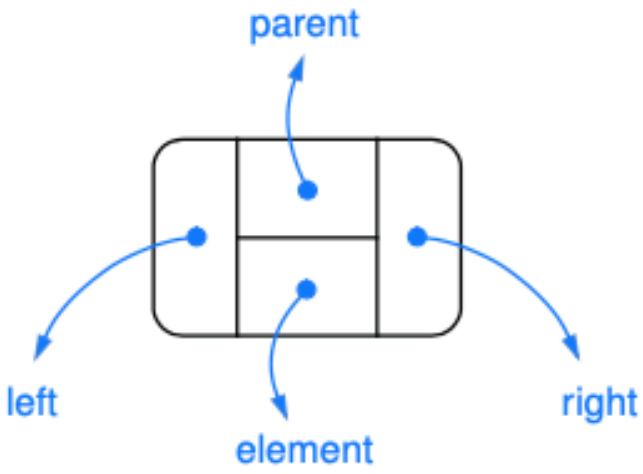
`LinkedBinaryTree`
Arbre binaire dans un tableau

Structure chaînée pour un arbre binaire
(`LinkedBinaryTree`)

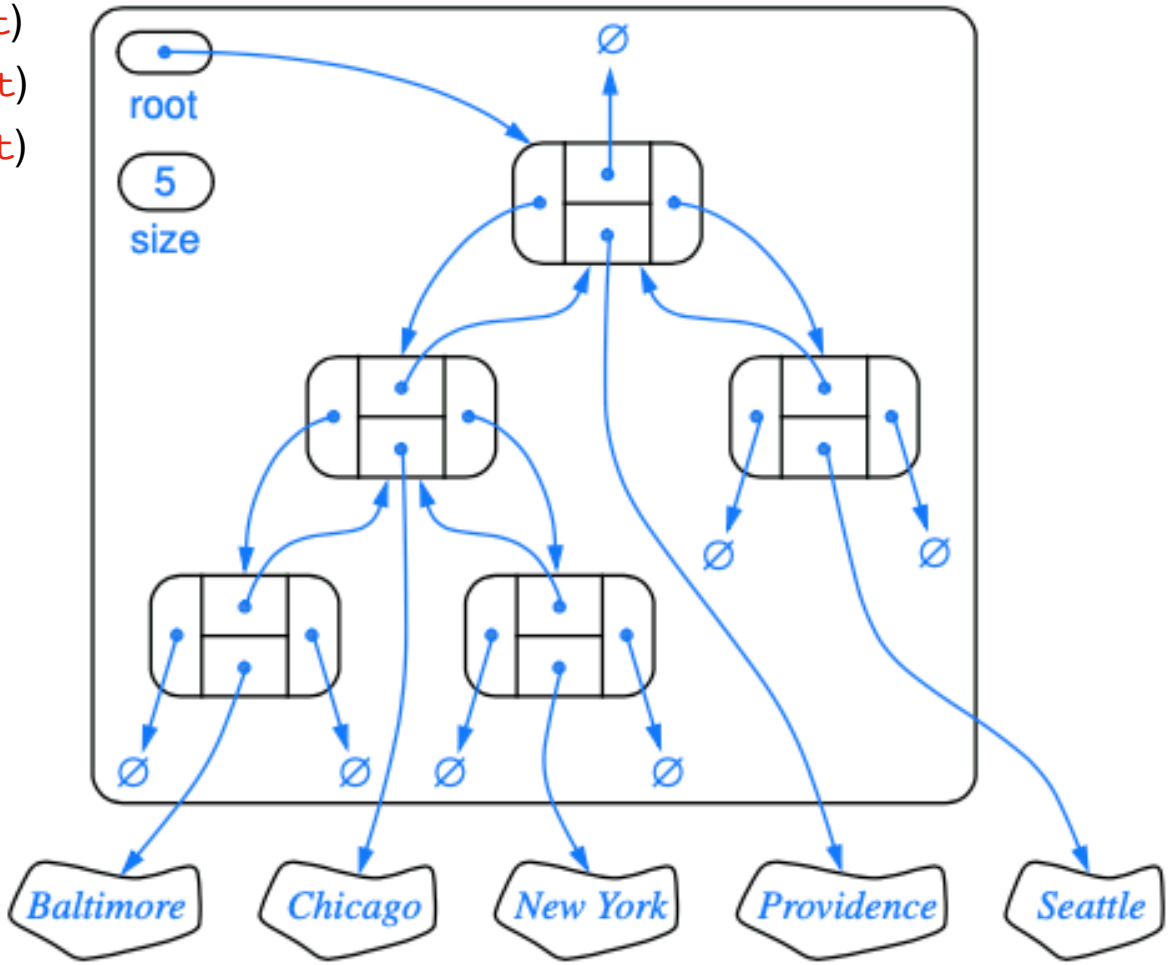
Un nœud est un objet contenant :

