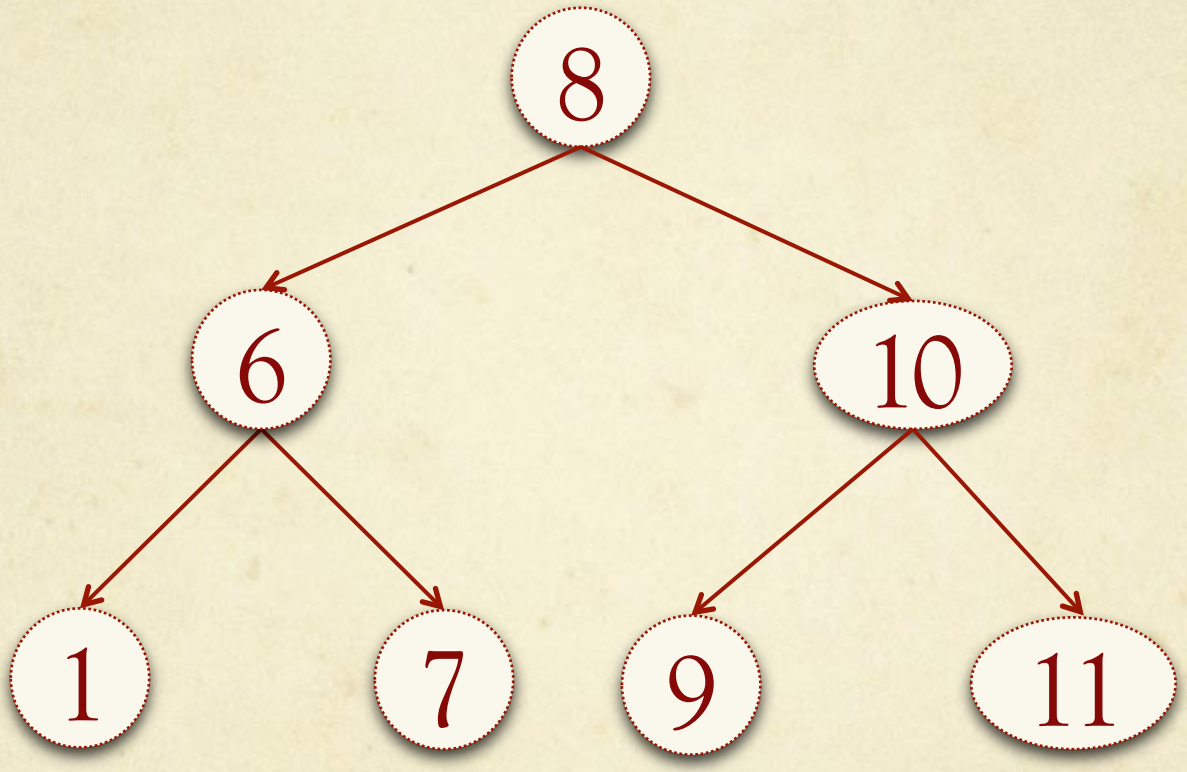
