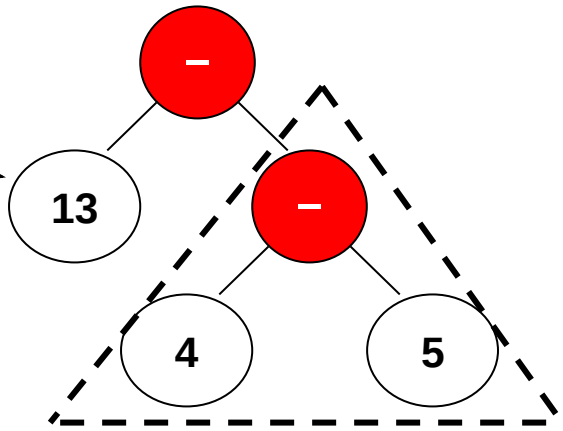
LA MACCHINA A STACK

- E se i registri non bastano?
- E se non si vuole preoccuparsi di quali registri usare?
- Si può usare una macchina a stack!
  - ogni nodo-valore carica un valore sullo stack (PUSH)
  - ogni nodo-operatore causa il prelievo di due valori dallo stack (POP) e il collocamento sullo stack del risultato (PUSH)
  - alla fine basta prelevare il risultato dallo stack (POP)

POSTFISSA: `13 4 5 - -`

CODICE MACCHINA:

```
push 13
push  4
push  5
sub  [include 2 pop+1push]
sub  [include 2 pop+1push]
[segue pop finale]
```

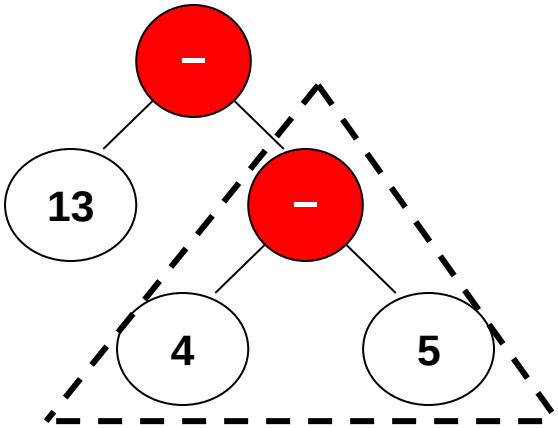


LA MACCHINA A STACK: ESEMPIO

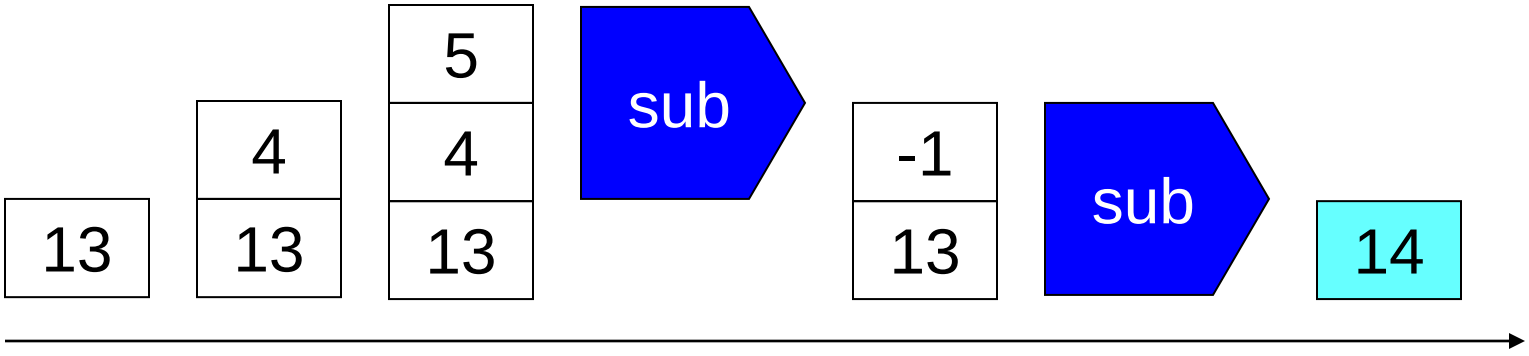
POSTFISSA: `13 4 5 - -`

CODICE MACCHINA:

```
push 13
push  4
push  5
sub  [include 2 pop+1push]
sub  [include 2 pop+1push]
[segue pop finale]
```



