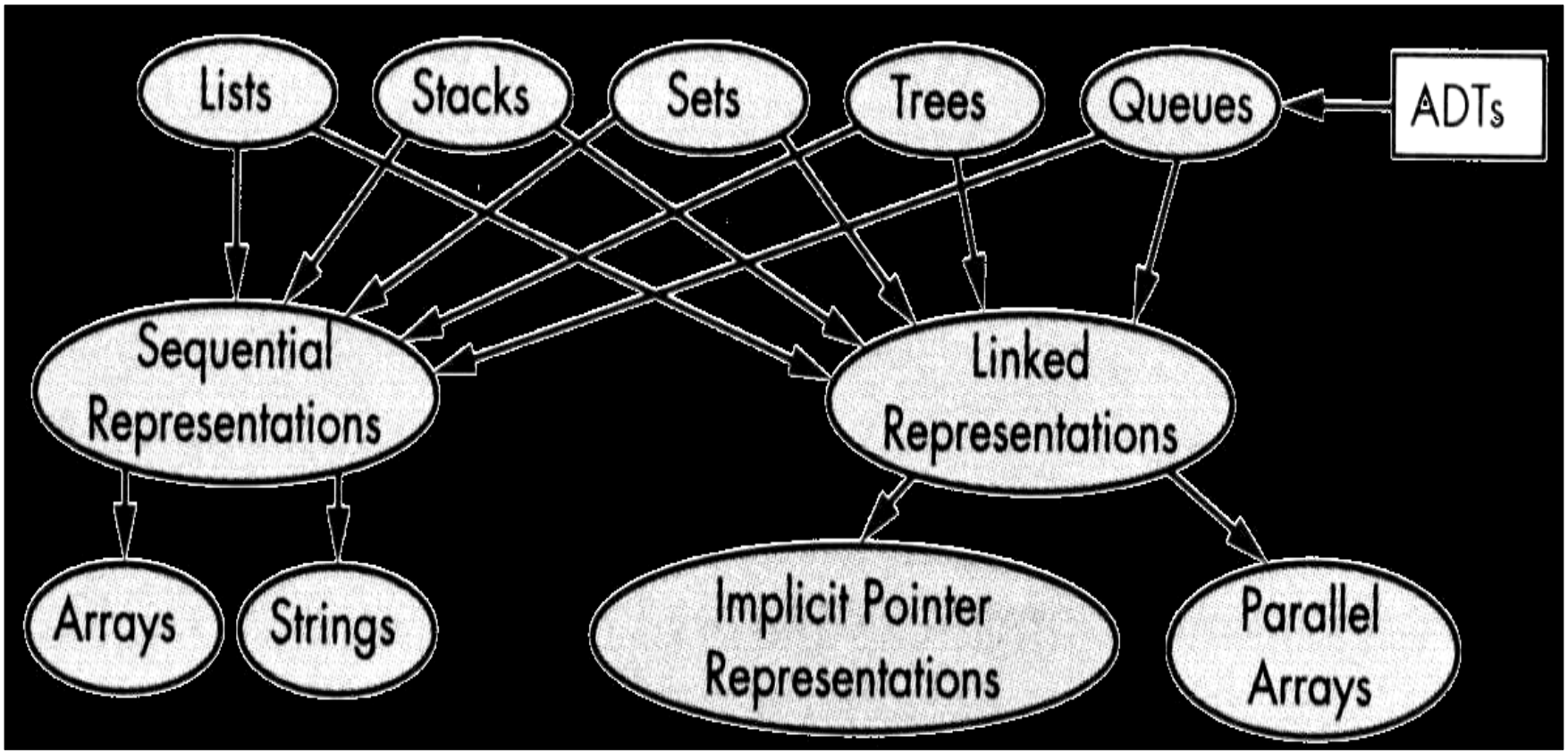
Implementazione ADT: Alberi

- Livelli di astrazione

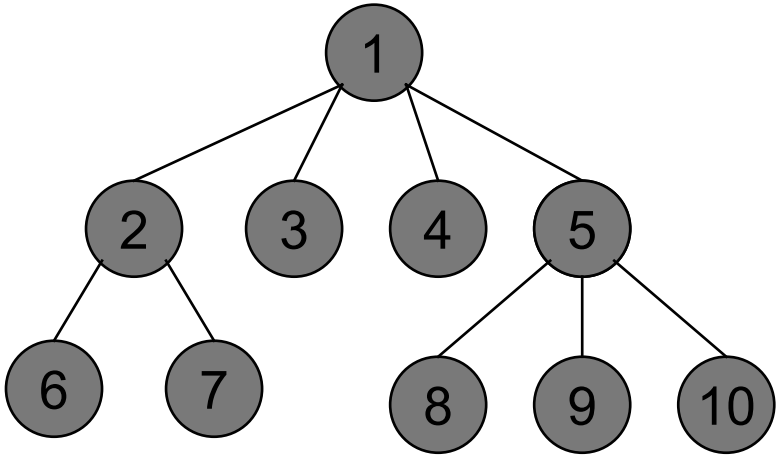


Esercitazione 5 (E6): alberi (tree)

- albero ≈ struttura dati fondamentale, soprattutto per le operazioni di ricerca
- tipi di albero con radice ordinati:
  - binari
  - avl, RedBlack, Btree, ecc.
- diverse implementazioni
  - array paralleli
  - strutture dinamiche (linked)
- maggior attenzione per gli alberi binari di ricerca