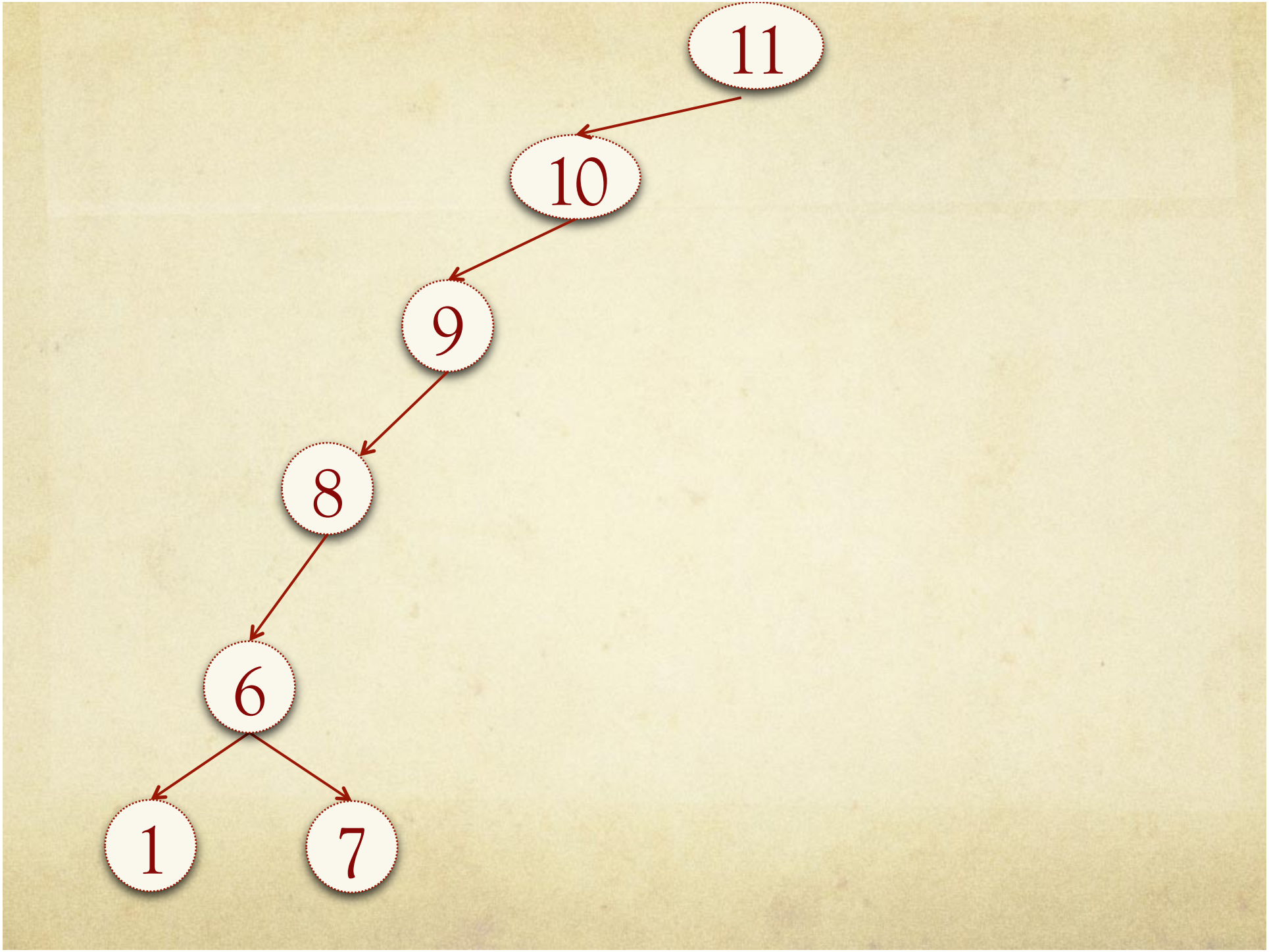
# Alberi binari di ricerca

