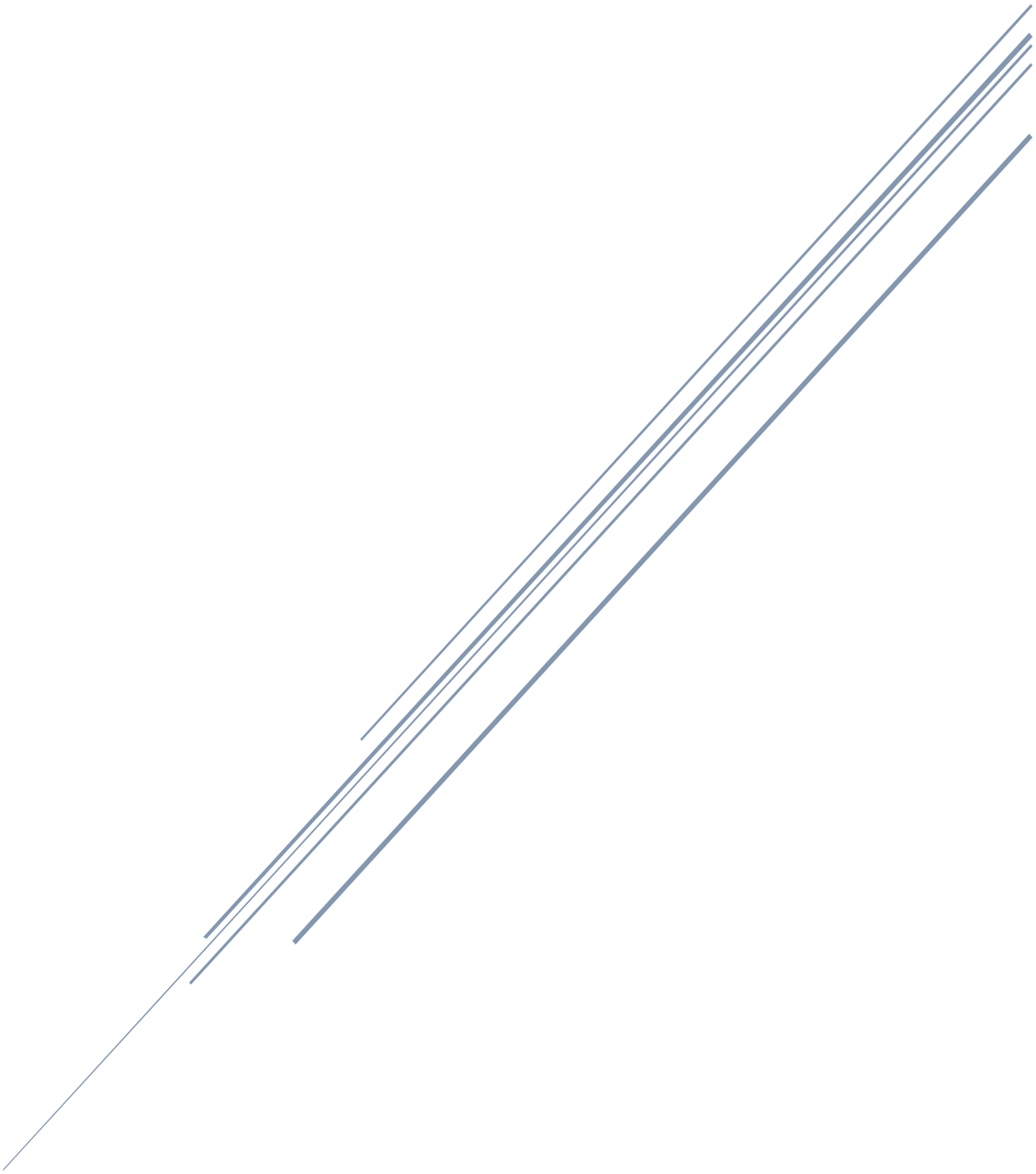
LABORATORI 4

Arbres Binaris i Recorreguts



Autors: Èric Bitriá i Adrià Fernández

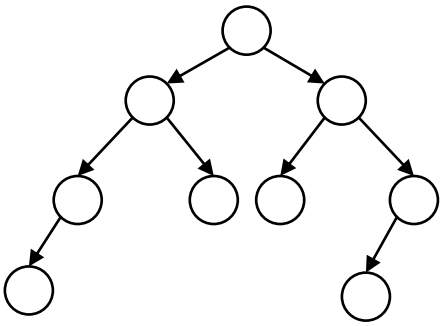
Índex

1. Introducció

Un arbre binari es un tipus d'estructura de dades en format d'arbre amb la característica que cada node del arbre podrà tenir com a màxim dos fills. En aquesta pràctica s'acabarà d'implementar la classe *LinkedBinaryTree*, la qual segueix les especificacions de la interfície *BinaryTree* i contindrà elements genèrics.

Cada element que formarà part del arbre [1] seran nodes enllaçats, on cadascun contindrà un total de 5 paràmetres com veurem al llarg de tota la pràctica:

- **Node <E> left:** Referència al node fill esquerre.
- **Node <E> rightOrNext:** Referència al node fill dret o referència al següent node.
- **E element:** La informació o dada que contindrà el node.
- **int size:** Representarà la mida d'aquell node.
- **boolean isRightChild:** Element booleà que determinarà si el node té fill dret o no.



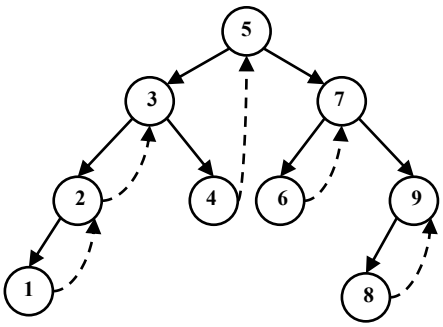
[1] Arbre Binari

Com hem vist en els paràmetres de cada node tenim una implementació lleugerament diferent a l'habitual, la referència *rightOrNext* podrà ser la del fill dret o al següent node en un recorregut. L'avantatge d'afegir una referència al següent node en un recorregut és que aprofitem una referència que seria *null* per facilitar la implementació posterior al recorregut in-ordre del arbre.

Un recorregut en in-ordre és un mètode per recórrer els nodes d'un arbre binari. En aquest recorregut, es visiten els nodes de l'arbre en la següent seqüència:

- Visiteu el subarbre esquerre.
- Visiteu el node actual.
- Visiteu el subarbre dret.

Si afegim les referències del següent node a visitar, en comptes d'una referència null en el fill dret, aquest recorregut és simplifica i en la majoria dels casos haurem a aquest[2]:

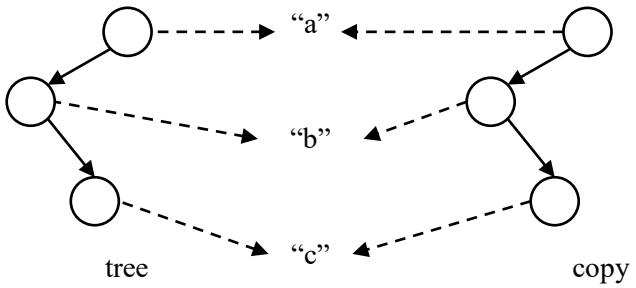


[2] Recorregut in-order d'un arbre binari, on cada nombre del node representa l'ordre del recorregut

2. Nou Constructor: Copy

La primera tasca a realitzar serà la implementació d'un constructor que realitzarà una copia del arbre enviat per paràmetre. Aquesta copia tindrà una característica especial, doncs no copiarà els elements del arbre sinó que mantindrà una referència d'aquests a efectes que si es modifica l'element d'un arbre la copia d'aquest arbre també quedarà afectada.