- une référence à un élément (`element`)
- une référence au nœud parent (`parent`)
- une référence au nœud à gauche (`left`)
- une référence au nœud à droite (`right`)



Un arbre binaire est un objet contenant :

- une référence à son nœud racine (`root`)
- un entier pour garder sa taille (`size`)



Opérations de mise à jour d'un `LinkedBinaryTree`

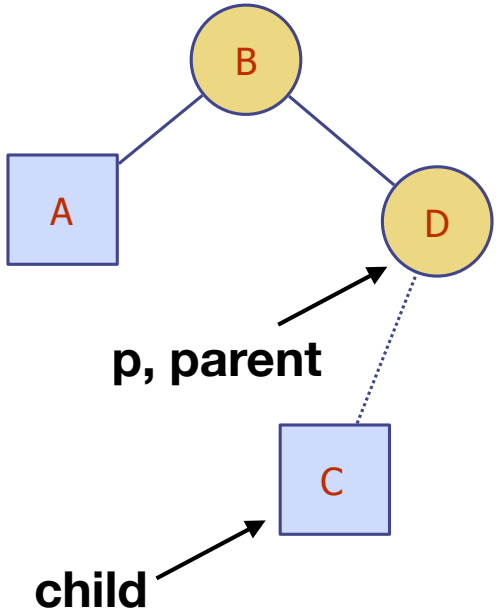
(toutes exécutent en $O(1)$ en pire cas)

`addRoot( e )`	Crée une racine pour un arbre vide, stocke `e` comme élément et renvoie la position de cette racine ; une erreur se produit si l'arbre n'est pas vide.
`addLeft( p, e )`	Crée un enfant gauche de la position `p`, stockant l'élément `e`, et renvoie la position du nouveau nœud ; une erreur se produit si `p` a déjà un enfant gauche.
`addRight( p, e )`	Crée un enfant droit de la position `p`, stockant l'élément `e`, et renvoie la position du nouveau nœud ; une erreur se produit si `p` a déjà un enfant droit.
`set( p, e )`	Remplace l'élément stocké à la position `p` par l'élément `e`, et renvoie l'élément précédemment stocké.
`attach( p, T1, T2 )`	Attache la structure interne des arbres `T1` et `T2` en tant que sous-arbres gauche et droit, respectivement, de la position de la feuille `p` et réinitialise `T1` et `T2` comme arbres vides ; une condition d'erreur se produit si `p` n'est pas une feuille.
`remove( p )`	Supprime la position `p`, la remplace par son enfant (le cas échéant) et renvoie l'élément qui était stocké à `p` ; une erreur se produit si `p` possède deux enfants.

Allons voir le code...