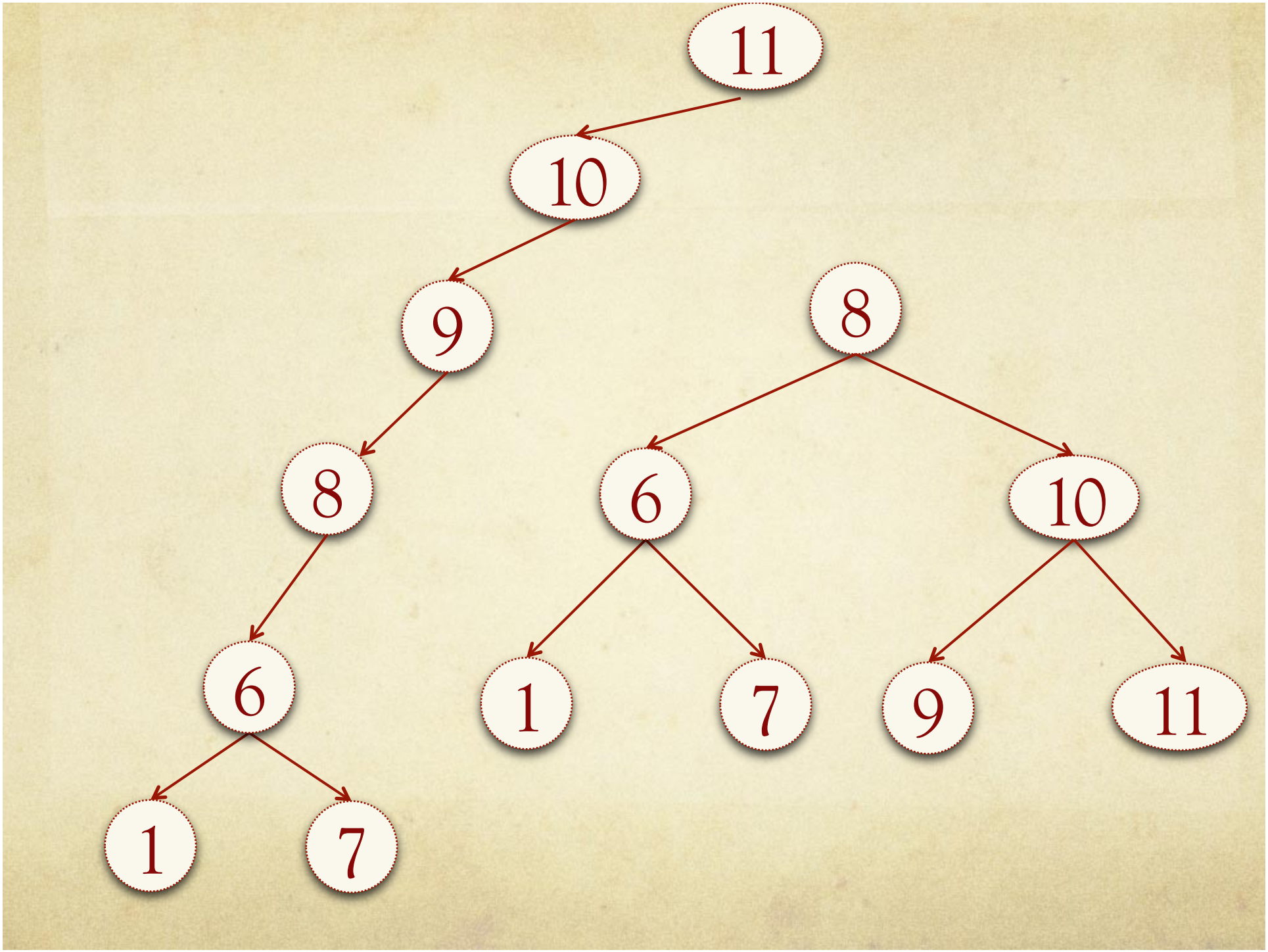
# Definizione

Un albero si dice albero binario di ricerca è un albero binario in cui:

- Ogni nodo è caratterizzato un valore chiamato chiave
- L'insieme delle chiavi è totalmente ordinato.
- Per ogni nodo u vale questa proprietà:
    - le chiavi presenti nel sottoalbero sinistro sono minori di u
    - le chiavi presenti nel sottalbero destro sono maggiori di u


8
6
10
1
7
9
11


11
10
9
8
6
1
7


11
10
9
8
8
6
10
6
1
7
9
11
1
7

# Implementazione

# Implementazione

```
private class Node{

	Comparable elem;
	Node father, left, right;

	public  Node(Comparable elem, Node father){
		this.elem = elem;
		this.father = father;
		left = null;
		right = null;
	}

}
```

