

STRUTTURE DATI: OLTRE LE LISTE

- Essendo in grado di ospitare una *collezione di elementi* il cui numero può variare dinamicamente, le liste risolvono un ampio ventaglio di problemi
- Le liste sono strutture dati *sequenziali*, e poco adatte a gestire *grandi quantità di dati* quando le operazioni più frequenti / importanti implicano un accesso *non sequenziale*

Alberi 1

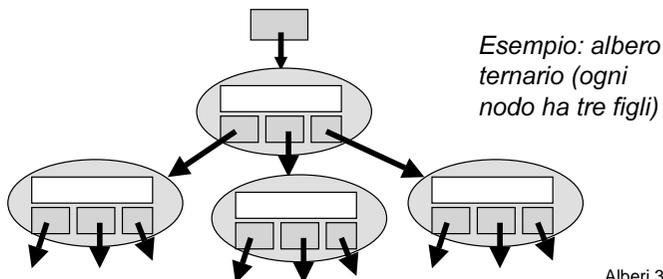
STRUTTURE DATI: OLTRE LE LISTE

- Ad esempio, in una lista, la *ricerca di un elemento* richiede un tempo proporzionale alla lunghezza della lista
 - se la lista contiene centinaia di migliaia di elementi, tale operazione è molto inefficiente
- La gestione di grandi quantità di dati può quindi avvantaggiarsi dall'adozione di strutture dati *non sequenziali*, che rendano più veloci (potenzialmente) alcune categorie di operazioni, in particolare la *ricerca*

Alberi 2

ALBERI

- Un *albero* è una *collezione di elementi* organizzati in modo *non sequenziale*, secondo un *grafo aciclico* caratterizzato da
 - una *radice*
 - *due o più (sotto)alberi figli*

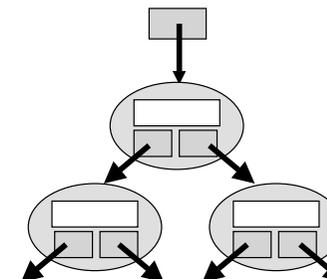


Alberi 3

ALBERI BINARI

Il caso più tipico di albero è l'albero *binario*

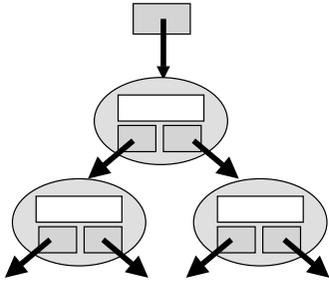
- un elemento
- due (sotto)alberi figli, *sinistro* e *destro*



Alberi 4

ALBERI BINARI

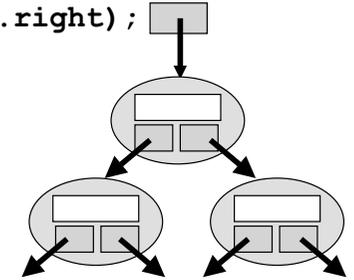
```
class TreeNode {
    int item; // informazione del nodo
    TreeNode left; // figlio sinistro
    TreeNode right; // figlio destro
}
```



Alberi 5

CONTA NODI

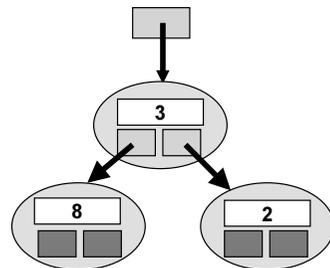
```
static int countNodes(TreeNode root) {
    if (root == null) return 0;
    else
    {int count = 1;
     count += countNodes(root.left);
     count += countNodes(root.right);
     return count; }
} // end countNodes()
```



Alberi 6

ALBERI: RAPPRESENTAZIONE

- Un *albero* è praticamente sempre rappresentato tramite nodi concatenati
- Ogni nodo è composto di *tre parti*: un *valore* e due *riferimenti ai figli* (i sottoalberi sinistro e destro)
- La *rappresentazione fisica* è solitamente basata su puntatori e allocazione dinamica