Podem representar com quedaria un arbre i la seva copia en aquest diagrama [3]:



[3] Visualització d'un arbre i la seva copia compartint referències al mateix element

La crida del constructor rebrà per paràmetre el arbre que volem copiar, aquí podem fer les primeres comprovacions on si l'arbre és null retornarem null i en cas contrari podrem començar a copiar els nodes.

Els constructors en aquesta classe han d'inicialitzar l'arrel del arbre, pel que necessitarem cridar una funció auxiliar enviant l'arrel:

```
public LinkedBinaryTree(LinkedBinaryTree<E> tree) {
    root = tree == null ? null : Node.copy(tree.root);
}
```

Aquesta funció es situarà dins de la classe interna Node i realitzarà l'acció de copiar cada node del arbre de forma recursiva. És possible que algun node que intentem copiar pugui ser null, per això la primera comprovació haurà de ser aquesta:

```
if(node == null) return null;
```

Ara que no ens hem de preocupar si el node a tractar és null, haurem de crear un nou node mantenint la referència al element del node original, i creant copies dels nodes dret i esquerre. Amb la recursivitat ho tenim molt fàcil, dins del propi constructor del node podem fer una crida recursiva sobre els fills drets i esquerres del node original, el qual rebem per paràmetre.

```
return new Node<E>(
    copy(node.left),
    node.element,
    copy(node.rightOrNext)
);
```

Ara la nostra funció ja es capaç de recórrer tot el arbre. Fixem-nos que al construir el nou node li enviem *node.element*, això ens permet enviar la referència del element del node original.

En aquesta implementació les referències dels elements del arbre es veuran afectades entre copia i original solament si els elements que contenen son mutables, s'entrarà més en detall sobre aquesta característica a l'apartat dels tests ja que es podrà veure millor amb els exemples i tests.

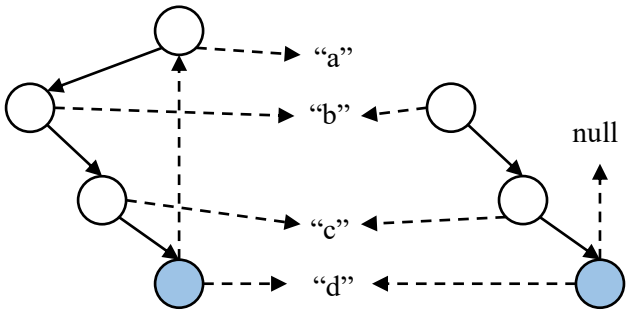
Requerirem d'un altra modificació en relació al apartat 2 on atribuïm referències en InOrdre. Per tant haurem de diferenciar si la referència és el següent node en in ordre o el fill dret, sinó tindríem un problema on la funció copy entraria en un bucle infinit. Disposem de la funció right() que retorna en funció del booleà del Node el fill dret o null.

```
Node<E> right() { return isRightChild ? rightOrNext : null; }
```

Finalment tenim completada la funció copy():

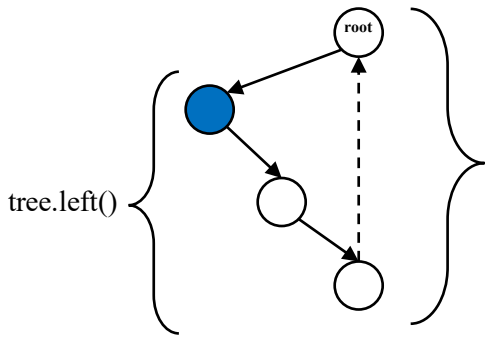
```
static <E> Node<E> copy(Node<E> node){
    if(node == null) return null;
    return new Node<E>(
        copy(node.left),
        node.element,
        copy(node.copy())
    );
}
```

Com veurem més endavant en el constructor en InOrdre no ens haurem de preocupar quan fem la copia d'un subarbre i alguna referència tindria que apuntar a algun node del arbre original [4].

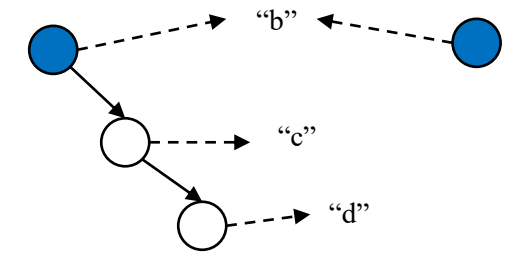


[4] Diagrama de dos arbres binaris, on l'arbre de la dreta es la copia del fill esquerre del arbre esquerre

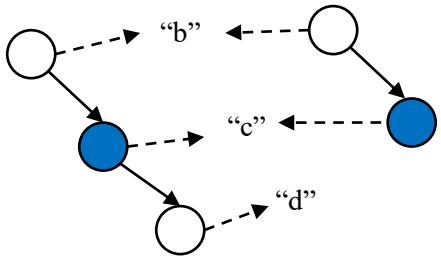
Per acabar d'il·lustrar com la funció copy() es mostren un seguit de diagrames sobre com interactua aquesta funció en el arbre de la figura [4] des de la crida al constructor fins l'execució final:



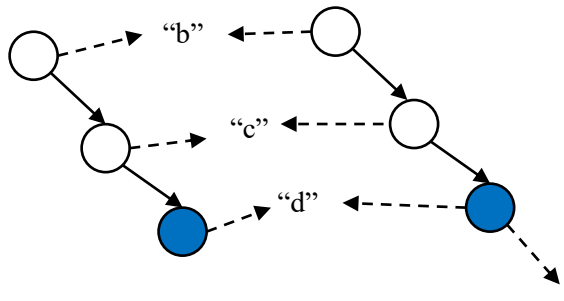

```
var newCopy =
 new LinkedBinaryTree(tree.left())
```




```
return new Node<E>(
    copy(node.left), //null
    node.element,
    copy(node.right())//Call
```




```
return new Node<E>(
    copy(node.left), //null
    node.element,
    copy(node.right())//Call
```




```
return new Node<E>(
    copy(node.left), //null
    node.element,
    copy(node.right())//null
```

[5] Construcció del arbre copia amb la funció copy,

3. Enfilament simple en in-ordre

Com s'ha mencionat prèviament a la introducció, cada node del arbre binari que no disposi d'un fill dret podrà tenir una referència al següent node seguint un recorregut en in-ordre per així facilitar la implementació del iterador. Aquest apartat es centra en com s'ha aconseguit implementar tal funcionalitat explicant quins passos o consideracions s'han anat tenint al llarg del plantejament.

Necessitem modificar el constructor de la classe Node per afegir aquesta funcionalitat i el primer pas serà definir quan afegirem una referència al següent element en in-ordre o una referència al fill dret:

```
boolean isRightChild;
```

La comprovació és molt senzilla, si el fill dret és null no hi ha fill dret. Amb aquesta condició haurem de modificar el valor del booleà isRightChild i així diferenciarem entre nodes:

```
this.isRightChild = right != null;
```

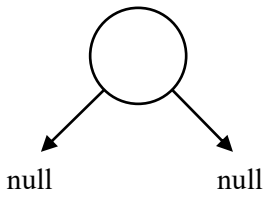
3.1 Definint els casos possibles

Com en moltes implementacions i problemes que s'han vist en aquesta assignatura la forma més adequada d'abordar un problema es veient que volem aconseguir i classificar aquest objectius en uns de més concrets i senzills. D'aquesta manera podrem crear petites implementacions de un en una i avançar en el problema.