Evoluzione dello stack nel tempo:



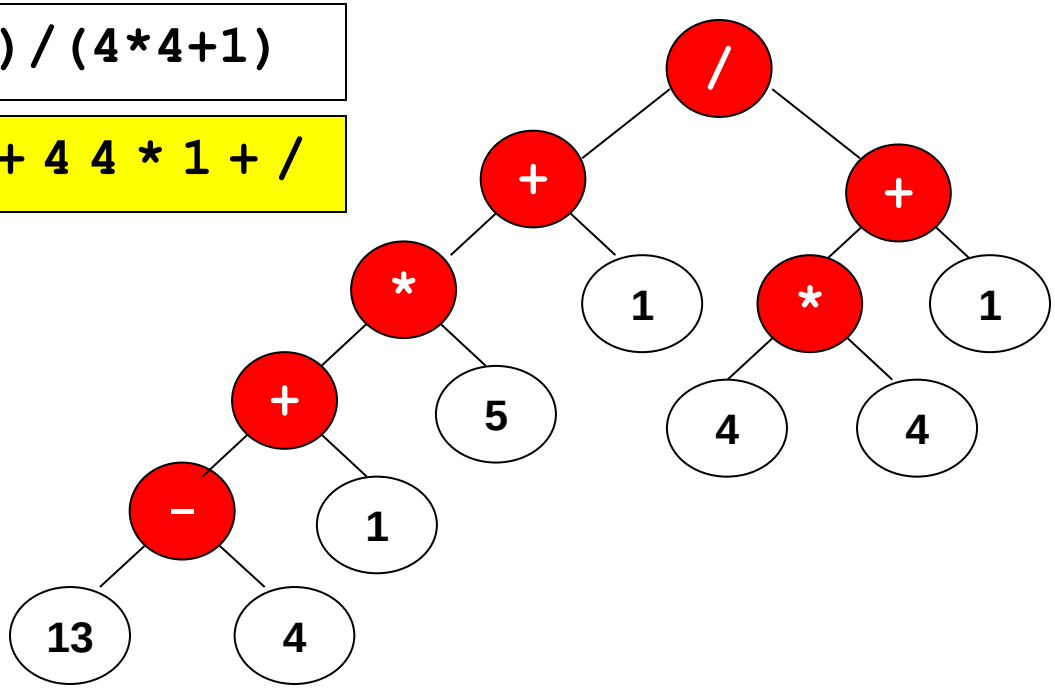
UN ALTRO ESEMPIO

INFISSA: `(((13-4)+1)*5)+1)/(4*4+1)`

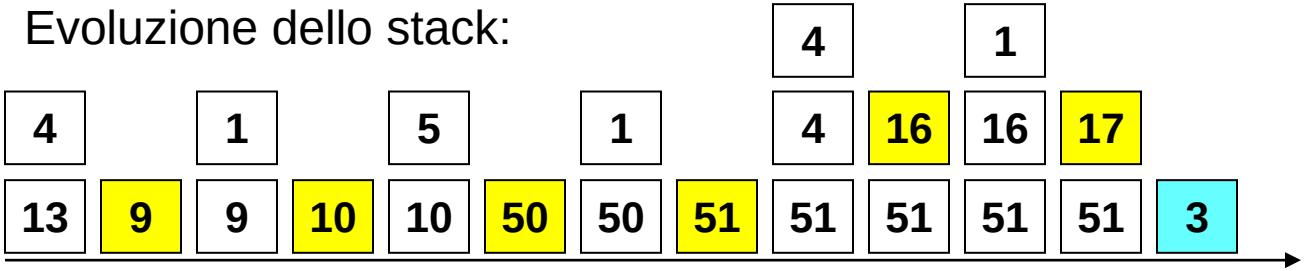
POSTFISSA: `13 4 - 1 + 5 * 1 + 4 4 * 1 + /`