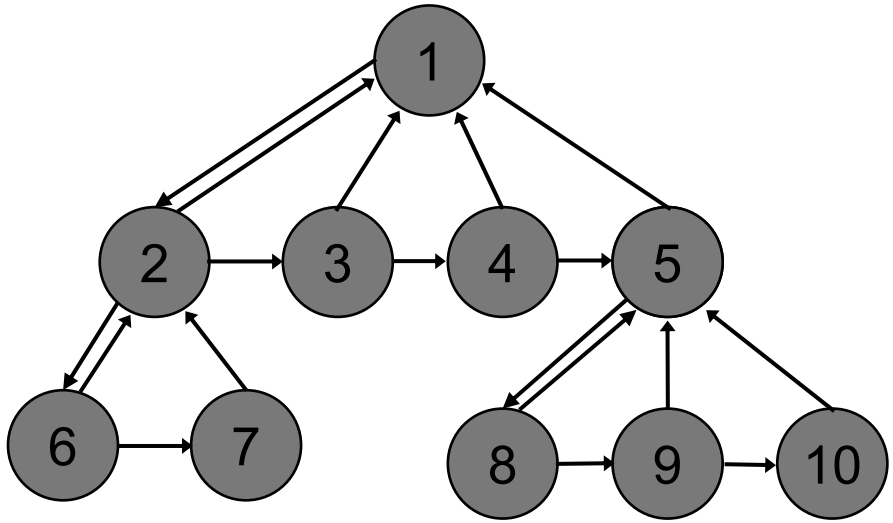
E6: alberi

- Esempio di rappresentazione



Rappresentazione classica

Rappresentazione modulare

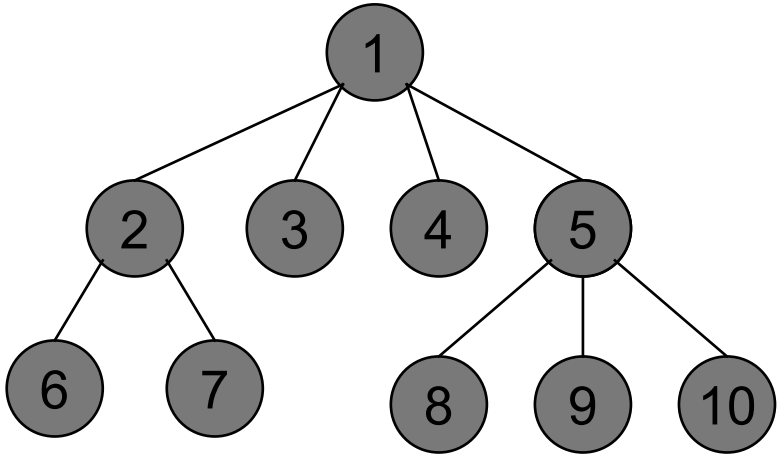


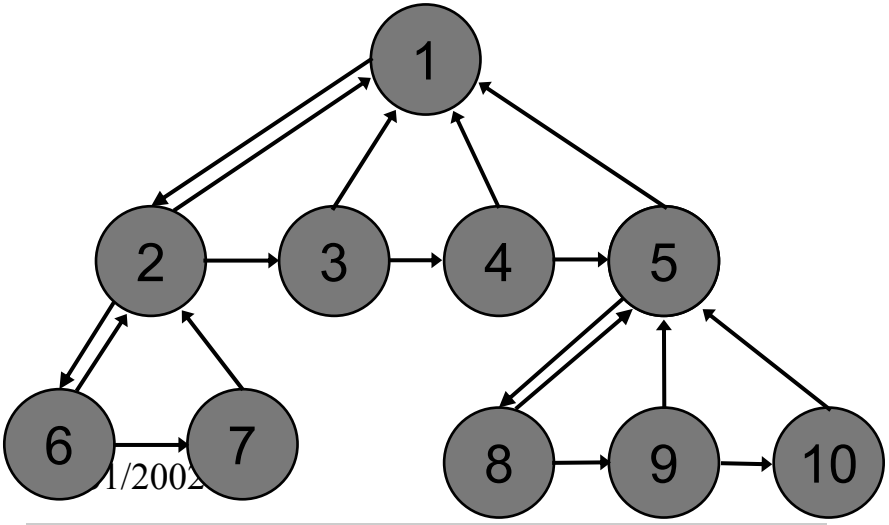
E6: alberi - implementazione

- Un'implementazione con vettori paralleli
  - 3 vettori di dimensione = # nodi:
    - Padre, Figlio, Fratello
  - Padre[*i*]= *j* se *j* è padre di *i*, *0* se è radice
  - Figlio[*i*]= *j* se *j* è il primo figlio di *i*, *0* se è foglia
  - Fratello[*i*]= *j* se *j* è il fratello successivo di *i*, *0* se non ci sono fratelli successivi