Així doncs, quins casos possibles tenim? Comencem pels més senzills.

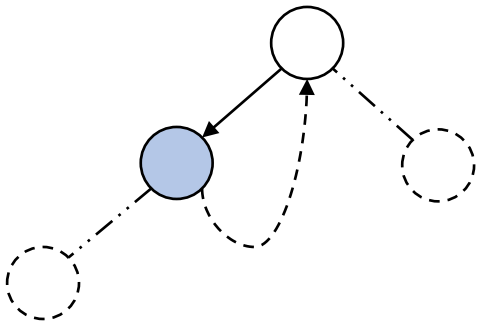
3.1.1 Casos simples

No hi ha cas més senzill que quan el constructor solament rep l'arrel sense fills, evidentment el node no podrà fer referència a cap fill. [6]:



[6] Arbre binari solament amb l'arrel

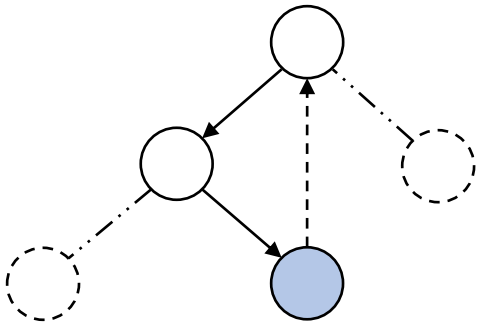
Un altre cas és quan un node té un fill esquerre, i aquest fill, en cas de no tenir fill dret, sempre farà referència al seu pare [7]:



[7] Representació d'arbre on el fill esquerre, en cas de no tenir fill dret sempre fa referència al seu pare

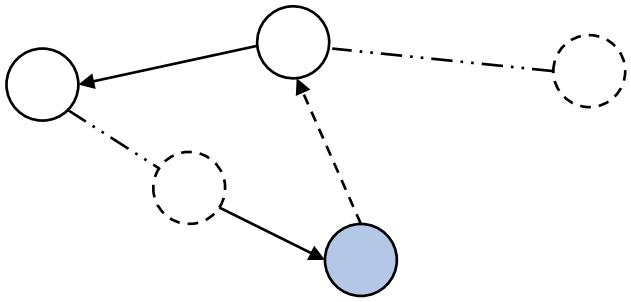
3.1.2 Cas no tant simple

Donat una estructura d'arbre binari d'aquest estil:



[8] Representació d'arbre binari, on el node blau es fill dret d'un altre node, i la seva següent referència és el node pare del anterior

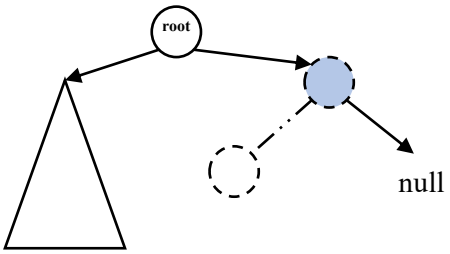
Com podem veure en el diagrama [8] un node intern del arbre que sigui fill dret d'un altre node, la seva següent referència (en cas de no tenir fill dret) apuntarà com a mínim a dos nivells superiors. És a dir apuntarà a l'últim node que tenia un fill esquerre del qual deriva aquest fill dret.



[9] Diagrama generalitzat del diagrama [8], on els nodes rallats representen indefinits nombre de nodes que poden existir entre cada extrem.

3.1.3 Cas null

Un aspecte interessant d'aquest cas es que el node de més a la dreta del arbre sempre serà l'últim node en el recorregut i la seva següent referència tindrà que ser null [10].



[10] Representació d'arbre binari, on el node situat més a la dreta ha de fer referència a null

Realment aquest cas es deriva del apartat 3.1.2 amb la característica que el node no pot apuntar al pare que te un fill esquerre, ja que no existeix.

3.2 Primera Aproximació: setInOrder

Definirem una funció auxiliar que la seva funcionalitat únicament serà afegir les referències dels nodes nous que haguem creat. Com que mantindríem un paràmetre per enviar tot el node i un altre per altres valors, com els nodes fills, la funció setInOrder serà de la forma:

```
private void setInOrder(Node<E> current, Node<E> parent){...}
```

I la crida d'aquesta funció dins del constructor Node:

```
setInOrder(this,null);
```

Com que el node que estem construint no tindrà pare aquest paràmetre inicialment tindrà que ser null, i el node que estem tractant serà *this*.

Ara que ja tenim la definició de la funció que crearà les referències hem de idear com ho farà. Tenint en compte el funcionament del recorregut en in-ordre i els casos que hem de tractar es pot arribar a una solució inicial.

Primer considerem com busquem un recorregut en in-ordre. Hem de moure'ns tot a la esquerra possible del arbre que estiguem tractant. Això ho podem aconseguir de forma molt senzilla amb una crida recursiva buscant, sempre i quant existeixi, el fill esquerre.

```
if(current.left != null) {
    setInOrder(current.left,current)
}{...}
```

Recordem també el cas simple 3.1.1 [7] on tot node que no tingui fill dret farà referència al seu pare, per aquest motiu en cas que es donés aquesta situació haurem d'enviar la referència al seu pare que en aquest cas és el node que estem tractant *current*.

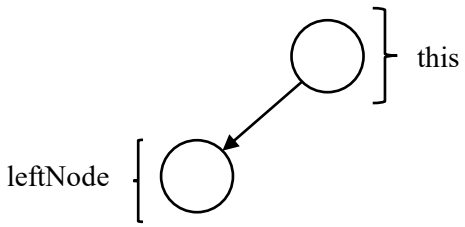
En cas que haguem arribat a un node sense fill esquerre haurem acabat el recorregut i ens haurem de preguntar si aquest node té fill dret o no. Centrem-nos de moment en el cas més senzill, si el node no té fill dret es compleix el cas 3.1.1 [7] i per tant la seva referència del fill dret anirà al seu pare, com que a la nostra funció enviem també el pare del node ho tenim molt fàcil:

```
else if(!current.isRightChild){
    current.rightOrNext = parent;
}
```

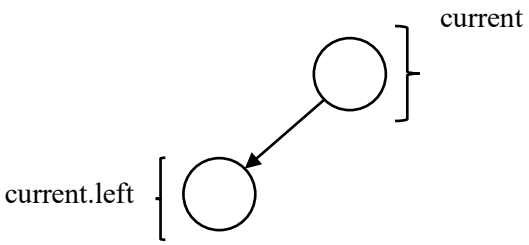
Cada node té un valor booleà que ens indica si el seu node dret és un fill dret o el següent, simplement comprovem que no es compleixi i assignem la referència.