CODICE MACCHINA:

```
push 13
push  4
sub
push  1
sum
push  5
mul
push  1
sum
push  4
push  4
mul
push  1
sum
div
[pop finale]
```



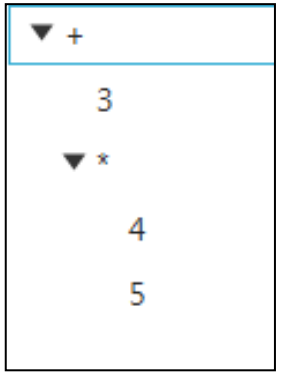
Evoluzione dello stack:



L'ESEMPIO CON `TreeItem<T>`

- Esempio: un albero per l'espressione 3+4*5

```
TreeItem<String> root = new TreeItem<>("+");
TreeItem<String> l = new TreeItem<>("3");
TreeItem<String> r = new TreeItem<>("*");
TreeItem<String> rl = new TreeItem<>("4");
TreeItem<String> rr = new TreeItem<>("5");
r.getChildren().add(rl);r.getChildren().add(rr);
root.getChildren().add(l); root.getChildren().add(r);
TreeView<String> treeview = new TreeView<>(root);
```



- Eseguendo le tre visite standard si ottiene:

```
preorder: +, 3, ×, 4, 5
postorder: 3, 4, 5, ×, +
inorder: 3, +, 4, ×, 5,
```

OCCHIO: in-order non esprime il livello (associatività) degli operatori

VALUTAZIONE DELL'ALBERO

- Per *valutare* un'espressione servono:
  - la visita in postorder
    - ce l'abbiamo già, ma produce una stringa: ci serve invece che accumuli il risultato in una struttura dati (ad es. una lista)
    - nuova funzione **`postorderEnumeration(root,list)`** che riceve una **`List<TreeNode<String>>`** da riempire
  - uno stack di appoggio → c'è nella JCF
    - **`Stack<Integer> stack = new Stack<Integer>();`**
- Ora basta *scandire la lista*:
  - quando si incontra un numero, lo si mette sullo stack → **`push`**
  - quando si incontra un operatore, si svolge l'operazione:
    - prendendo i due dati dallo stack → due **`pop`**
    - mettendo sullo stack il risultato → **`push`**

PRIMA IMPLEMENTAZIONE (orrenda)

Java

```
private static Integer calc(TreeItem<String> root) {
   Stack<Integer> stack = new Stack<Integer>();
   List<TreeItem<String>> list = new ArrayList<>();
   postorderEnumeration(root, list);
   for (TreeItem<String> item : list) {
      String value = item.getValue();
      try {
         Integer i = Integer.parseInt(value);
         stack.push(i);
      }
      catch (NumberFormatException e)  {
         // tutto ok, se non è un intero, value è l'operatore
         // qui va inserita la logica dell'operazione
      }
   }
   return stack.pop();
}
```

Non si può sapere se è un numero o un operatore → si tenta e si vede come va a finire (ARGH…!!)

La business logic nel catch?!? ORRENDO oltre ogni limite!!

PRIMA IMPLEMENTAZIONE (orrenda)

- La logica dell'operazione:

Java

```
catch (NumberFormatException e)  {
// tutto ok, se non è un intero, value è l'operatore
   Integer v2 = stack.pop();
   Integer v1 = stack.pop();
   switch (value) {
      case "+": stack.push(v1 + v2); break;
      case "-": stack.push(v1 - v2); break;
      case "*": stack.push(v1 * v2); break;
      case "/": stack.push(v1 / v2); break;
   }
}
```

La business logic nel catch?!?
ORRENDO oltre ogni limite!!