# Implementazione

```
private class Node{

      Comparable key;
      Object     value;
      Node father, left, right;

      public  Node(Comparable elem, Node father){
            this.elem = elem;
            this.father = father;
            left = null;
            right = null;
      }

}
```

# Implementazione

```
private class Tree{

      private Node root = null;

      /* mothods implementation */
      ….

}
```

# Visite

- Stessi metodi implementati per l'albero binario

# Lookup di un elemento

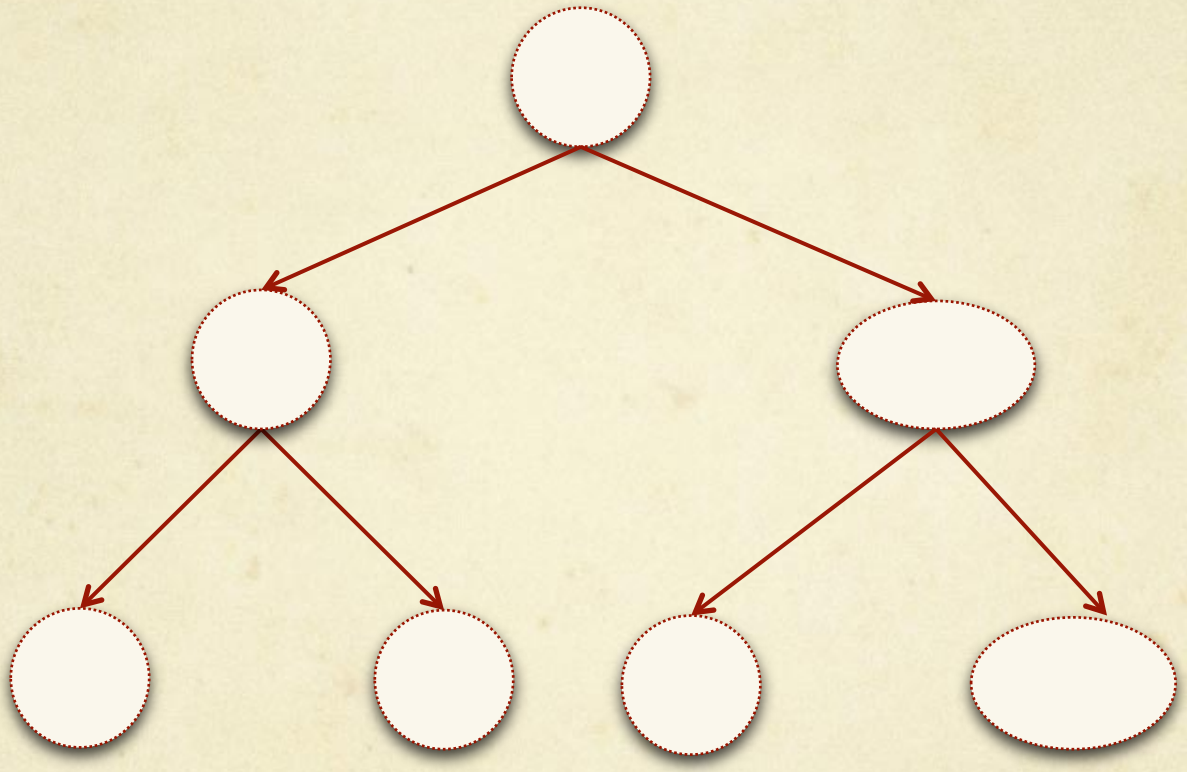
- La proprietà di ordinamento fa si che l'elemento si possa trovare solo in una e una sola posizione
- Partendo dalla radice si discende l'albero; ad ogni nodo si decide se proseguire la ricerca nel sottoalbero di destra o di sinistra