En les següents imatges es mostra l'execució des de la crida al constructor del node a l'assignació de la referència del node esquerre [10].

```
Node(leftNode,element,null)
```



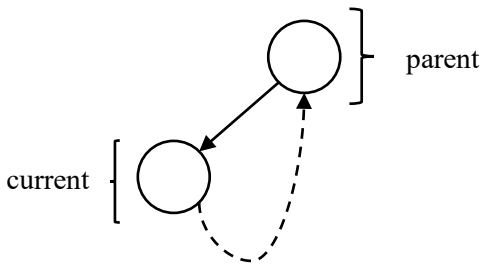
```
setInOrder(this,null);
```




```
    if(current.left != null){     //TRUE
        setInOrder(current.left,current)
    }

setInOrder(current.left,current);

    if(current.left != null){              //FALSE
    } else if(!current.isRightChild){    //TRUE
        current.rightOrNext = parent;
    }
```



[11] Procés d'assignació de referències en in-ordre en el primer cas

Ara ens queda considerar quan el node que estem tractant si que té fill dret:

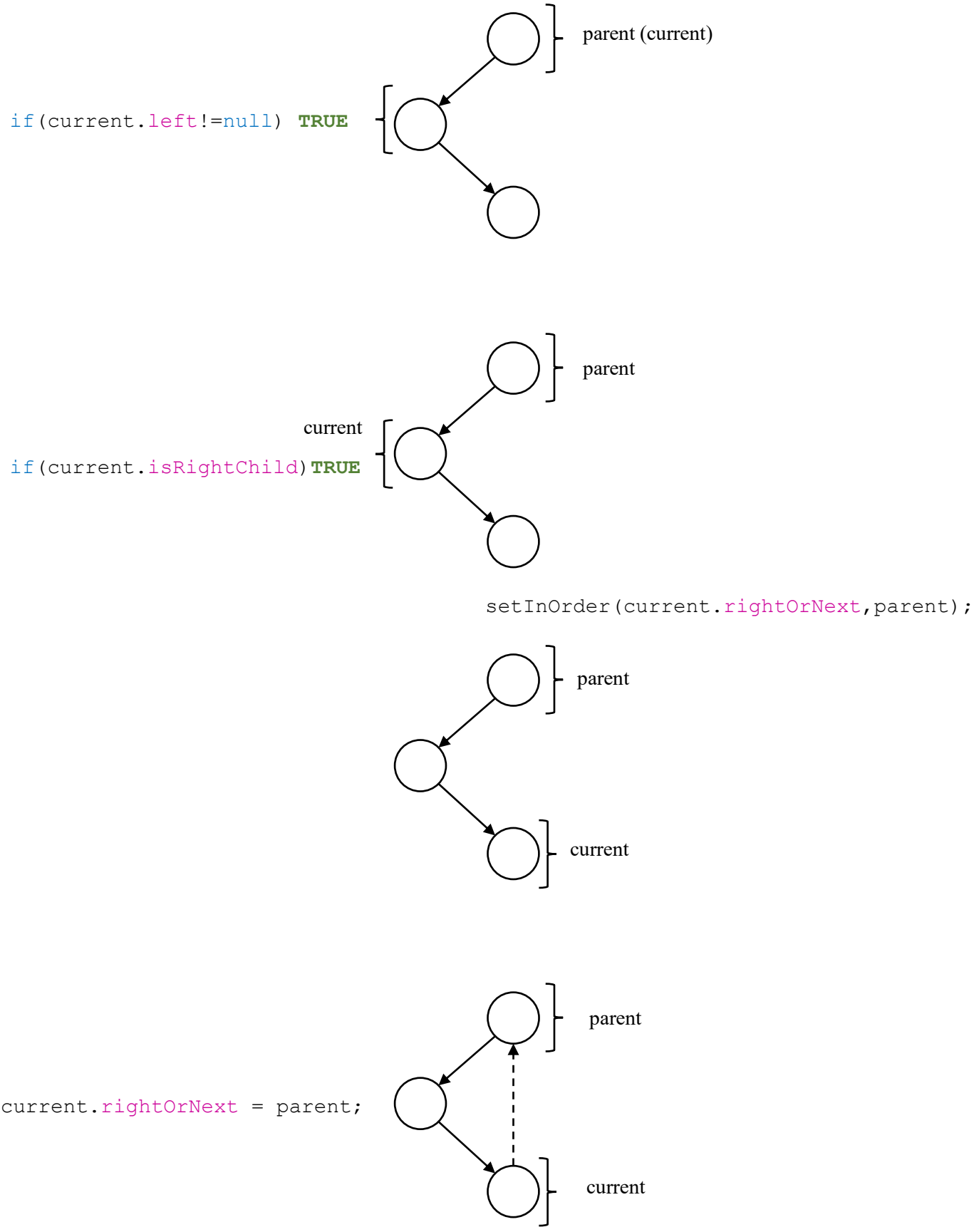
```
} if (current.isRightChild) {
    ???
}
```

Recordem que quan hem d'accedir a un node dret i assignar-li una referència, aquesta referència no anirà al seu node pare, anirà a l'últim node que ha tingut un fill esquerre i del qual ha derivat un o distints fills drets, veure apartat 3.2.1.

Però es pot donar la situació que aquest fill dret tingui un fill esquerre i per tant aquest tindrà prioritat, per això haurem de tornar a fer una crida a la funció per moure'ns per aquest fill.

```
setInOrder(current.rightOrNext,parent);
```

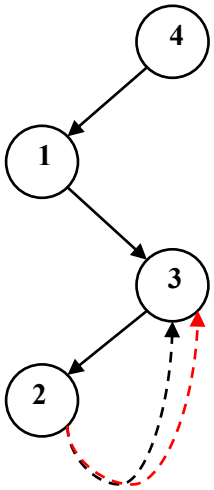
Ara ja no enviem *current* com a pare d'aquest node, ja que la referència serà mínim dos nivells més.



[12] Procés d'assignació de referència en el segon cas

Aquesta implementació a primera vista pot funcionar, però té un gran problema, i és en les crides recursives. Quan aquesta funció construeix un arbre amb molts pocs nodes pot funcionar com és d'esperar, el problema ve quan volem construir un arbre amb milers de nodes.

Això es degut que aquesta funció recorre el arbre sempre que pot per la esquerra, no es queda solament amb el node nou a construir. D'aquest problema de recorregut en deriva un altre, fiquem per exemple [13]:



[13] Problema de la primera implementació de setInOrder

Quan la funció recorri l'arbre es quedarà en l'últim node i assignarà una referència al seu pare, referència que ja havia set atribuïda en el moment de construir el node 3. Això es degut que donat un node, la nostra funció sempre buscarà el node fill més a la esquerra d'aquest i sobreescriurà referències que ja estaven assignades, a més de fer un recorregut innecessari i no assignant la referència, en aquest cas, del node 3 al 4.

3.3 Resultat final: SetInOrder

Ara que tenim una funció que més o menys ens fa la feina anem a millorar-la per acabar de complir el seu objectiu.

Recordem per última vegada què volem fer quan construïm un node. Si aquest node té un fill esquerre volem que aquest, a ser possible, tingui una referència al nou node (3.1.1). Si el node nou té un fill dret no haurem de fer res, ja que no hi haurà nodes del fill dret que apuntin mai a aquest (3.1.3). Per tant primer comprovem si el node que construïm té fill esquerre o no.

```
if(left != null) {
    setInOrder(left,this);
```

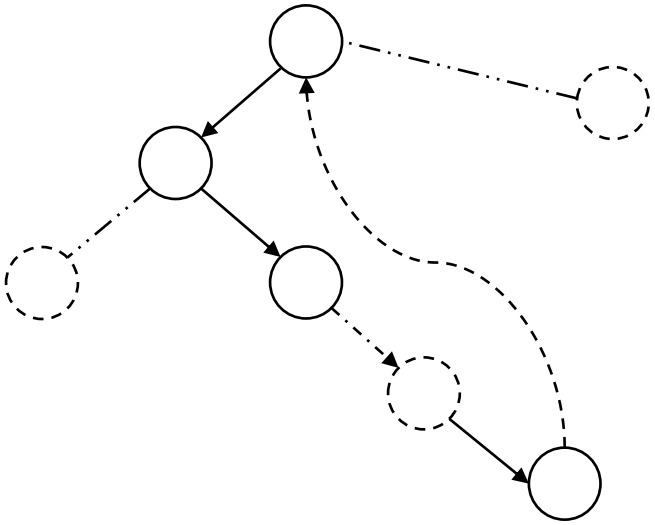
El node que volem que tingui una referència serà *left* i la referència serà el node creat, *this*. Dins de la funció comprovarem si el node que estem tractant *current* té fill dret o no, si no en té, molt senzill assignem el rightOrNext al pare [11 (Última imatge)]:

```
} else {
    current.rightOrNext = parent;
}
```

Si el node té fill dret es complica una mica més.

```
if(current.isRightChild){
    ???
}
```

En aquest cas ens hauríem de situar en aquest node dret per fer-nos la mateixa pregunta, té fill dret o podem assignar-li una referència? Podem veure aquesta situació en el següent diagrama [14]:



[14] Assignació de referències a nodes superiors

Al final és com s'ha mencionat prèviament, buscarem el node respecte l'últim node amb un fill esquerre que estigui més a la dreta per poder assignar-li la referència. És per aquest motiu que no cal preocupar-nos per els nodes dret de l'arrel ja que la referència serà *null* al no existir un node superior que compleixi això.

