Insegnamento di Laboratorio di algoritmi e strutture dati

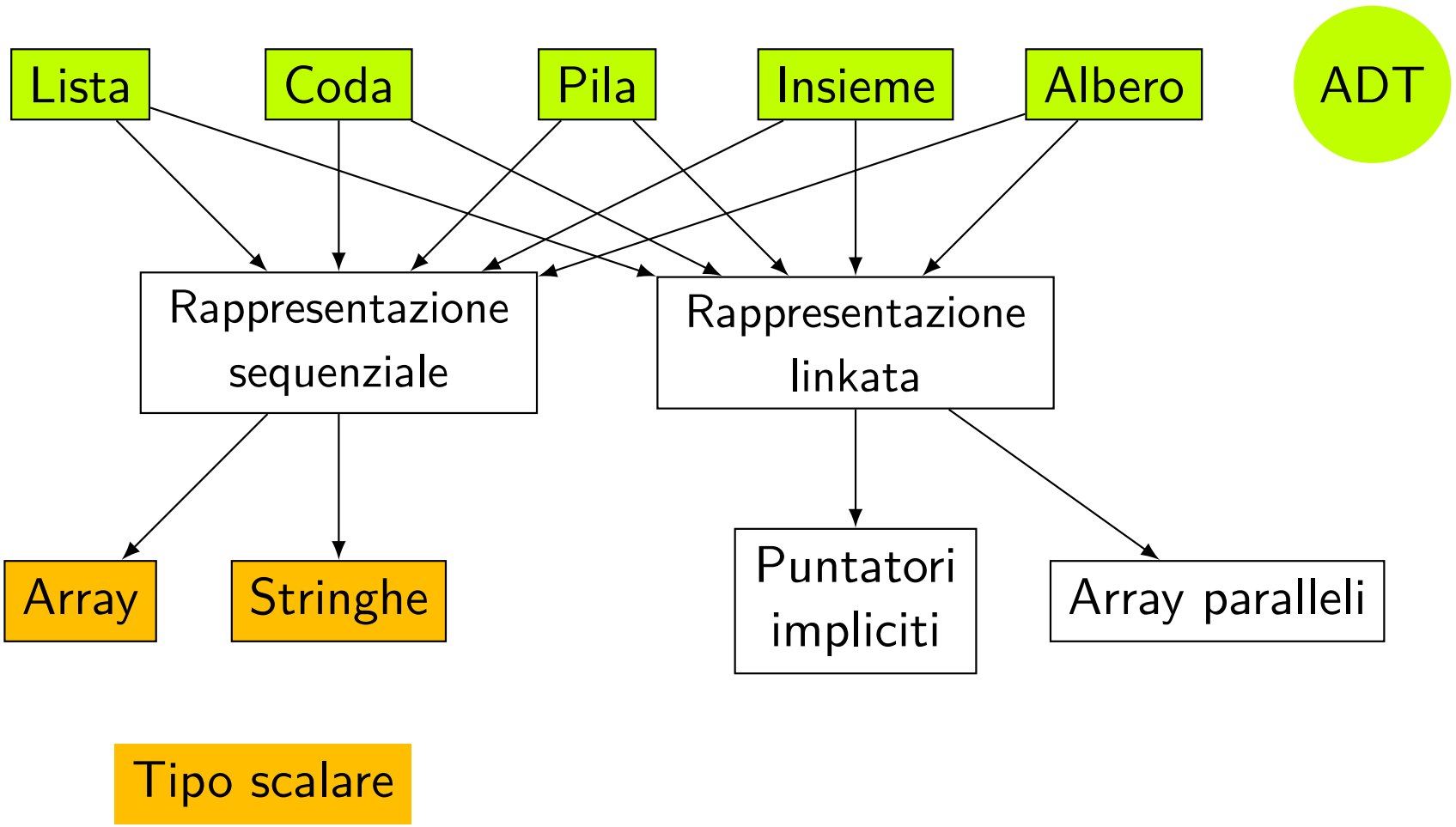
ADT Albero

Roberto Posenato

ver. 0.7, 29/01/2008

ADT Albero

Schema generale ADT

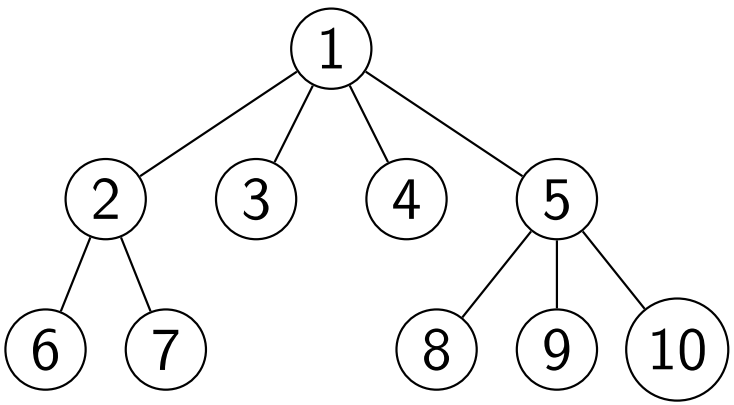


Albero

- 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.

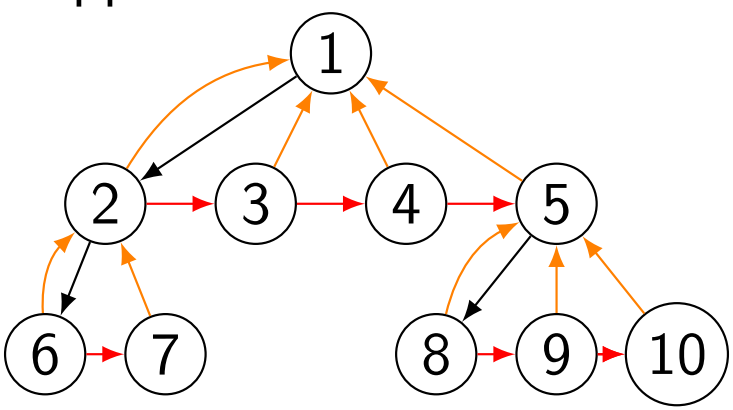
Albero

Rappresentazione albero generale



Rappresentazione classica

Rappresentazione modulare



Albero

Implementazione albero generale con strutture dinamiche

Interfaccia per rappresentare un nodo dell'albero generico

```
/** ... */
public interface Node {
    /** @return vero se il nodo non contiene informazione,
    * false altrimenti. */
    boolean isEmpty();

    /** @return l'informazione contenuta nel nodo. */
    Object getElement();

    /** Aggiorna l'informazione associata al nodo.
    * @param o la nuova informazione. */
    void setElement(Object o);

    /** @return il numero dei figli del nodo. */
    int degree();
...
```

Albero

Implementazione albero generale con strutture dinamiche

continua interfaccia per rappresentare un nodo dell'albero generico

```
    /** ...
     * @param i indice del son.
     * @return riferimento al nodo son di indice i-esimo.
     * @throws IndexOutOfBoundsException se i è negativo
     * o i >= degree() */