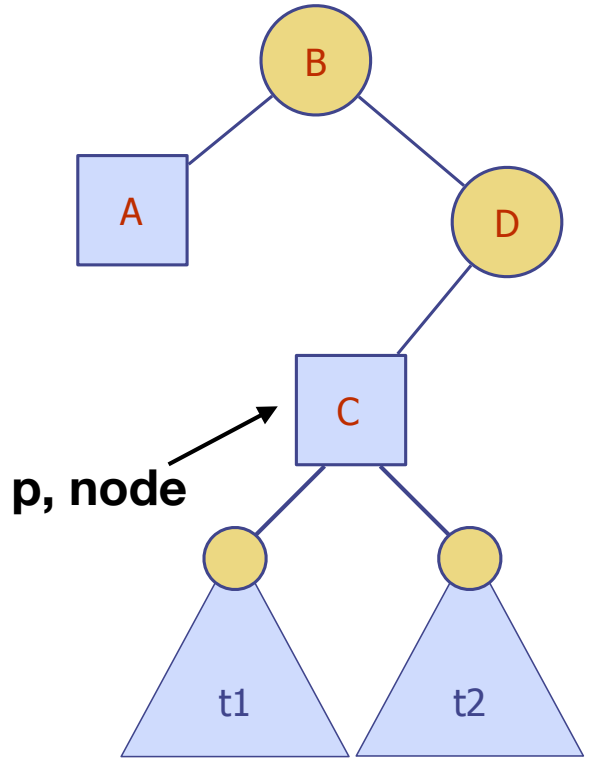
`LinkedBinaryTree.java`

```
// create a new left child of position p storing element e, return its position
 public Position<E> addLeft( Position<E> p, E e ) throws IllegalArgumentException {
     Node<E> parent = this.validate( p );
     if( parent.getLeft() != null )
         throw new IllegalArgumentException( "p already has a left child" );
     Node<E> child = createNode( e, parent, null, null );
     parent.setLeft( child );
     this.size++;
     return child;
 }
```



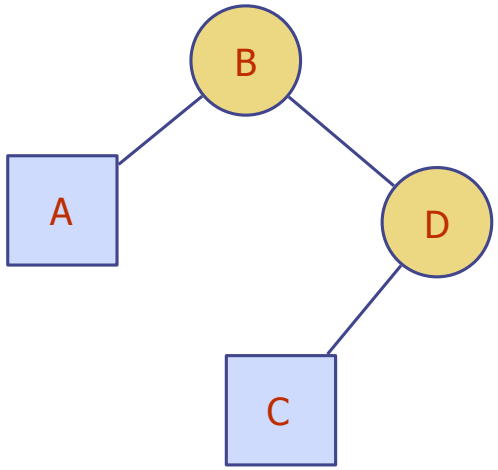
`addLeft( p, E = C )`

```
public void attach( Position<E> p, LinkedBinaryTree<E> t1, LinkedBinaryTree<E> t2 ) throws IllegalArgumentException {
    Node<E> node = this.validate( p );
    if( isInternal( p ) ) throw new IllegalArgumentException( "p must be a leaf" );
    this.size += t1.size() + t2.size();
    if( !t1.isEmpty() ) { // attach t1 as left subtree
        for( Position<E> pos: t1.positions() ) // change the container of the positions in t1
            setContainer( pos, this );
        t1.root.setParent( node );
        node.setLeft( t1.root );
        t1.root = null;
        t1.size = 0;
    }
    if( !t2.isEmpty() ) { // attach t2 as left subtree
        for( Position<E> pos: t2.positions() ) // change the container of the positions in t2
            setContainer( pos, this );
        t2.root.setParent( node );
        node.setRight( t2.root );
        t2.root = null;
        t2.size = 0;
    }
}
```



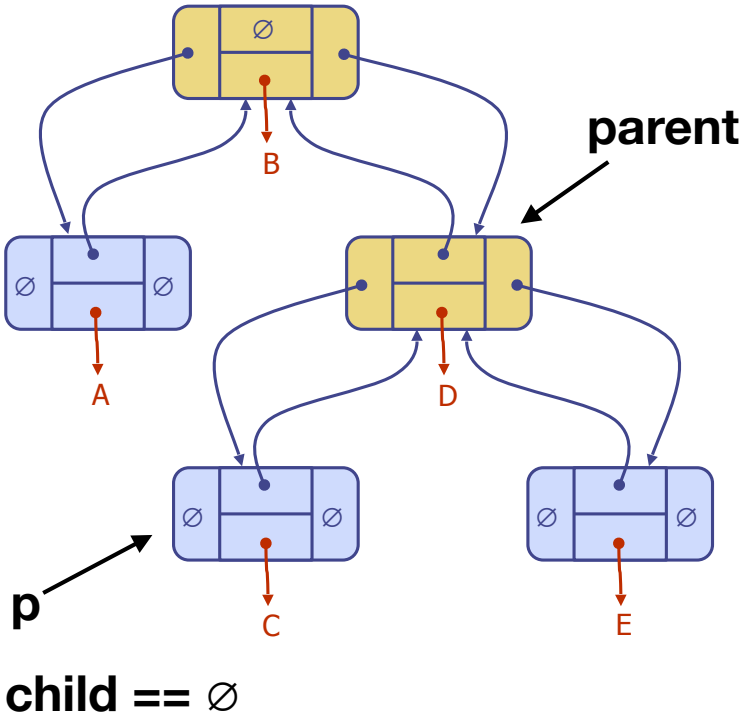
👉 attach fonctionne sur les feuilles uniquement

```java
// remove the node at position p, replace it with its child, if any
public E remove( Position<E> p ) throws IllegalArgumentException {
    Node<E> node = this.validate( p );
    if( numChildren( p ) == 2 )
        throw new IllegalArgumentException( "p has two children" );
    Node<E> child = ( node.getLeft() != null ? node.getLeft() : node.getRight() );
    if( child != null )
        child.setParent( node.getParent() ); // child's grandparent becomes parent
    if( node == root )
        this.root = child; // child becomes root
    else {
        Node<E> parent = node.getParent();
        if( node == parent.getLeft() )
            parent.setLeft( child );
        else
            parent.setRight( child );
    }
    this.size--;
    E tmp = node.getElement();
    node.setElement( null ); // garbage collection
    node.setLeft( null );
    node.setRight( null );
    node.setParent( node ); // convention for defunct node
    return tmp;
}
```