En conclusió sempre ens hem de moure cap a la dreta d'aquest últim node, enviant també, la referència del node següent:

```
setInOrder(current.rightOrNext, parent);
```

Així doncs, la funció final es aquesta:

```
private void setInOrder(Node<E> current, Node<E> parent){
    if(current.isRightChild){
        setInOrder(current.rightOrNext, parent);
    } else {
        current.rightOrNext = parent;
    }
}
```

Podíem arreglar la funció anterior si s'afegia un condicional que propagues les referències entre tots els nodes però si analitzem el cost d'aquesta funció respecte la primera implementació, ja no recorrem tots els fills esquerres i solament es converteix lineal en funció del nombre de fills drets. Per tant en cas que haguem de referenciar un node superior [14], que és el cas recursiu, això té un cost lineal i en el cas de referenciar el fill esquerre és constant.

En conclusió, la primera versió ens hauria servit per convertir un arbre complet sense referències a priori en referències en in-ordre, però en aquesta implementació busquem fer-ho a nivell del constructor de nodes.

4. InOrderIterator

El *InOrderIterator* serà una classe privada dins del *LinkedBinaryTree* que tindrà una sèrie de mètodes per iterar dins del arbre binari seguint un recorregut en in-ordre.

4.1 Constructor

Per poder iterar per dins del arbre tindrem que tenir dues variables que guardaran els nodes del recorregut:

```
private Node<E> next;
private Node<E> lastReturned;
```

Els noms s'expliquen per si mateixos, *next* serà el següent node en in-ordre i *lastReturned* serà el últim node que s'ha retornat del *next*.

El constructor haurà d'inicialitzar aquestes variables, com que acabem de inicialitzar el iterador, el *lastReturned* tindrà que ser *null*. I amb el next tindrem que fer una crida a una funció auxiliar per situar-nos al primer node en in-ordre.

```
public InOrderIterator(){
    lastReturned = null;
    next = leftmostNode(LinkedBinaryTree.this.root);
}
```

4.2 Set()

La funció set modificarà l'element del últim node retornat per el que li enviem per paràmetre. Primer haurem de comprovar que l'últim element retornat un node, és a dir que no sigui null. I si aquest no es el cas podem modificar l'element.

```
public void set(E e){
    if(lastReturned == null){
        throw new IllegalStateException("...");
    }
    lastReturned.element = e;
```

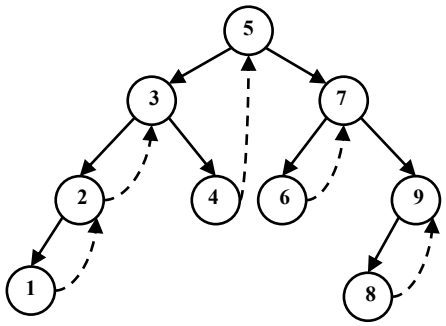
4.3 hasNext()

Es un booleà que indica si existeix un següent node en la iteració. Comprovem si *next* es diferent de *null*.

```
public boolean hasNext(){
    return next != null;
```

4.4 next()

El next() és la funció principal del iterador i retorna l'element del següent node (en cas d'existir) seguint un recorregut en in-ordre. Recordem que un recorregut en in-ordre es basa en moure'ns fins al node de més a la esquerra sempre que sigui possible i a continuació anirem ascendint i visitant els fills drets [15].



[15] Recorregut in-order d'un arbre binari, on cada nombre del node representa l'ordre del recorregut

La nostra forma d'implementar-ho juga molt amb com inicialitzem el constructor del iterador.

4.4.1 leftmostNode()

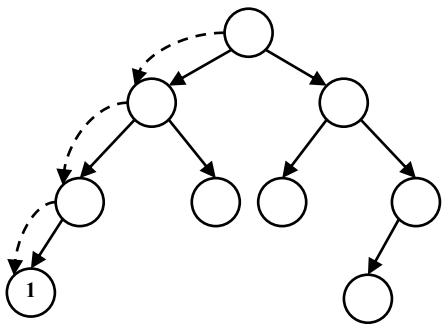
Donat un arbre el primer node que considerarem en in-ordre serà aquell que està més a la esquerra, per tant quan inicialitzem un iterador ens haurem de situar en aquell node. Per això cridem a la funció leftmostNode(), la qual simplement recorre el arbre sempre que pot per la esquerra.

Es una funció similar a la primera versió del setInOrder(), ja que solament requereix d'una crida recursiva en cas de que el node que estem tractant té un fill esquerre.

```
private Node<E> leftmostNode(Node<E> node){
    return node.left != null ? leftmostNode(node.left) : node;
}
```

Després al fer els testos vam veure que teníem problemes amb arbres buits, per això es va afegir una comprovació quan el node fos null. Ja que no podem accedir al fill esquerre de null.

Si el node té fill esquerre ens seguim movent, sinó voldrà dir que ja estem al node de més a la esquerra i per tant el tindrem que retornar.



[16] Node resultant, 1, de la crida de la funció leftmostNode

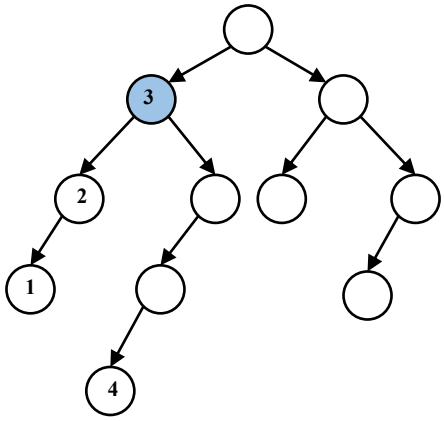
4.4.2 nextInOrder()

Ara que sempre estarem situats a la primera posició ens haurem de moure per dins del arbre. Com que tenim referències al següent node en in-ordre, gràcies al constructor dels nodes es molt senzill accedir al següent element, doncs només hem de comprovar que el node tingui una referència al node següent i no al seu fill dret. Per tant tenim dos casos:

Quan el node que estem tractant no té fill dret hem de retornar la seva referència:

```
} else {
    return node.rightOrNext;
}
```

Quan el node si que té un fill dret es una mica més complicat, primer fiquem aquest arbre com exemple:



[17] Situació on el node blau té fill dret

Imaginem que ens trobem en el node de color blau amb el número 3, aquest node sí que te un fill dret, però si retornéssim el seu fill dret no estaríem fent un recorregut en in-ordre. Per retornar el node correcte hem de moure'ns sempre el màxim a la esquerra possible.