    Node getSon(int i)
            throws IndexOutOfBoundsException;

    /** ...
     * @param i indice del son
     * @param o la nuova informazione.
     * @throws IndexOutOfBoundsException se i è negativo
     * o i > degree() */
    void setSon(int i, Object o)
            throws IndexOutOfBoundsException;
...
```

Albero

Implementazione albero generale con strutture dinamiche

continua interfaccia per rappresentare un nodo dell'albero generico

```
    /** Rimuove il figlo i-esimo.
     * @param i indice del son.
     * @throws IndexOutOfBoundsException se i è negativo
     * o i >= degree() */
    void removeSon(int i)
            throws IndexOutOfBoundsException;

    /** ... */
    Node getFather();

    /** Assegna un nuovo padre al nodo.
     * Non è richiesta l'equivalenza con p.setSon(i, this).
     * @param p nuovo nodo padre. */
    void setFather(Node p);
}
```

Albero

Implementazione albero generale con strutture dinamiche

Una classe che implementa Node

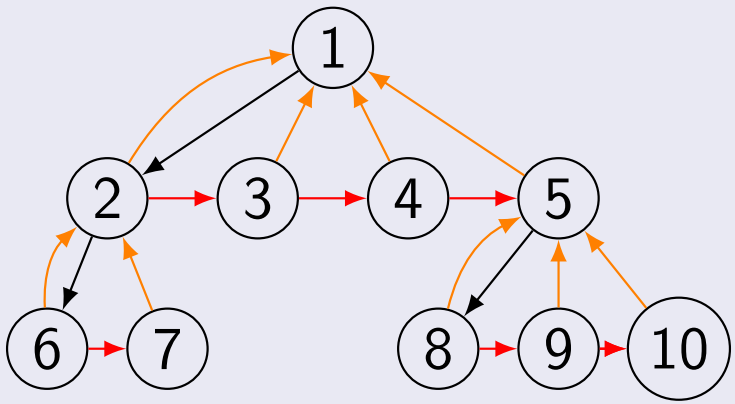
```
/** ... */
public class GenericNode implements Node {
    /** Data of the node. */
    Object element;

    /** ... */
    Node father;

    /** first son */
    Node son;

    /** the right node of same level */
    Node brother;
    ...
```

Da completare per esercizio!