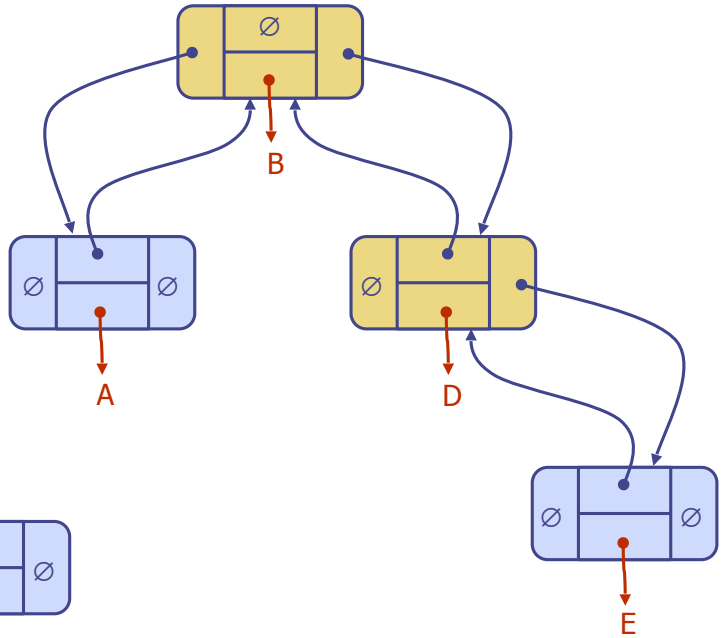


👉 remove fonctionne sur les noeuds qui possèdent au plus 1 enfant (A, D et C)

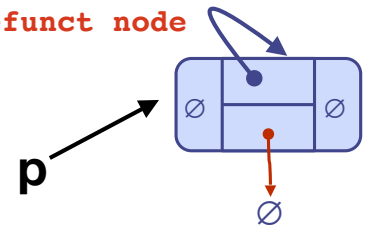
remove(feuille)