Com que ja tenim una funció que ens fa aquesta feina, solament ens falta cridar-la.

```
if (node.isRightChild){
    return leftmostNode(node.rightOrNext);
}
```

Si compactem una mica la funció tindrem:

```
private Node<E> nextInOrder(Node<E> node){
    return node.isRightChild ?
        leftmostNode(node.rightOrNext) : node.rightOrNext;
}
```

4.4.3 next()

Amb les funcions auxiliars definides podem acabar de completar el next(). Primer comprovem que el següent node a retornar no sigui *null* i llençar la excepció corresponent.

```
if(!hasNext()){ throw new NSEE("...")}
```

Hem d'actualitzar el *lastReturned* amb el que era el *next* anterior.

```
lastReturned = next;
```

A continuació haurem d'obtenir el següent node que atribuirem a next, solament cal una crida a la funció nextInOrder.

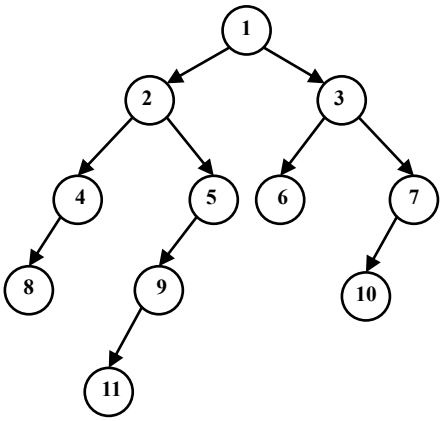
```
next = nextInOrder(next);
```

I finalment retornem l'element del next previ, com que ara es troba en lastReturned haurem de retornar-lo:

```
return lastReturned.element;
```

5. levelOrderIterator

El levelOrderIterator, com el seu nom indica, és una forma de recórrer un arbre binari per nivells [18].



[18] Arbre binari recorregut en per nivells

Per realitzar aquesta implementació no podrem utilitzar cap algorisme recursiu, per la naturalesa del recorregut, és per això que necessitarem fer una implementació iterativa utilitzant cues.

5.1 Constructor

Aquesta implementació té la característica que no es calcula el següent element a mesura que anem fent next(), sinó que primer s'emmagatzemen tots els resultats en ordre i es van retornant de la cua amb cada crida del next.

És per això que el constructor tindrà que inicialitzar una cua de nodes amb tots els resultats:

```
LevelOrderIterator{
    lastReturned = null;
    queue = LevelOrder(LinkedBinaryTree.this.root);
}
```

lastReturned simplement ens guardarà el últim node que s'ha retornat.

5.2 set()

La funció set modificarà l'element del últim node retornat per el que li enviem per paràmetre. Primer haurem de comprovar que l'últim element retornat un node, és a dir que no sigui null. I si aquest no es el cas podem modificar l'element.

```
public void set(E e){
    if(lastReturned == null){
        throw new IllegalStateException("...");
    }
    lastReturned.element = e;
}
```

5.3 hasNext()

Es un booleà que indica si existeix un següent node en la iteració. En aquest cas comprovem si la cua no és buida.

```
public boolean hasNext(){
    return !queue.isEmpty();
```

5.4 next()

La funció next() farà ús de la cua que conté tots els nodes ordenats per nivells. Apart de les comprovacions i excepcions quan no hi ha més elements:

```
if(!hasNext()){
    throw new NSEE("...");
}
```

Haurem d'agafar el següent node de la cua, això ho aconseguim amb un simple poll() sobre la *queue* i l'atribuïm al *lastReturned*.

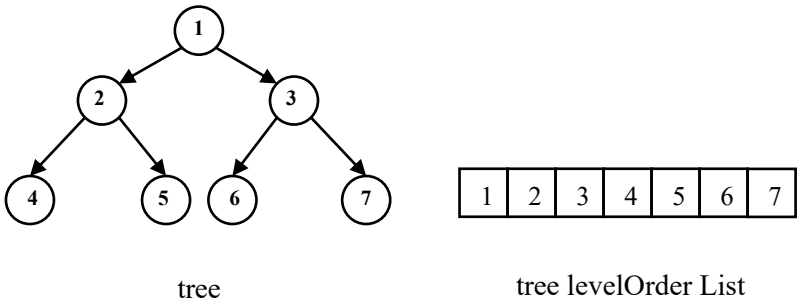
```
lastReturned = queue.poll();
```

Finalment retornem l'element d'aquest últim node.

```
return lastReturned.element;
```

5.5 levelOrder()

La funció levelOrder es la que ens permet construir una cua ordenada en funció els nivells del arbre binari. En l'enunciat se'ns proposava utilitzar el LinkedList tant com el ArrayDeque. I nosaltres hem volgut provar a fer els dos.



[19] Representació d'arbre binari a la esquerra i la seva llista a la dreta contenent els elements ordenats per nivells

5.5.1 levelOrder: LinkedList

El objectiu es molt senzill, donat un arbre, volem crear una llista que contingui de forma ordenada els elements del arbre per nivells de dalt a baix i de esquerra a dreta [19].

El primer que haurem de fer serà inicialitzar una nova llista que contindrà els resultats.

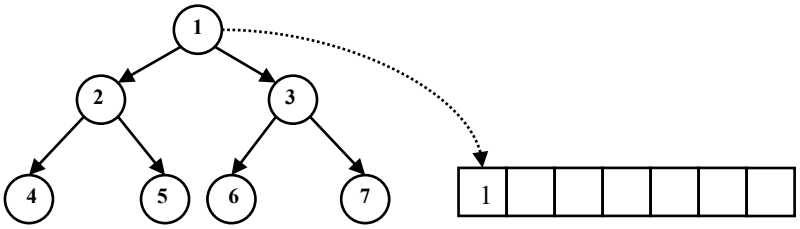
```
var result = new LinkedList<Node<E>>();
```

Necessitarem fer una comprovació de l'arrel que rebem, no sigui cas que aquesta sigui *null*.

```
if (root == null) return result;
```

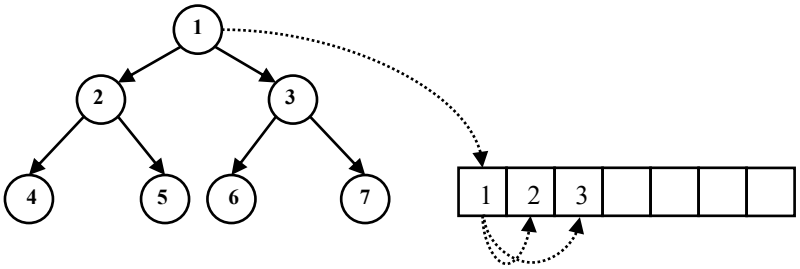
Ara podem començar a pensar com s'aniran afegint els elements dins de la llista *result*.

Donat un node hem d'assegurar-nos que aquest node i els seus possibles fills estiguin dins de la llista seguint el seu ordre per nivells. Així que per començar haurem d'afegir l'arrel dins de la llista [20].



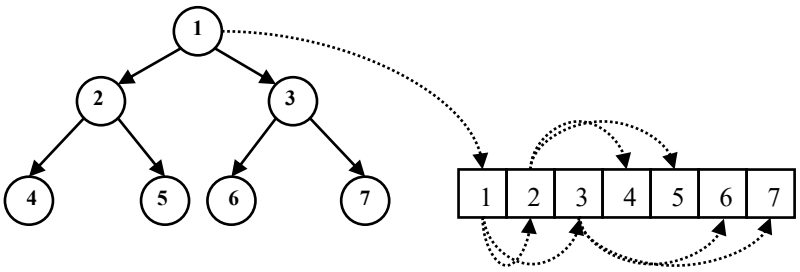
[20] Afegim l'arrel al principi de la llista

Ara ens interessa accedir als fills de l'arrel que està dins de la llista, i així afegir-los [21].