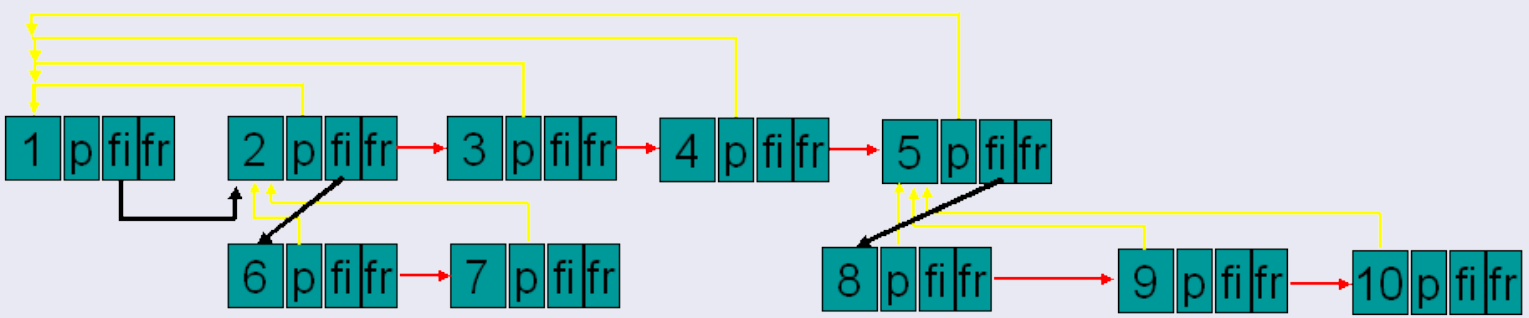
Albero

Rappresentazione e implementazione albero generale

Rappresentazione modulare



Implementazione con la classe `GenericNode`



Albero

Albero generale: operazioni fondamentali

Quali operazioni?

- per gli alberi radicati, le operazioni sono a basso livello (orientate ai nodi);
- per gli alberi con particolari strutture, le operazioni sono anche a più alto livello (vedi alberi di ricerca);
- per tutti, sono definite le operazione di visita

Albero

Albero generale: operazioni di 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)

Albero

Albero generale: visita pre-ordine

Interfaccia albero radicato

```
/** ... */
public interface Tree {
    /**
    * @return the root node of the tree,
    * null if it is empty
    */
    Node getRoot();

    /**
     * Update the root to n.
     * @param n the new root
     */
    void setRoot(Node n);
...
```

Albero

Albero generale: visita pre-ordine

continua Interfaccia albero radicato

```
    /** Traversal type */
    static enum Traversal {

        /** Visits current node first, sons after.*/
        PREORDER,

        /** Visits left son first,
        * current node after and right son last.
        * This visit is defined only for binary tree. */
        INORDER,

        /** Visits sons first, current node after */
        POSTORDER
    }
...
```

Albero

Albero generale: visita pre-ordine

continua Interfaccia albero radicato

```
    /** Makes a visit operation on node n.
     * @param n node to visit */
    void visit(Node n);

    /** Makes a type traversal of this tree.
     * For each node, calls #visit(Node).
     *
     * @param type a kind of traversal.
     */
    void traversal(Traversal type);
}
```

Albero

Albero generale: visita pre-ordine

Visita pre-ordine ricorsiva

```
/**
* Visita l'albero in pre-ordine in modo ricorsivo.
* @param nodo nodo da dove iniziare la visita.
*/
private void preOrdine(GenericNode node) {
    if (node != null) {
        visit(node);
        GenericNode s = node.son;
        while (s != null) {
            preOrdine(s);
            s = s.brother;
        }
    }
}
```

Albero

Albero generale: visita post-ordine

Visita post-ordine ricorsiva

```
/**
* Visita l'albero in post-ordine in modo ricorsivo.
*
* @param nodo nodo da dove iniziare la visita.
*/
private void postOrdine(GenericNode node) {
    if (node != null) {
        GenericNode s = node.son;
        while (s != null) {
            preOrdine(s);
            s = s.brother;
        }
        visit(node);
    }
}
```