E6: alberi - implementazione

- Esempio implementaz. con vettori paralleli





	P	Fi	Fr
1	0	2	0
2	1	6	3
3	1	0	4
4	1	0	5
5	1	8	0
6	2	0	7
7	2	0	0
8	5	0	9
9	5	0	10
10	5	0	0

E6: alberi - implementazione

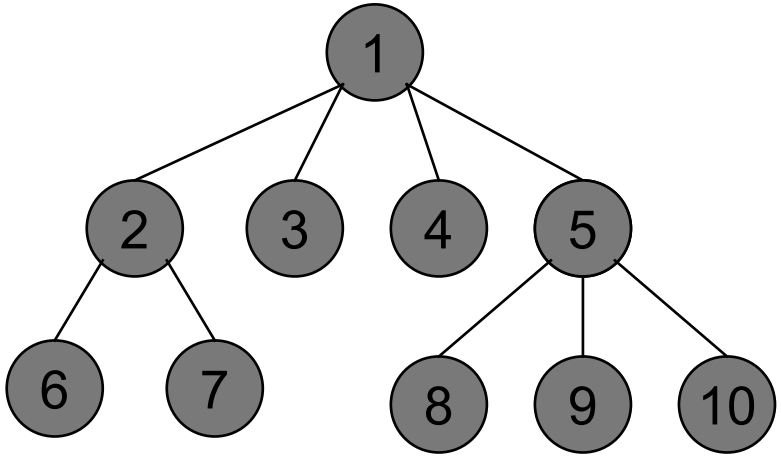
- Un'implementazione con strutture dinamiche
  - struttura dati di base:

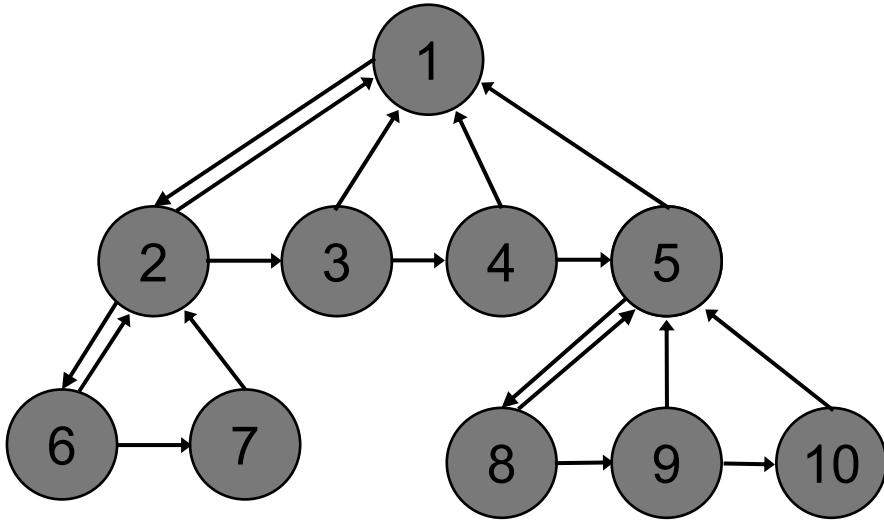
```
public class Node {
   Object elemento;              //le variabili sono visibi
   Node padre, figlio, fratello;//a livello package
   ...
}
```