[21] Accedim a l'arrel per obtenir els seus fills i afegir-los.

Podem repetir aquesta operació amb els diferents nodes que hi trobem [22].



[22] Seguim afegint els nodes dels diferents fills.

La pregunta es en quin punt haurem de parar de realitzar aquesta operació, com que disposem de la mida del arbre accedint al *root.size* poder realitzar un bucle que es realitzi tantes vegades com de mida és el arbre.

Ara podem convertir aquest raonament en codi, hem dit que primer afegirem l'arrel a la llista:

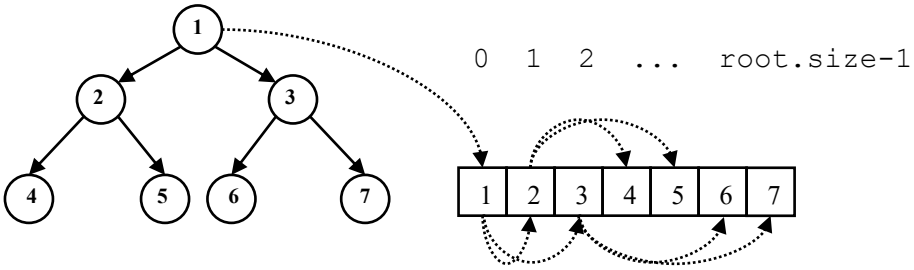
```
result.add(root);
```

A continuació haurem de realitzar un bucle que vagi afegint els fills a la llista, com que hem dit que és realitzarà en funció de la mida del arbre utilitzem el *root.size*:

```
for(int i = 0; i < root.size; i++){...}
```

Per accedir al node que volem obtenir els seus fills podem realitzar un get() sobre la llista, enviant-li la posició en la que estem "i".

```
var node = result.get(i);
```



[23] Iteració sobre la llista des de la posició 0 fins root.size-1

Tot seguit haurem d'afegir aquells nodes que siguin diferents de *null*:

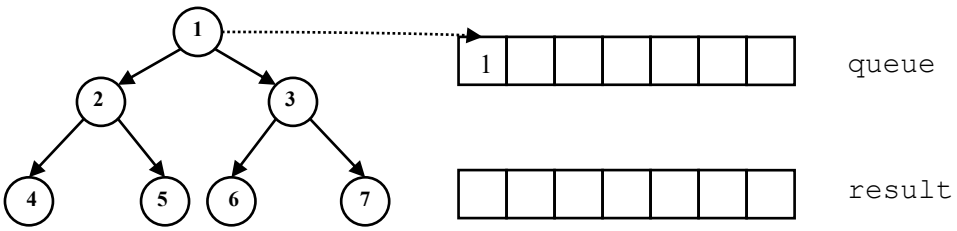
```
if(node.left != null) result.add(node.left);
if(node.right() != null) result.add(node.rightOrNext);
```

Encara que es una implementació completa i retorna el resultat a esperar, té un gran problema respecte a la següent implementació que hem fet amb ArrayDeque, i és la seva complexitat. Aquesta funció té un cost $O(n^2)$, això és a causa de com funciona el mètode get() en una LinkedList, doncs cada vegada que es crida al get ha de recórrer tota la llista de principi a fi fins trobar el element en el índex donat. Motiu pel que aquesta implementació no es bona candidata.

5.5.2 levelOrder: ArrayDeque

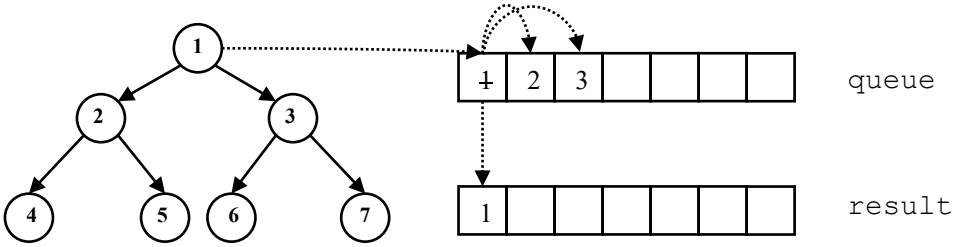
Per arreglar el problema anterior i fer la funció més eficient, ens podem ajudar d'una nova cua, de forma que anirem afegint els nodes fill a aquesta cua i posteriorment els traurem per ficar-los al resultat final.

El plantejament és molt similar, primer afegirem l'arrel a la cua auxiliar.



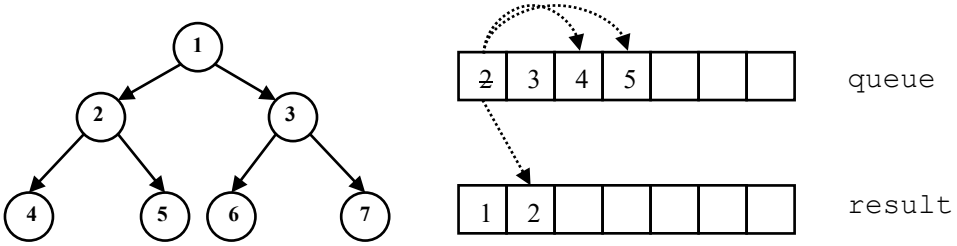
[24] Afegim l'arrel a la cua auxiliar "queue"

Ara haurem d'obtenir el node que es troba en la *queue* i afegir-lo a la cua final. Com que aquest element ja el tenim ficat l'haurem d'eliminar de la *queue.*



[25] Afegim l'arrel a result i els seus fills a la queue

A continuació haurem de fer el mateix amb els seus fills:



[26] Repetim l'operació anterior amb els seus fills

I anem repetint aquesta operació fins omplir la llista result, la pregunta ara es quin moment acabarem de realitzar aquesta operació? Quan la queue estigui completament buida, el que significa que ja no hi hauran més elements per afegir al resultat.

Si convertim el plantejament en codi tenim primer, la inicialització de la cua auxiliar i afegim l'arrel:

```
Deque<Node<E>> queue = new ArrayDeque<>();
queue.add(root);
```

A continuació haurem de realitzar el bucle de la operació fins que la cua auxiliar estigui buida.

```
while (!queue.isEmpty()) {...}
```

Ara haurem d'obtenir el node que estem tractant, amb l'operació poll() podem obtenir el següent element de la cua a la vegada que l'eliminem d'aquesta.

```
Node<E> node = queue.poll();
```

I a continuació l'afegirem a la llista resultat.

```
result.add(node);
```

A continuació haurem de tractar els fills (en cas d'existir) del node, per això com hem fet amb l'arrel, si son diferents de null els afegirem a la cua auxiliar.

```
if(node.left != null) queue.add(node.left);
if(node.right() != null) queue.add(node.rightOrNext);
```

Finalment tenim la funció levelOrder acabada:

```
private Deque<Node<E>> levelOrder(Node<E> root){
    var result = new ArrayDeque<>();
    if (root == null) return result;
    Deque<Node<E>> queue = new ArrayDeque<>();
    queue.add(root);
    while (!queue.isEmpty()) {
        Node<E> node = queue.poll();
        result.add(node);
        if(node.left != null) queue.add(node.left);
        if(node.right() != null) queue.add(node.rightOrNext);
    }
    return result;
}
```

El fet d'utilitzar una cua auxiliar ens permet aprofitar-nos del mètode poll() per obtenir un cost constant, i així el cost d'aquesta funció es converteix en lineal O(n). L'únic inconvenient es que utilitzem més espai a memòria.