Albero

Albero generale: visita pre-ordine iterativa

Visita pre-ordine iterativa

```
/**
 * Visita l'albero in pre-ordine iterativo da nodo.
 * @param nodo punto di inizio visita.
 */
void preOrdineI(GenericNode nodo) {
   Pila pila = new PilaArray(nodo),
        pilaDeiFigli = new PilaArray();
   Node corrente, son;

   while (!pila.isEmpty()) {
      corrente = (GenericNode) pila.pop();
      visit(corrente);
      son = corrente.son;
...
```

Albero

Albero generale: visita pre-ordine iterativa

continua Visita pre-ordine iterativa

```
        //Memorizzo lista dei figli rovesciata
        while (son != null) {
            pilaDeiFigli.push(son);
            son = son.brother;
        }
        //Riverso la lista figli dritta in pila
        while (!pilaDeiFigli.isEmpty()) {
            pila.push(pilaDeiFigli.pop());
        }
    }
}
```

Albero

Albero generale: implementazione di `traversal(Traversal type)`

`traversal(Traversal type)`

```
public void traversal(Traversal type) {
    switch (type) {
        case INORDER:
            throw new IllegalArgumentException();
        case PREORDER:
            preOrdine(root);
            break;
        case POSTORDER:
        default:
            postOrdine(root);
            break;
    }
}
```

Albero binario di ricerca
Introduzione

Albero binario di ricerca:

- albero radicato ordinato dove ogni nodo ha al più due figli: figlio sx e figlio dx
- nuovo tipo di visita: **in-ordine**
  - il nodo corrente è visitato dopo la visita del figlio sx e prima della visita figlio dx.
- le operazioni sono definite a livello di albero:
  - `get(Comparable)`, `put(Comparable)`, `remove(Comparable)`
- le operazioni di inserimento/cancellazione devono preservare l'ordinamento.

Albero binario di ricerca

Albero binario: interfaccia

Interfaccia

```
/** ... */
public interface BinaryTree extends Tree {
    /** Puts item into the tree in a position that
     * preserves the natural order. ... */
    void put(Comparable item);

    /** Gets item.
     * @param item
     * @return the node that contains item if it is
     * present, null otherwise. ...*/
    BinaryNode get(Comparable item);

    /** Remove item ... */
    void remove(Comparable item);
}
```

Albero binario di ricerca

Visita in-ordine

Visita in-ordine ricorsiva di un albero binario

```
/**
 * Visita l'albero binario in-ordine in modo ricorsivo.
 *
 * @param nodo punto inizio visita.
 */
void inOrdine(BinaryNode nodo) {
    if (nodo != null) {
        inOrdine(nodo.leftSon);
        visit(nodo);
        inOrdine(nodo.rightSon);
    }
}
```

Per visita pre e post-ordine i metodi sono analoghi.

Albero binario di ricerca

Ricerca elemento

Ricerca di un elemento

```
/** Cerca item nell'albero.
 * @param item elemento da cercare
 * @return il nodo che contiene item se esiste, null altr.
 * @throws NullPointerException if item is null */
BinaryNode get(Comparable item) {
  if (item==null)
    throw new NullPointerException();
  BinaryNode node = root; int result;
  while (node != null) {
    if ((result=item.compareTo(node.element))<0)
       node = node.leftSon;
    else if (result == 0) return node;
    else node = node.rightSon;
  }
  return node;
}
```

Albero binario di ricerca

Inserimento elemento

Inserimento di un elemento

Volendo fare una procedura ricorsiva, prima si offre il metodo di chiamata pubblico.

```
/** Inserisce item nel nodo corretto di questo albero.
 * L'albero è mantenuto ordinato secondo l'ordine naturale
 * della classe di item.
 *
 * @param item oggetto da inserire nell'albero.
 * @throws NullPointerException if item is null
 */
public void put(Comparable item) {
    if (item==null)
        throw new NullPointerException();
    root = putStartingFrom(item, root, null);
}
```

Albero binario di ricerca

Inserimento elemento

Metodo ricorsivo di inserimento di un elemento

```
/** Inserisce item se non è già presente in questo albero
 * a partire da nodo mantenendo l'ordine dell'albero.
 * @param item elemento da aggiungere.
 * @param nodo punto di partenza.
 * @param padre padre di nodo.
 * @return nodo se diverso da null, nuova root
 *         albero che contiene item altrimenti. */
private BinaryNode putStartingFrom(Comparable
    item, BinaryNode nodo, BinaryNode padre){
    int result;
    if (nodo == null) { // definisco un nuovo nodo
        BinaryNode n = new BinaryNode(item);
        n.father = padre;
        return n;
    }
...
```