Alberi 7

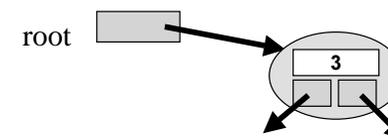
ALBERI: RAPPRESENTAZIONE

- In questo contesto, l'*albero vuoto* è rappresentata dall' assenza di nodi

root null

– un puntatore (o riferimento) nullo

- Quando ci sono elementi, l'*albero non è vuoto* e consente l'accesso ai dati



Alberi 8

VISITA DI UN ALBERO

- Con il termine *visita* si intende *l'elencazione*, secondo un qualche criterio, *di tutti gli elementi della struttura*, senza ripetizioni
- Data la natura *intrinsecamente non sequenziale* dell'albero, non esiste un concetto di "sequenza naturale" degli elementi (come esisteva invece per la lista)
- Occorre definire *uno o più criteri di visita* che assicurino comunque di elencare, ciascuno una sola volta, tutti gli elementi

Alberi 9

VISITA DI UN ALBERO

Dato che un albero è definito come una struttura caratterizzata da

- un elemento
- N (sotto)alberi

vi sono due criteri naturali di visita:

- *visita in ordine anticipato (preorder)*: prima la radice, poi tutti i sottoalberi
- *visita in ordine posticipato (postorder)*: prima tutti i sottoalberi, poi la radice

Alberi 10

VISITA DI UN ALBERO BINARIO

Nel caso particolare di un albero binario, che ha due sottoalberi, è possibile definire un ulteriore criterio "naturale" di visita:

- *visita in ordine (inorder)*: prima il sottoalbero di sinistra, poi la radice, poi il sottoalbero di destra

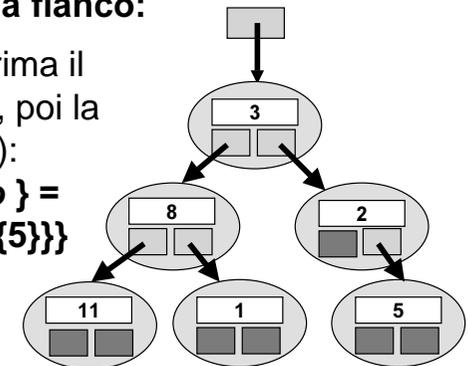
Questo criterio *ha senso solo per alberi binari*, in quanto per un albero generico con N figli esisterebbero N possibili visite "in ordine", nessuna più significativa delle altre

Alberi 11

VISITA in ORDINE

Nel caso dell'albero a fianco:

- *visita in ordine* (prima il sottoalbero sinistro, poi la radice, poi il destro):
 $\{\text{sinistro}, 3, \text{destro}\} = \{\{\{11\}, 8, \{1\}\}, 3, \{2, \{5\}\}\}$



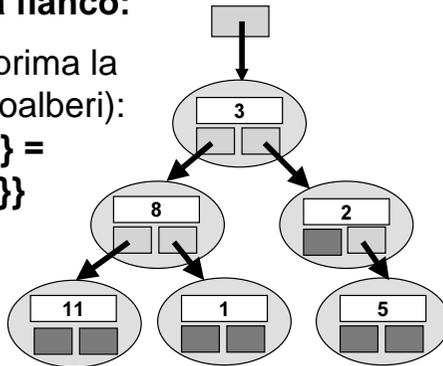
Alberi 12

ESEMPIO

Nel caso dell'albero a fianco:

- **visita in preorder** (prima la radice, poi tutti i sottoalberi):
 $\{3, \text{sinistro}, \text{destro}\} = \{3, \{8, \{11, 1\}\}, \{2, \{5\}\}\}$

- **visita in postorder** (prima tutti i sottoalberi, poi la radice):
 $\{\text{sinistro}, \text{destro}, 3\} = \{\{11, 1\}, 8, \{\{5\}, 2\}, 3\}$



Alberi 13

VISITA IN AMPIEZZA

Un terzo criterio sfrutta la struttura dell'albero, che definisce naturalmente dei *livelli*

- livello 0: la radice
- livello 1: le radici dei figli
- livello 2: le radici dei figli dei figli ...