# Minimo e massimo

- Minimo: la foglia più a sinistra
- Massimo: la foglia più a destra

# Successore

- Successore: dato un nodo u è il nodo che contiene il più piccolo valore maggiore di u
- Due casi:
  - U ha il figlio destro
    - Successore: minimo del sottoalbero destro di u
  - U non ha il figlio destro
    - Successore: primo avo per il quale u si trova nel sottoalbero sinistro

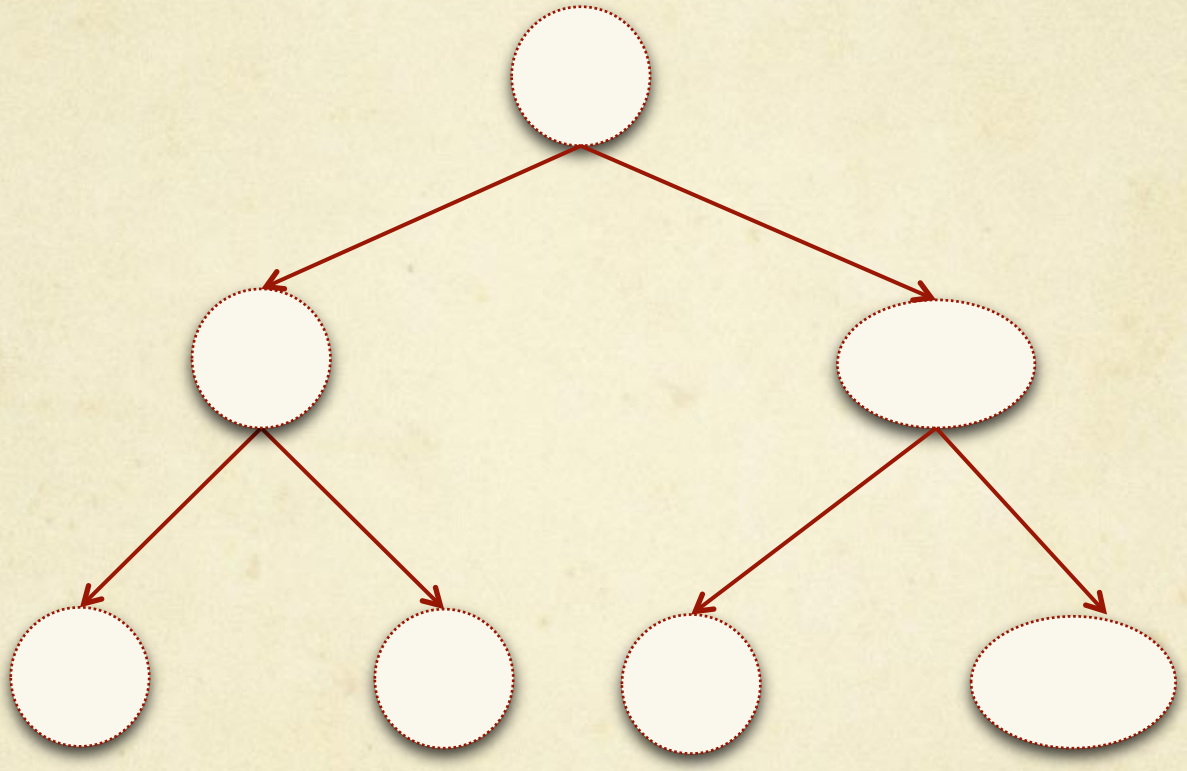


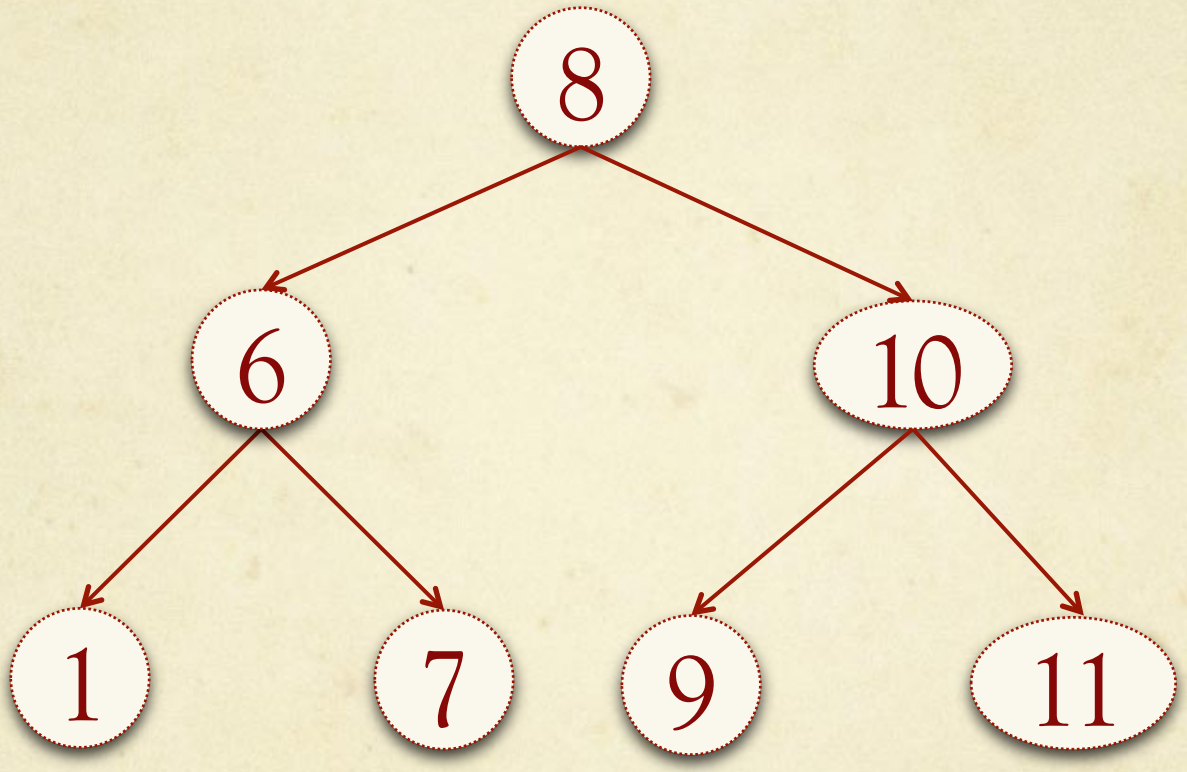
U ha il figlio destro:

Successore: minimo del sottoalbero destro di u

U non ha il figlio destro:

Successore: primo avo per il quale u si trova nel sottoalbero sinistro




8
6
10
1
7
9
11

# Inserimento

- Occorre rispettare le proprietà dell'albero binario di ricerca
- Ricerco la posizione nella quale devo inserire il nodo, ovvero ricerco quale dovrà essere il nodo padre
- Casi particolari:
  - se l'albero non contiene elementi il nuovo valore verrà inserito nel nodo radice
  - se l'albero contiene già un elemento con la stessa chiave, si modifica il nodo esistente sostituendone il valore