Albero binario di ricerca

Inserimento elemento

continua metodo ricorsivo di inserimento di un elemento

```
  if ((result=item.compareTo(nodo.element))<0){
    nodo.leftSon = putStartingFrom(
      item, nodo.leftSon, nodo);
    return nodo;
  }

  if (result == 0) return nodo;

  nodo.rightSon = putStartingFrom(
    item, nodo.rightSon, nodo);
  return nodo;
}
```

Albero binario di ricerca

Cancellazione elemento

Cancellazione di un elemento

```
/**
 * Cancella <code>item</code> se esite.
 *
 * @param item elemento da cancellare.
 * @throws NullPointerException if item is null
 */
public void remove(Comparable item) {
    BinaryNode daCanc = get(item);

    if (daCanc != null)
        deleteNode(daCanc);
}
```

Albero binario di ricerca

Cancellazione elemento

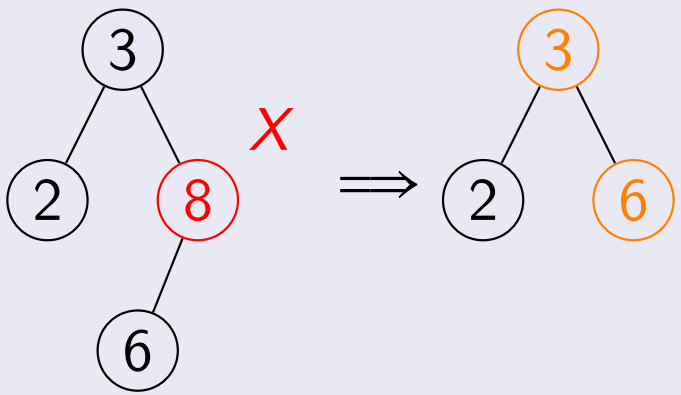
Si consideri che si debba cancellare il nodo X. I casi possibili cancellazione sono:

1. X **non ha figli** $\Rightarrow$ Si cancella X e si è finito.
2. **Manca uno dei figli di** X $\Rightarrow$ Il figlio presente prende il posto di X.
3. **Ci sono entrambi i figli di** X **e il figlio sinistro ha solo il figlio sinistro** $\Rightarrow$ Il figlio sinistro di X prende il posto di X e il figlio destro di X diventa figlio destro di X.
4. **Ci sono entrambi i figli di** X $\Rightarrow$ Il nodo X deve essere sostituito dal nodo di valore maggiore presente nel sotto albero sinistro.
   Il nodo di valore maggiore è il nodo più a destra del sotto albero sinistro di X.