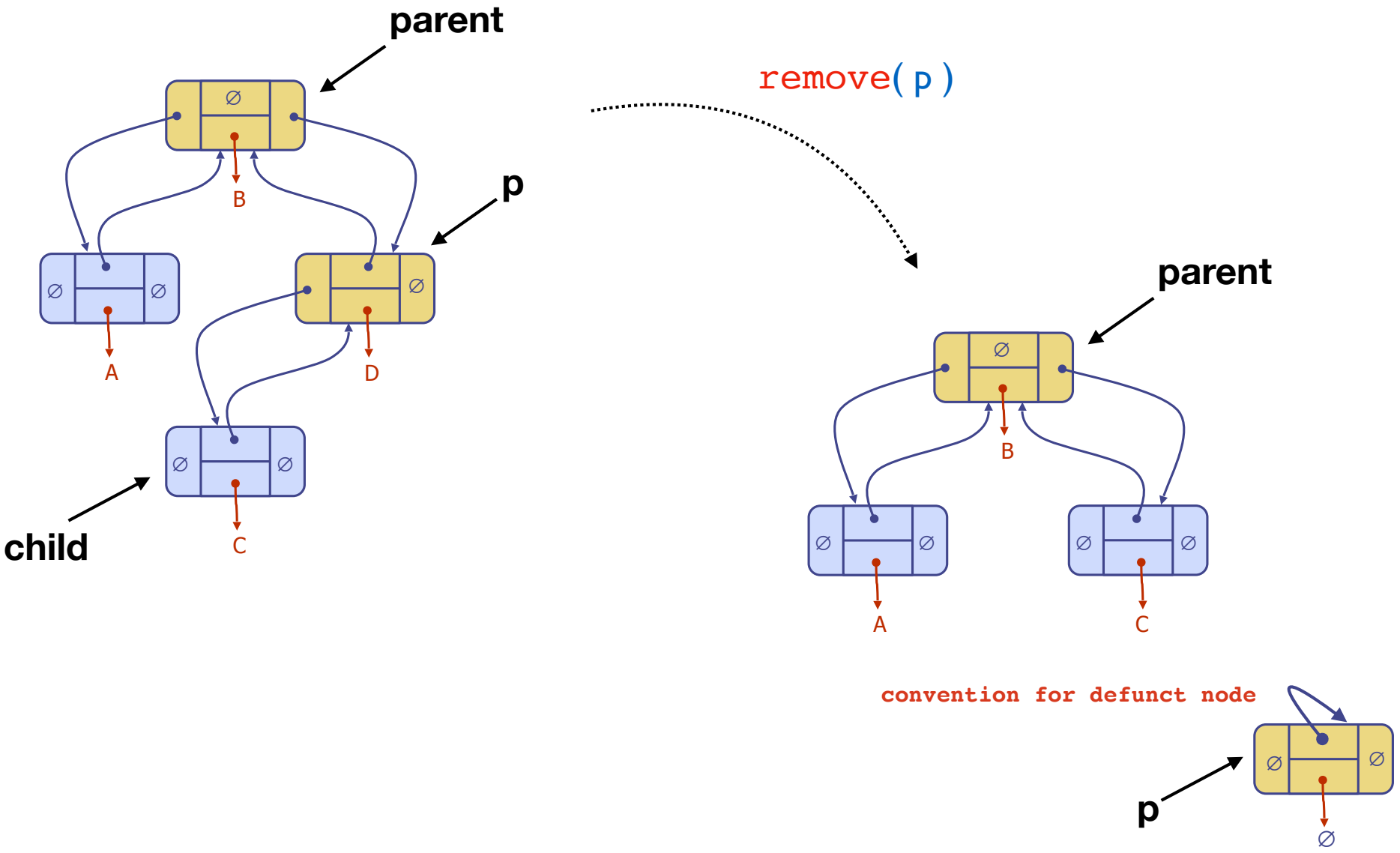
remove(p)