# Inserimento

- Caso generale:
  - Il nuovo nodo viene inserito come foglia

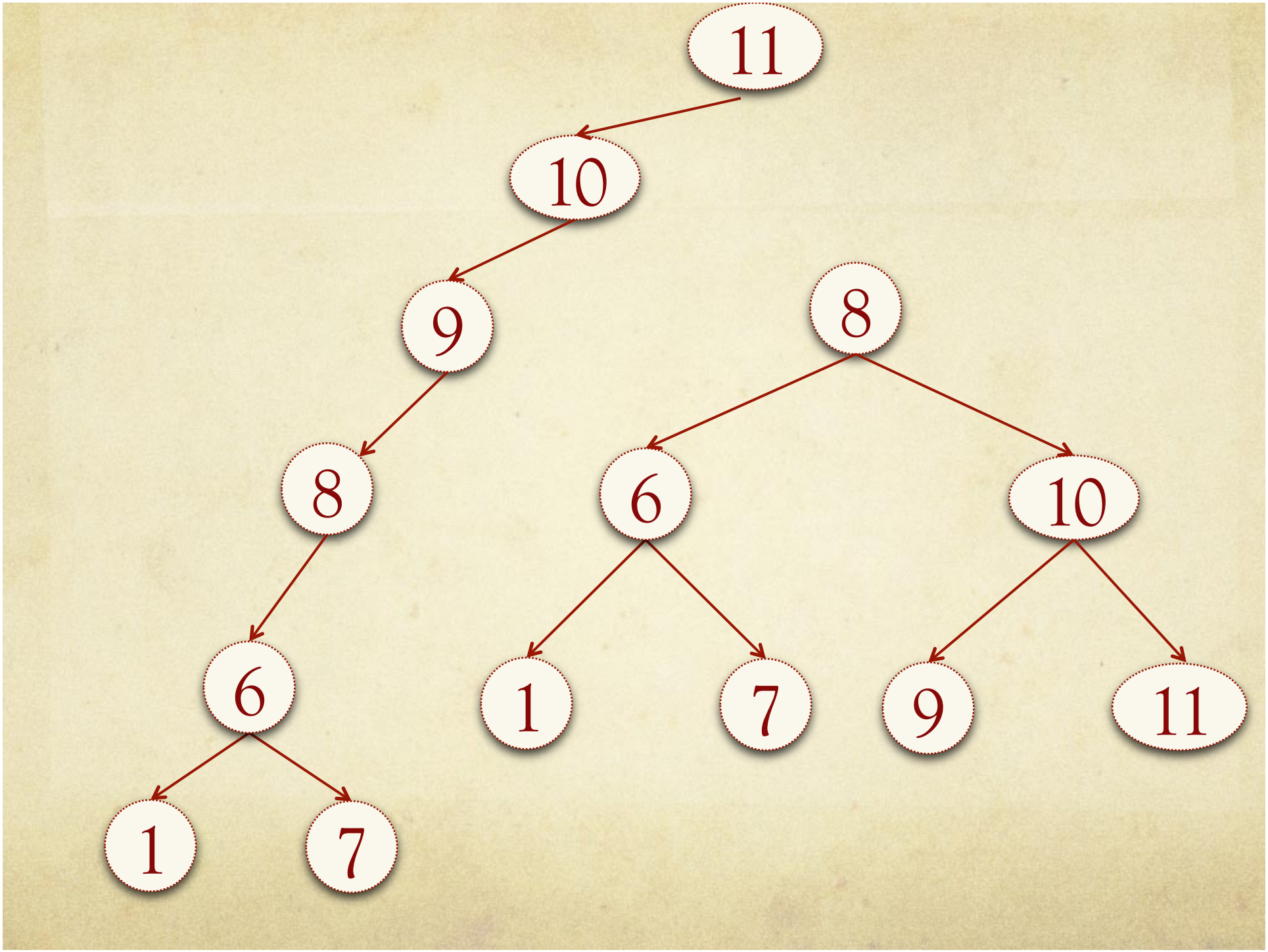
# Inserimento: implementazione

- Cerco la posizione dove inserire l'elemento
- Creo il nuovo nodo da inserire
- Gestisco i casi particolari
  - Albero vuoto
  - Elemento già presente
- Inserisco l'elemento nella posizione trovata

```
/**
 * @param elem : elemento da inserire
 */
public void insert(Comparable elem){

   // ricerca posisione dove inserire il nodo
   BSTNode tmp = root;          // usata per discendere l'albero
   BSTNode tmpFather = null;    // conterrà il padre del nodo da inserire
   while (tmp != null){
           tmpFather = tmp;
           // verifico se devo proseguire la visita nel sottolabero sinistro o destro
           if (elem.compareTo(tmp.getElem())<0){
                   tmp = tmp.getLeft();
           }else{
                   tmp = tmp.getRight();
           }
   }

   // creo il nodo da inserire
   BSTNode newNode = new BSTNode(elem,tmpFather);

   // gestisco caso particolare
   if (root == null){
          root = newNode;
   }else{
      // lo inserisco nella posizione trovata prima
      // ovvero lo inserisco o come figlio sinistro o destro di tmpFather
      if (elem.compareTo(tmpFather.getElem())<0){
              tmpFather.setLeft(newNode);
       }else{
              tmpFather.setRight(newNode);
       }
   }
}
```

# Esercizio

- Individuare l'ordine di inserimento degli elementi per ciascuno degli alberi mostrati nella slide successiva.


11
10
9
8
8
6
10
6
1
7
9
11
1
7

# Cancellazione

- Il nodo da eliminare potrebbe essere:
  - Una foglia
  - Avere un solo figlio
  - Avere due figli

# Cancellazione: caso 1

- Il nodo da eliminare è una foglia
  - Lo elimino

# Cancellazione: caso 2

- Il nodo da eliminare ha un solo figlio
  - Elimino il nodo u
  - L'unico figlio di u prende il posto di u (attacco il figlio di u al padre di u)

# Cancellazione: caso 3

- Il nodo da eliminare ha due figli
- Devo trovare un nodo da mettere al suo posto in modo tale che l'albero rimanga un albero di ricerca
  - Può essere il successore o il predecessore
  - Sostituisco u con il suo successore