Albero binario di ricerca

Cancellazione elemento

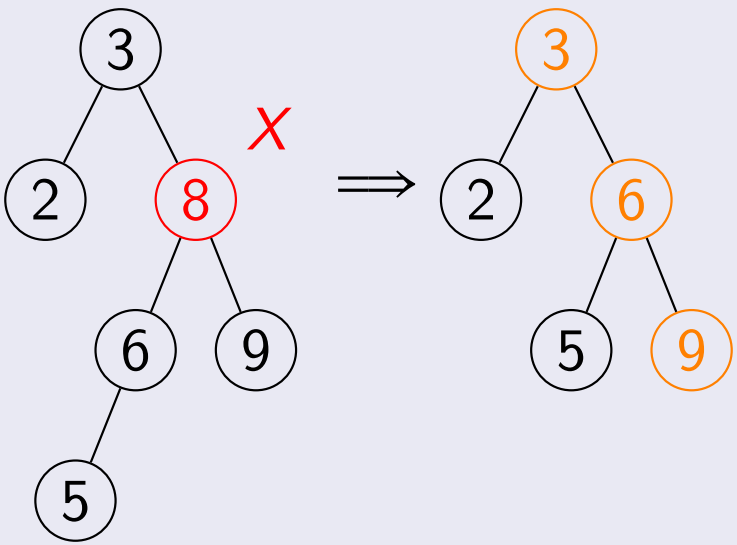
Manca uno dei figli di X



Albero binario di ricerca

Cancellazione elemento

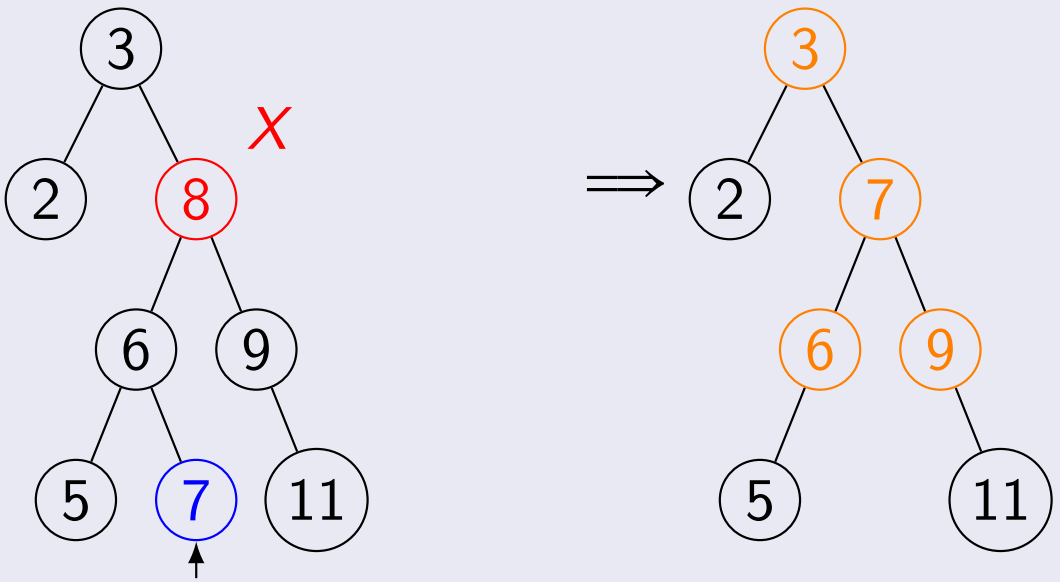
Ci sono entrambi i figli di X e il figlio sinistro ha solo il figlio sinistro



Albero binario di ricerca

Cancellazione elemento

Ci sono entrambi i figli di X



Elemento maggiore inferiore a X

Albero binario di ricerca

Cancellazione elemento

Metodo di cancellazione di un elemento

```
/**Cancella il nodo daCanc mantenendo
 * l'ordine dell'albero.
 * @param daCanc nodo da cancellare.
 */
private void deleteNode(BinaryNode daCanc) {
    BinaryNode temp; // mantiene il riferimento al nodo
                     // che andrà a sostituire il nodo daCanc

    if (daCanc.leftSon == null) //non c'è sx
        temp = daCanc.rightSon;     //prendo il dx
    else {
        if (daCanc.rightSon == null) //non c'è dx
            temp = daCanc.leftSon; // prendo il sx
        else { // ci sono entrambi i figli...
...
```

Albero binario di ricerca

Cancellazione elemento

continua metodo ricorsivo di cancellazione di un elemento

```
        if (daCanc.leftSon.rightSon==null){
            // se il leftSon ha solo leftSon
            temp = daCanc.leftSon;
            temp.rightSon=daCanc.rightSon;
            temp.rightSon.father = temp;
         } else {
           //cerco nodo precedente nella sequenza di visita in ordine
            temp = inOrderPrec(daCanc);
            // scambio gli elementi
            daCanc.elemento=temp.elemento;
            deleteNode(temp);
            temp = daCanc;
         }
      }
   }
...
```

Albero binario di ricerca

Cancellazione elemento

continua metodo ricorsivo di cancellazione di un elemento

```
    // ora si può fare la sostituzione di daCanc con temp
    if (daCanc == root) {
        // se è root, aggiusta il padre a null
        root = temp;
        if (root != null) root.father = null;
    } else {
        if (daCanc.father.leftSon==daCanc)
            // daCanc è figlio sx
            daCanc.father.leftSon = temp;
        else daCanc.father.rightSon = temp;
        if (temp != null)
            temp.father = daCanc.father;
    }
}
```

Alberi

Compito esercitazione

Compito esercitazione

- Implementare la classe albero binario di ricerca completando i metodi `put()`, `remove()`, `get()`, `visit()`, `inOrdine()`, `preOrdine()`, `postOrdine()`;
- Scrivere 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 ricorsivo!)
- Stampare l'albero di ricerca che si ottiene dopo aver inserito $2, 1, 4, 10, 3, 7, 11, 11$ e stampare dopo che si è cancellato l'elemento di valore 4