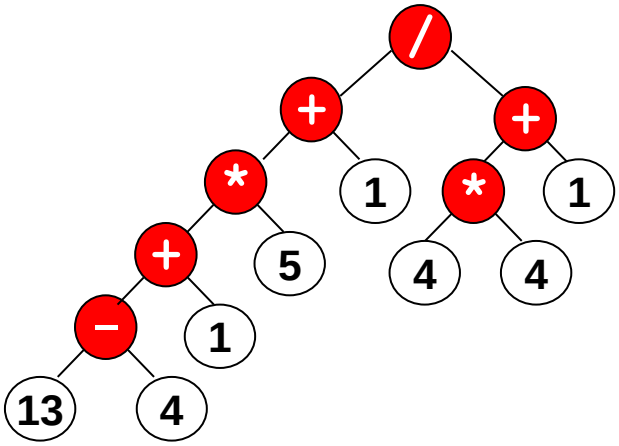
Potremmo usare anche simboli di operazioni diversi, ad esempio ":" per la divisione!

```
Il risultato di  3 4 5 * +  è  23
```

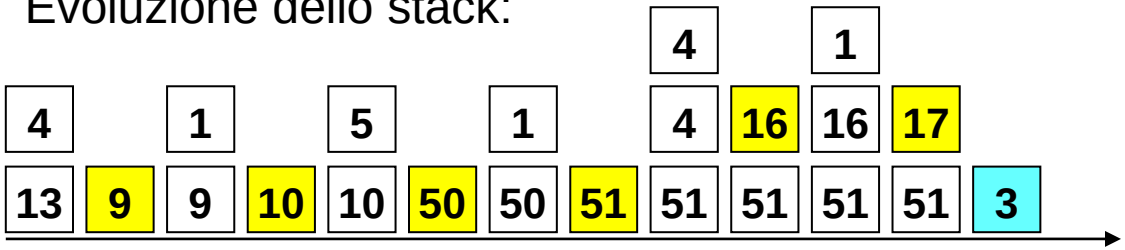
VALUTAZIONE del Tree con Stack

INFISSA: `(((13-4)+1)*5)+1)/(4*4+1)`

POSTFISSA: `13 4 - 1 + 5 * 1 + 4 4 * 1 + /`



Evoluzione dello stack:

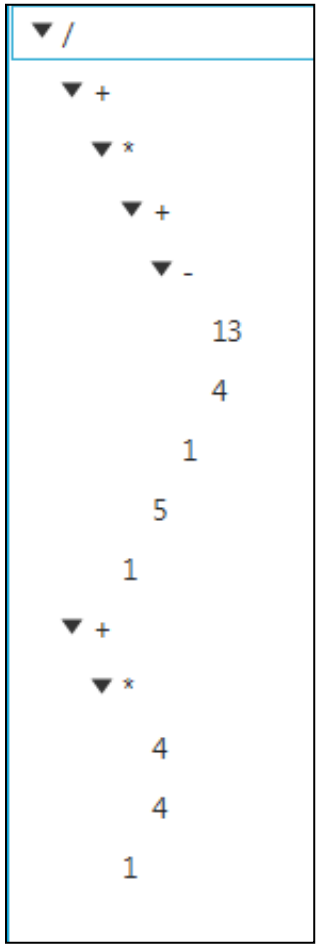



```
preorder: /, +, *, +, -, 13, 4, 1, 5, 1, +, *, 4, 4, 1
postorder: 13, 4, -, 1, +, 5, *, 1, +, 4, 4, *, 1, +, /
inorder: 13, -, 4, +, 1, *, 5, +, 1, /, 4, *, 4, +, 1,
espressione: (((((13)-(4))+(1))*(5))+(1))/(((4)*(4))+(1))
```