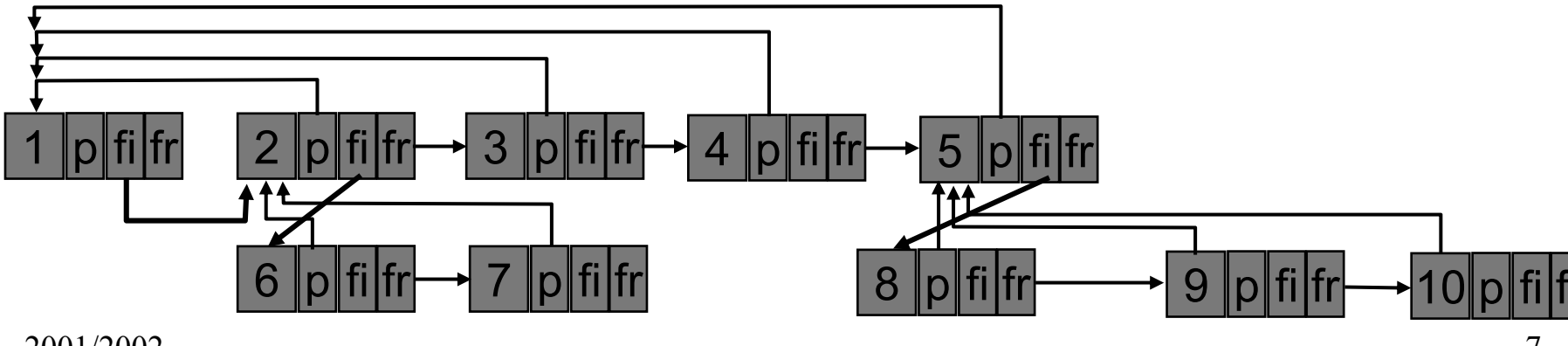
  - un albero: *Node radice = new Node();*

E6: alberi - implementazione

- Esempio implem. con strutture dinamiche







E6: alberi - implementazione

- Quali operazioni?
  - per gli alberi radicati ordinati le operazioni sono a basso livello (orientate ai nodi)
  - per alberi particolari si definiscono invece operazioni generali a più alto livello (vedi alberi di ricerca)
  - per tutti, si definiscono le **operazione di visita**

E6: alberi - operazione visita

- Visita (attraversamento) =
  - operazione dove ciascun nodo è esaminato esattamente una volta in qualche particolare ordine
- Tipi visite più comuni:
  - **pre-ordine,**
  - **post-ordine,**
  - **in-ordine,** (solo per gli alberi binari)

E6: alberi - visita pre-ordine

- Visita pre-ordine ricorsiva

```
/**
 * visitaPreOrdine(Node nodo) esegue una visita in pre-ordine dell'alber
 * con radice nodo. In ogni nodo visitato, richiama la procedura
 * visita(Object)
 */
void visitaPreOrdine(Node nodo) {
  if (nodo != null) {                          // verifico se siamo fine ricorsione
    visita(nodo);                              // prima visito nodo
    Node figlio = nodo.figlio;
    while (figlio != null) {      // per ciascun figlio (sottoalbero) eseguo visita
      visitaPreOrdine(figlio);                 // pre-ordine in modo ricorsivo
      figlio = figlio.fratello;
    }
  }
}
```

E6: alberi - visita post-ordine

- Visita post-ordine ricorsiva

```
/**
 * visitaPostOrdine(Node nodo) esegue una visita in post-ordine dell'albero
 * con radice nodo. In ogni nodo visitato, richiama la procedura
 * visita(Object)
 */
void visitaPostOrdine(Node nodo) {
  if (nodo != null) {                          // verifico se siamo fine ricorsione
    Node figlio = nodo.figlio;
    while (figlio != null) {   // per ciascun figlio (sottoalbero) eseguo visita
      visitaPostOrdine(figlio);                // post-ordine in modo ricorsivo
      figlio = figlio.fratello;
    }
    visita(nodo);                              // visito nodo
  }
}
```