- **visita in ampiezza (breadth-first)**: prima la radice (livello 0), poi tutti i nodi del I° livello, poi tutti quelli del II° livello, ecc.

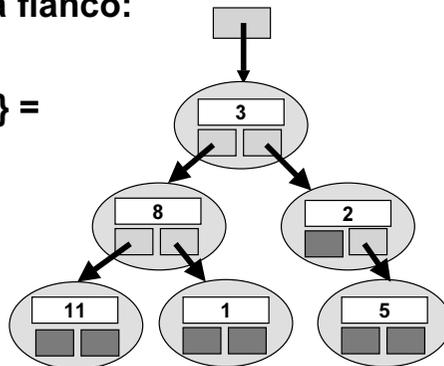
Questo criterio è più complesso da realizzare, perché deve seguire tutte le diverse ramificazioni

Alberi 14

ESEMPIO

Nel caso dell'albero a fianco:

- **visita in ampiezza**
 $\{3, \text{livello1}, \text{livello2}\} = \{3, \{8, 2\}, \{11, 1, 5\}\}$



Alberi 15

ALGORITMI DI VISITA

- Data la natura *intrinsecamente non sequenziale* dell'albero, non è possibile scorrere uno a uno gli elementi con un ciclo!
 - non c'è un concetto di "elemento successivo"
- Occorre necessariamente adottare un *approccio ricorsivo* (come struttura dati!)
 - si opera sulla radice (prima, in mezzo, o dopo)
 - si richiama ricorsivamente la visita sui sottoalberi figli

Alberi 16

VISITA IN ORDINE ANTICIPATO realizzazione STATICA

```
public class Tree {...
static public void preOrder (TreeNode root)
{if ( root != null ) { // solo se non vuoto
  System.out.print(root.item + " "); // root
  preOrder (root.left ); // albero sinistro
  preOrder (root.right ); // sottoalbero destro
}
} // end preOrder ()
}
```

Si sfrutta la parte di classe, come libreria

Alberi 17

ALBERO come struttura dati

```
class TreeNode {
  // Nodo di un albero per alberi binari
  // ogni nodo contiene una stringa
  TreeNode left; // sottoalbero sinistro
  TreeNode right; // sottoalbero destro
  String item; // stringa del nodo

  // Costruttore
  public TreeNode(String str)
    { str.item = str;} // mancano link
} // end class TreeNode

e per partire?
root = new TreeNode(newItem);
```

Alberi 18

VISITA IN ORDINE ANTICIPATO

```
root = new TreeNode(newItem);
```

```
public class Tree { //nodo di albero non vuoto
  ...
  public void preOrder()
  {System.out.println(item);
  left.preOrder();
  right.preOrder();
}
}
```

Alberi 19

VISITA IN ORDINE POSTICIPATO

```
root = new TreeNode(newItem);
```

```
public class Tree { //albero non vuoto
  ...
  public void postOrder() {
  left.postOrder();
  right.postOrder();
  System.out.println(item);
}
}
```

Alberi 20

VISITA IN ORDINE (solo alberi binari)

```
root = new TreeNode(newItem);

public class Tree { //albero non vuoto
    ...
    public void inOrder() {
        left.inOrder();
        System.out.println(item);
        right.inOrder();
    }
}
```

Alberi 21

VISITE & ALGORITMI

- Poiché la visita ricorsiva è *il solo modo per scorrere uno ad uno tutti gli elementi di un albero*, tutti gli algoritmi che operano su alberi sono una "variazione sul tema" di uno degli algoritmi di visita
- Cambia solo l'operazione da fare sulla radice:
 - non più una semplice stampa (print)...
 - ... ma confronti, controlli, test ...