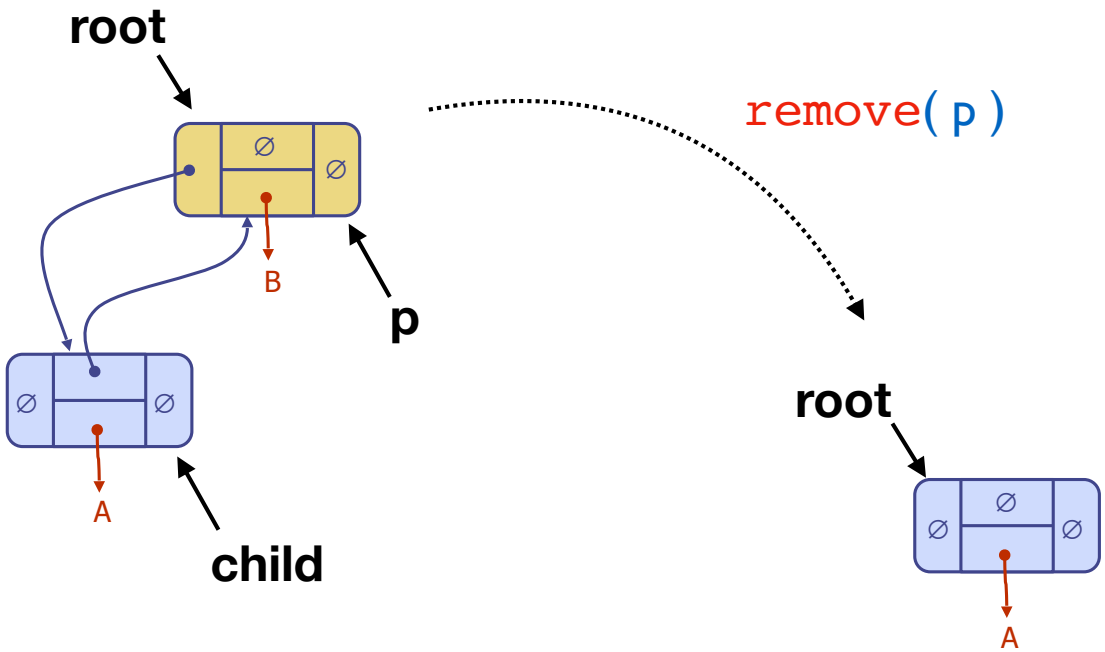
convention for defunct node



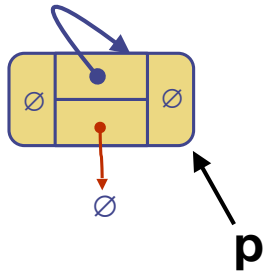
remove(monoparent)



remove(racine)



convention for defunct node



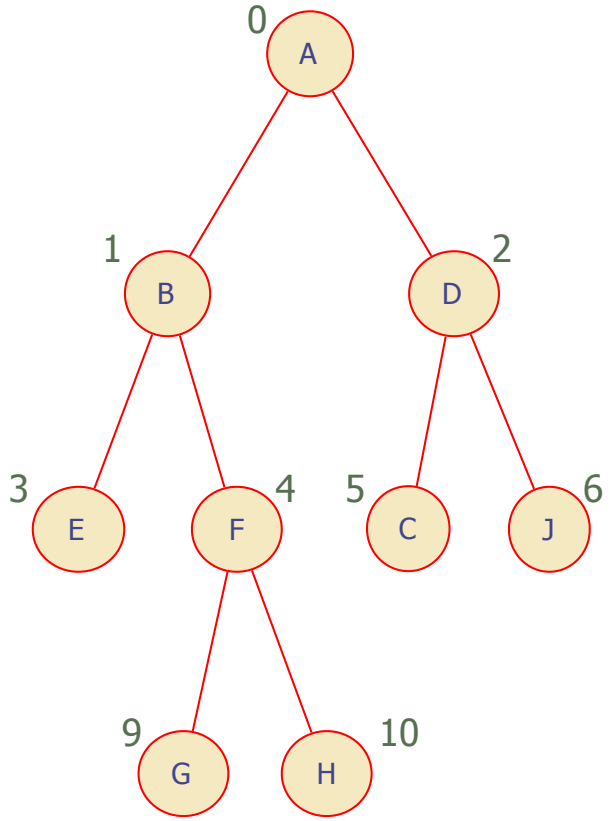
Représentation d'un arbre binaire dans un tableau

Les nœuds sont stockés dans un tableau

Pour toute position `p` de `T` , soit **f(p)** l'entier défini comme suit :

- Si `p` est la racine de `T`, alors **f(p) = 0**
- Si `p` est l'enfant gauche de `q`, alors **f(p) = 2 f(q) + 1**
- Si `p` est l'enfant droit `q`, alors **f(p) = 2 f(q) + 2**

```
getLeft( p ) = 2 * p + 1;
getRight( p ) = 2 * p + 2;
getParent( p ) = ⌊( p - 1 ) / 2⌋
```



Conclusions du module

- Nous avons regardé une structure chaînée pour des arbres binaires
- Nous avons défini les opérations de mises à jour des arbres binaires
- Nous avons implémenter `LinkedBinaryTree`
- Nous avons vu où et comment changer le container des noeuds d'un arbre qui se fait fusionner à un autre arbre
- Nous avons vu la méthode `remove` mais seulement pour noeuds qui possède au plus 1 enfant
- Nous avons regardé une représentation dans un tableau pour des arbres binaires