    - Cerco il successore di u
    - Ne copio i valori nel nodo u
    - Lo cancello

# Cancellazione: implementazione

- Cerco il nodo da eliminare
- Se ha due figli (caso 3):
  - Cerco il successore (s)
  - Ne copio i valori all'interno del nodo da eliminare
  - Elimino il successore (caso 1 o 2)
- Se ha al più un figlio (caso 2 e 1)
  - Lo sostituisco a u
- Gestisco il caso particolare in cui il nodo da eliminare sia la radice

```
/**
 *
 * @param elem : elemento da cancellare
 */
    public void delete2(Comparable elem){

        // cerco il nodo da eliminare
        BSTNode u = searchRic(elem);

        if (u != null){
           // verifico se ha entrambi i figli (caso 3)
           if (u.getLeft()!= null && u.getRight()!=null){
               // cerco il sucessore
               BSTNode s = succ(u);
               // ne copio i valori all'interno di u
               u.setElem(s.getElem());
               // faccio diventare s il nodo da eliminare
               u=s;
           }
           // arrivati a questo punto u deve al più un figlio (casi 1 e 2)
           // controllo se ha il figlio destro o sinistro
           BSTNode t;
           if (u.getRight() != null){ // ha solo il figlio destro
               t = u.getRight();
           }else {
               t = u.getLeft();
           }

           // elimino u modificando i puntatori del padre di u
           // ma devo controllare che u non sia la radice
           if (u == root){
               root=t;
           }else{
               // l'eventuale figlio di u lo collego al padre
               // verifico se u era fisglio destro o sinistro
               if (u.getFather().getRight() == u){ // u è figlio destro
                   u.getFather().setRight(t);
               }else{                              // u è figlio sinistro
                   u.getFather().setLeft(t);
               }
           }
        }
    }
```

# Esercizio

- Dato un albero binario di ricerca implementare i seguenti metodi sia in modo ricorsivo che iterativo
  - Ricerca minimo e massimo
  - Ricerca di un elemento
  - Ricerca del successore e del predecessore
  - Inserimento
  - Cancellazione