Alberi 22

ESEMPIO

Ricerca di un elemento in un albero

- se l'albero è vuoto, l'elemento non c'è altrimenti (visita in pre-order)
- se tale elemento è la radice, lo si è trovato, altrimenti va cercato nei sottoalberi figli

```
public boolean member(Object e) {
    if (e.equals(item)) return true;
    if (left.member(e)) return true;
    else return right.member(e);
}
```

Alberi 23

ESERCIZIO

Scrivere un algoritmo che, dato un albero di interi, conti gli elementi maggiori di un dato X

- Prendiamo un algoritmo di visita e, al posto della stampa, inseriamo il test richiesto
- Restituiamo il risultato del conteggio nei figli, *aumentato di 1 se il test è risultato positivo*

Esempio Java (caso albero non vuoto)

```
public int countGreater (Comparable x) {
    return left.countGreater (x) +
           right.countGreater (x) +
           (item.compareTo(x)>0 ? 1 : 0);
}
```

Alberi 24

ALBERI BINARI DI RICERCA

- Una *albero binario di ricerca* è un albero binario i cui elementi rispettano tutti una data relazione d'ordine
 - gli elementi devono appartenere a un dominio in cui esista una relazione d'ordine totale
- In particolare, detto R il valore della radice:
 - il sottoalbero di sinistra contiene solo elementi *minori o uguali* a R
 - il sottoalbero di destra contiene solo elementi *maggiori* di R

Alberi 25

ALBERI BINARI DI RICERCA

Perché creare alberi siffatti?

- Un albero binario di ricerca *velocizza enormemente la ricerca di un elemento*, perché *esclude metà albero a ogni confronto*
- Infatti, l'esito del confronto dice inequivocabilmente da quale parte sta l'elemento:
 - nel sottoalbero di sinistra, se l'elemento cercato è *minore* della radice
 - nel sottoalbero di destra, se l'elemento cercato è *maggiore* della radice

Alberi 26

ALBERI BINARI DI RICERCA

Perché creare alberi siffatti?

- Un albero binario di ricerca *velocizza enormemente la ricerca di un elemento*, perché *esclude metà albero a ogni confronto*
 - Infatti, l'esito del confronto dice inequivocabilmente da quale parte sta l'elemento:
 - nel sottoalbero di sinistra, se l'elemento cercato è *minore* della radice
 - nel sottoalbero di destra, se l'elemento cercato è *maggiore* della radice
- Dopo K confronti, ha operato K dimezzamenti
→ ha esplorato uno spazio di 2^K elementi
 - Ricerca binaria: per esplorare N elementi occorrono al più $K = \log_2 N$ confronti.

Alberi 27

ALBERI BINARI DI RICERCA

Come creare un albero binario di ricerca?

- Aggiungendo via via i nuovi elementi in modo da rispettare la relazione d'ordine

Nel caso di alberi esistenti, si aggiungono nuovi nodi

- se l'elemento è minore o uguale alla radice, l'elemento va inserito nel sottoalbero di sinistra
- se è maggiore, va nel sottoalbero di destra
- quando si arriva a un sottoalbero vuoto, si aggiunge l'elemento come nuova foglia

Alberi 28

ALBERO come struttura dati ordinata

```
static TreeNode root; // radice dell'albero
// albero vuoto (empty), root null

static boolean treeContains
    (TreeNode node, String item) {
//albero vuoto
if (node == null){return false;}
// albero da esplorare
else if(item.equals(node.item)){return true;}
else if(item.compareTo(node.item) < 0)
    {return treeContains(node.left,item);}
else {return treeContains(node.right,item);}
} // end treeContains()
```