E6: alberi - visita pre-ordine iterativa

- Visita pre-ordine ITERATIVA

```
/** visitaPreOrdineI(Node nodo) esegue una visita in pre-ordine dell'albero
 * con radice nodo. In ogni nodo visitato, richiama la procedura
 * visita(Object) */
void visitaPreOrdineI(Node nodo) {
  Pila pila = new PilaArray();                    //struttura di supporto
  Node n;
  pila.push(nodo);
  while (! pila.eVuota()) {
      n = (Node) pila.pop();
      if (n != null) {
        visita(n);
        Pila pilaFigli = new PilaArray();  //pila di supporto per memorizzare i fig
        Node son = n.figlio;
```

E6: alberi - visita pre-ordine iterativa

- Visita pre-ordine ITERATIVA - continua

```
        while (son != null) {                    //memorizzo tutti i figli nella pilaFig
          pilaFigli.push(son);
          son = son.fratello;
        }
        while (! pilaFigli.eVuota()) {           //inserisco nella pila di chiamata
                                                 // i figli, nell'ordine corretto
          pila.push(pilaFigli.pop());
        }
      }
    }
}
```

E6: alberi - alberi binari di ricerca

- Albero binario di ricerca = albero radicato ordinato dove ogni nodo interno ha al più due figli: ***figlio sx e figlio dx***
- nuovo tipo di visita = in-ordine
  - il nodo è visitato dopo la visita del figlio sx e prima della visita figlio dx.
- le operazioni sono definite a livello di albero:
  - find(Object), insert(Object), delete(Object)
- le operazioni di inserimento/cancellazione devono preservare l'ordinamento

E6: alberi binari ricerca-visita in-ordine

```
/**
 * visitaInOrdine(Node nodo) esegue una visita in-ordine
 * dell'albero BINARIO con radice nodo. In ogni nodo visitato,
 * richiama la procedura visita(Object)
 */
void visitaInOrdine(Node nodo) {
  if (nodo != null) {          // verifico se siamo fine ricorsione
    visitaInOrdine(nodo.figlioSx);
    visita(nodo);              // visito nodo
    visitaInOrdine(nodo.figlioDx);
  }
}
```

- Per visita pre e post-ordine le procedure sono analoghe

E6: alberi binari ricerca - find

```
/**
 * find(Object item) cerca l'oggetto item all'interno dell'albero
 * Restituisce il nodo che contiene item se esiste, null altriment
*/
public BinaryNode find (Object item) {     //versione iterativa
   BinaryNode node = radice;
   int result;
   while (node != null) {
     if ( (result = ((Comparable) item).compareTo(node.elemento())) < 0) {
       node = node.figlioSx;                  // item può essere nel sottoalbero sx
     } else
       if (result == 0)                        //trovato!
         return node;
       else
         node = node.figlioDx;                // item può essere nel sottoalbero dx
   }
   return node;  }                            // la ricerca è fallita, ritorno null
```

E6: alberi binari ricerca - insert

```
/**
 * inserisci(Object item) inserisce l'oggetto item in un nodo x
 * all'interno dell'albero in modo tale che il sottoalbero sx
 * di x contenga elementi inferiori e il sottoalbero dx di x
 * elementi superiori a item
 */

public void insert(Object item) {
   radice = insertItem(item, radice, null);
 }
```

E6: alberi binari ricerca - insert

```
private BinaryNode insertItem(Object item, BinaryNode nodo,
        BinaryNode padre) {
  int result;
  if (nodo == null) {          // fine ricerca posizione, definisco un nuovo nod
    BinaryNode n = new BinaryNode(item);
    n.padre = padre;
    return n;
  } else {                     //cerco posizione dove inserire l'elemento
    if ( (result = (Comparable) item).compareTo(nodo.elemento)  < 0 ) {
      nodo.figlioSx = insertItem(item, nodo.figlioSx, nodo);
      return nodo;
    } else {
      if (result == 0) return nodo;
      else {
        nodo.figlioDx = insertItem(item, nodo.figlioDx, nodo);
        return nodo;
      }
    }  }  }
```