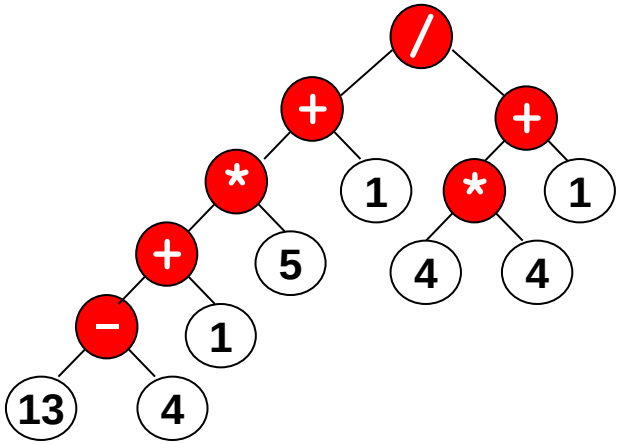
```
Il risultato di  13 4 - 1 + 5 * 1 + 4 4 * 1 + /  è  3
```

LA CORRISPONDENTE `TreeView`

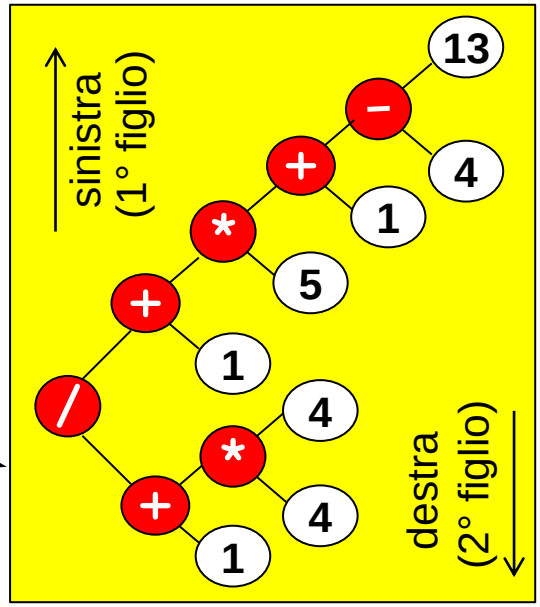
POSTFISSA: **`13 4 - 1 + 5 * 1 + 4 4 * 1 + /`**



OSSERVA:
il layout sembra "opposto", perché si espande verso il basso e verso destra



Capito ciò, basta ruotarlo di 90° per riconoscerlo:



IMPLEMENTAZIONE ALTERNATIVA

Java

```
private static Integer calc(TreeItem<String> root) {
   Stack<Integer> stack = new Stack<Integer>();
   List<TreeItem<String>> list = new ArrayList<>();
   postorderEnumeration(root, list);
   for (TreeItem<String> item : list) {
      String value = item.getValue();
      if (value.matches("[0-9]")) {
            Integer i = Integer.parseInt(value);
            stack.push(i);
      }
      else {
         // se non è un intero, dev'essere l'operatore
         // qui va inserita la logica dell'operazione
      }
   }
   return stack.pop();
}
```

Sfrutta le regex di Java per verificare se è un int

IMPLEMENTAZIONE ALTERNATIVA

- La logica dell'operazione:

Java

```
else {
   // logica dell'operazione
   Integer v2 = stack.pop();
   Integer v1 = stack.pop();
   switch (value) {
      case "+": stack.push(v1 + v2); break;
      case "-": stack.push(v1 - v2); break;
      case "*": stack.push(v1 * v2); break;
      case ":": stack.push(v1 / v2); break;
   }
}
```

```
Il risultato di  3 4 5 * +  è  23
```

```
Il risultato di  3 4 + 5 *  è  35
```