Alberi 29

ALBERO ORDINATO: esplorazione non ricorsiva

```
static boolean treeContainsNR // non ricorsiva
    (TreeNode node, String item) {
    TreeNode runner = node;
    // per cercare nell'albero dalla radice
    while(true)
    {if (runner == null) { return false;}
    else if(item.equals(runner.item))
        { return true;}
    else if(item.compareTo(runner.item)< 0)
        {runner = runner.left;}
    else {runner = runner.right;}
    }} // end treeContainsNR();
```

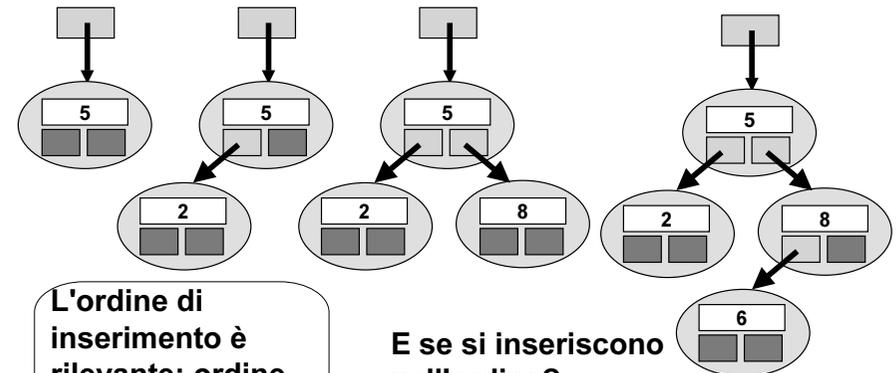
Alberi 30

ALBERO BINARIO di RICERCA

```
static void treeInsert(String newItem) {
if(root == null){root = new TreeNode(newItem); return;}
TreeNode runner = root; // alberi non nulli
while(true)
{if (newItem.compareTo(runner.item) < 0)
    {if (runner.left == null)
        {runner.left = new TreeNode(newItem); return;}
    else runner = runner.left;}
else{ // > 0
    if (runner.right == null)
        {runner.right = new TreeNode(newItem); return;}
    else runner = runner.right;}
} // end while
} // end treeInsert()
```

Alberi 31

Inserimenti successivi



L'ordine di inserimento è rilevante: ordine diverso implica albero diverso

E se si inseriscono nell'ordine?
4, 3, 7, 9
3, 4, 9, 7

Alberi 32

MIGLIORE PROGETTO

Possiamo distinguere tra l'albero come struttura dati (root) e i nodi dell'albero

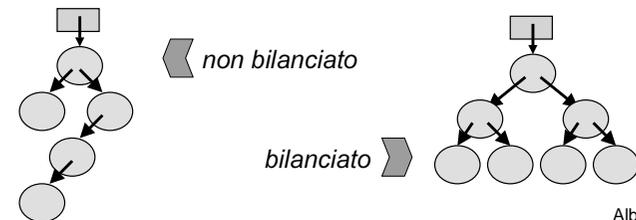
- Ogni singolo elemento può essere la radice e i metodi non sono di classe ma di istanza
- l'inserimento avviene chiedendo al nodo corrente, che passa al successivo

Considerare i problemi in questo caso e come dobbiamo modificare il costruttore

Alberi 33

ALBERI BILANCIATI

- Un albero i cui elementi sono distribuiti uniformemente fra i due sottoalberi si dice *bilanciato*
- Molti algoritmi funzionano meglio con alberi bilanciati
- Esistono algoritmi per rendere bilanciati, spostando alcuni elementi, alberi che non lo siano



Alberi 34

JAVA: GLI ALBERI PREDEFINITI

Il package `java.util` definisce:

- *interfacce* che rappresentano *tipi di dati astratti di contenitori* (cfr. `SortedSet`)
- *classi* che forniscono *specifici contenitori concreti* (cfr. `TreeSet`, che usa `TreeMap`)

I metodi principali (`add`, `remove`, `contains`, `isEmpty`, ...) sono introdotti da `Collection` e specializzati dalle varie classi

Alberi 35

Alberi 36