E6: alberi binari ricerca - delete

- La cancellazione deve preservare l'ordine nell'albero

```
public void delete(Object item) {
   //cerco il nodo che contiene item
   BinaryNode daCancellare = find(item);
   if (daCancellare == null)
      return;
    else
      deleteNode(daCancellare);          //cancello dall'albero
 }
```

E6: alberi binari ricerca - delete

```
private void deleteNode(BinaryNode daCancellare) {
    BinaryNode temp;

    if (daCancellare.figlioSx == null)          //non c'è il figlio sx, prendo il dx
      temp = daCancellare.figlioDx;             //(che può essere null)
    else {
      if (daCancellare.figlioDx == null)        //esiste solo figlio sx
        temp = daCancellare.figlioSx;
      else {                                    // ci sono entrambi i figli...
        if (daCancellare.figlioSx.figlioDx == null) { //se il figlioSx ha solo figlioS
          temp = daCancellare.figlioSx;
          temp.figlioDx = daCancellare.figlioDx; //posso spostare il figlioDx
          temp.figlioDx.padre = temp;
        } else {
          temp = inOrderPrec(daCancellare); //cerco nodo precedente nella
                                            //sequenza di visita in ordine
```

E6: alberi binari ricerca - delete

```
        daCancellare.elemento = temp.elemento; //scambio gli elementi
        deleteNode(temp);                      //cancello il nodo temp
        temp = daCancellare; //allineo daCancellare e temp
                             //per gli aggiustamenti successivi
      }
    } }                                        //ora si può fare la sostituzione
  if (daCancellare == radice) {//verifico che non sia root
    radice = temp;
    if (radice != null) radice.padre = null;
  } else {
    if (daCancellare.padre.figlioSx == daCancellare) //daCancellare è figlio s
      daCancellare.padre.figlioSx = temp;
    else
      daCancellare.padre.figlioDx = temp;
    if (temp != null)
      temp.padre = daCancellare.padre;
  } }
```

E6: alberi - Iteratori

- **Iteratore**
  - struttura dati che permette di accedere a gli oggetti di una qualsiasi classe indipendentemente dall'implementazione di questa in modo ordinato e per una sola volta.
  - java.util.Iterator è un'interfaccia

    file://jdk1.3/docs/api/java/util/Iterator.html

  definita dai seguenti metodi:
  - **boolean hasNext();**
  - **Object next();**
  - **void remove();**

E6: alberi - Iteratori

- Per l'adt Albero gli Iteratori possono essere realizzati con i metodi di visita
- Esempio:

```
public class TreeIterator implements java.util.Iterator {
   private BinaryNode indice;
   private PilaArray nodiDaVisitare;
...
   TreeIterator(BinaryNode nodoInizio) {
        nodiDaVisitare.push(nodoInizio);
   }
```

E6: alberi - Iteratori

- Esempio (continua):

```
/** Deve restituire vero se ci sono ancora elementi da
 * visitare, falso altrimenti
*/
public boolean hasNext() {
   return ( ! nodiDaVisitare.eVuota() );
}
```

E6: alberi - Iteratori

- Esempio (continua):

```
public Object next() throw java.util.NoSuchElementException {
    if (! nodiDaVisitare.eVuota() ) {
        indice = nodiDaVisitare.pop();
        if (indice.figlioDx!=null)
            nodiDaVisitare.push(indice.figlioDx);
        if (indice.figlioSx!=null)
            nodiDaVisitare.push(indice.figlioSx);
        return indice;
    } else
        throws new java.util.NoSuchElementException();
}
```

E6: alberi - Iteratori

- Definizione di un **Iteratore** per BinaryTree

```
/*dentro definizione classe BinaryTree*/

public Iterator iterator() {
    return ( new TreeIterator(radice) );
}
```

E6: alberi - compito esercitazione

- Compito dell'esercitazione
  - realizzare la classe Albero binario di ricerca con tutte le operazioni
  - realizzare il metodo toString() per la classe albero binario che stampi la struttura dell'albero (Suggerimento: la stampa è più facile se è ruotata di 90°… e se si visita l'albero in un certo modo)
  - Stampare l'albero di ricerca che si ottiene con l'inserimento di 2,1,4,10,3,7,11,11 e la cancellazione di 4