Pel que hem vist per internet és la implementació més utilitzada en estructures d'aquest tipus.

6. Disseny de Tests

Per comprovar la correcta funcionalitat dels mètodes implementats hauríem de poder tenir un bon conjunt de tests que sigui capaç de comprovar tots els casos possibles. En el cas de la construcció d'arbres binaris amb nodes i recorreguts en ells, amb un seguit de tests que comprovin els casos definits en els mètodes recursius ja serien suficients, ja que si funciona en un cas si es generalitza de forma recursiva hauria de funcionar igualment.

6.1 CopyTest

Per comprovar la correcta funcionalitat de les copies d'un arbres ens hem d'assegurar de:

- Una copia ha de funcionar de forma independent.
- Les referències als seus elements han de ser compartides.
    - Això implica que si l'element es mutable, tant copia com original s'han de veure afectats.
    - I si no es mutable no es veuran afectats.
- Si es fa una copia d'un subarbre, aquesta copia no pot tenir referències a algun node que hagi estat exclòs del arbre original.

Tant la independència entre copia i arbre i les seves referències internes les podem comprovar utilitzant els iteradors que s'han implementat més endavant, en els casos més senzills fem servir la funció setRoot().

Els cas més senzill a comprovar serà quan realitzem la copia d'un arbre buit.

```
assertEquals(empty,new LinkedBinaryTree(empty));
```

Per verificar si les referències funcionen correctament disposem del mètode setRoot(E e), el qual modifica l'element de l'arrel del arbre. Així podem dissenyar un tes molt simple que inicialitza un arbre solament amb l'arrel i la seva copia:

```
var tree = new LinkekBinaryTree(null,2,null);
var copy = new LinkekBinaryTree(tree);
```

Com que els enters en Java no son elements mutables, quan modifiquem l'arrel d'aquest arbre la seva copia no s'hauria de veure afectada.

```
tree.setRoot(5);
assertEquals(2,copy.root());
```

Ara podem provar amb elements mutables i comprovar que tant original com copia es veuen afectats. Un exemple d'element mutable es un StringBuilder:

```
var tree = new LinkekBinaryTree(null,new StringBuilder("Hello"),null);
var copy = new LinkekBinaryTree(tree);
```

Els StringBuilder tenen diverses funcions per modificar el seu contingut en el nostre cas utilitzem la funció append(), que afegeix el text enviat per paràmetre al StringBuilder ja existent.

```
tree.root().append("world");
```

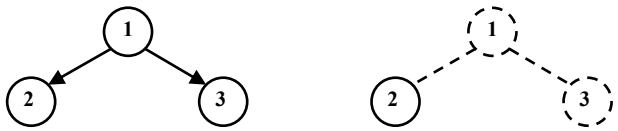
Si ara tant la copia com el original tenen l'arrel modificada amb els mateixos valors voldrà dir que es mantenen les referències al mateix element.

```
assertSame(tree.root(),copy.root());
```

Sempre podem afegir més tests jugant amb els fills drets i esquerres però sempre seran els mateixos casos ja que solament considerem nodes i els seus elements ja siguin mutables o immutables.

Per comprovar la independència entre nodes podem utilitzar els iteradors, de forma que si la copia i el original tenen un iterador, iterar sobre un arbre no afectarà al iterador de la seva copia.

Aquesta comprovació l'hem fet amb un arbre binari complet de dos nivells, i la seva copia esquerra:

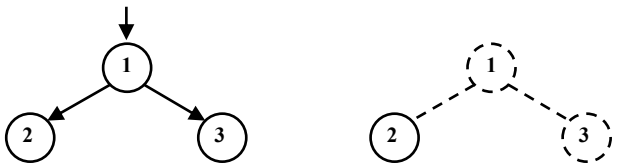


[27] Arbre binari de dos nivells a la esquerra, i sub-arbre esquerre a la dreta.

Ara si inicialitzem un iterador en tots dos arbres el seu next es situaria al mateix lloc, en cas de ser iterador en in ordre.

[28] Inicialització d'iteradors en inOrdre, fletxa apuntant a next

En canvi al realitzar un altre next la copia no tindrà següent element, ja que funcionen de forma independent.



[29] Fletxa del arbre original apuntant al següent element, però la copia ja no en té.

De forma similar hauria de passar amb el iterador per nivells, així ens assegurem que la copia i l'arbre original seran arbres independents.

També disposem de dos tests provant la iteració de la copia d'un sub-arbre, assegurant-nos que les referències a nodes superiors no es mantenen en la copia.

6.2 AbstractLinkedBinaryTreeTest

Abans de definir els tests dels iteradors, es important mencionar la classe de tests AbstractLinkedBinaryTreeTest, pel simple fet que ens facilitarà molt els tests a realitzar.

Com que no podem estar comprovant un assertEquals, next() i així repetidament s'ha definit un mètode "iterate" que retorna una llista amb tots els elements d'una iteració sobre un arbre. Així solament requerim d'un assertEquals amb aquest mètode i un altra llista ja definida per nosaltres amb els resultats esperats.

```
static <E> List<E> iterate(Iterator<E> it){...}
```

Com que treballarem amb dos tipus d'iteradors, estaria be tenir ja construïts arbres binaris i fer-los servir en tots dos classes de tests. I precisament és el que hem fet, ens hem construït una sèrie d'arbres binaris agafant tots els "casos possibles".

Considerem els nostres tests tots els casos possibles encara que no els podem tenir tots, ja que son infinits, degut que els arbres construïts presenten estructures que qualsevol arbre binari ha de contenir, per tant si estem segurs que aquestes estructures "bàsiques" funcionen correctament, podem estar bastant segurs que quan generalitzem també funcionaran.

- Arbres buits
- Arbres complets de nivell 1 (arrel), 2 o 3
- Arbres amb un fill dret
- Arbres amb un fill esquerre
- Arbres amb referències a nodes superiors

Cada arbre que creem estarà acompanyat d'un petit diagrama en format text mostrant l'estructura que tindria.

6.3 IteratorsTest

Com hem mencionat abans utilitzarem els mateixos arbres per tots dos tests, la estructura dels tests en iteradors tindrà la estructura següent:

- Inicialitzem llista amb els resultats esperats.
- assertEquals entre la llista esperada i la crida de la funció iterate.

Per exemple:

```
void inOrder1(){
    var expected = List.of("root","right");
    assertEquals(expected,iterate(rootRight.inOrderIterator()));
```

Altres comprovacions importants a considerar seran les excepcions i que els mètodes set(), hasNext() funcionin correctament. Això ho comprovem amb arbres d'un sol element, i executant next(), o set() en els diferents casos.

Podríem fer-ho amb arbres amb més elements però el fet de que es llenci una excepció sempre es redueix a casos simples:

- **NSEE:** Quan no hi ha més nodes en el recorregut, no importa si el arbre te 0 o 1000 elements, sempre es redueix al mateix cas (l'últim element).
- **ISE:** Quan no s'ha fet una crida al next(), aquesta excepció es independent a la mida del arbre.

En el cas de hasNext(), com que indirectament l'estem utilitzant en el mètode iterate, ja podem estar segurs que funcionarà de forma correcta.

Un aspecte a remarcar és la iteració en sub-arbres en el inOrderIterator, quan fem una crida al mètode right() o left() és retorna nodes compartits amb el original motiu pel que si fem una iteració sobre un sub-arbre esquerre, la iteració continuarà per tot el arbre.

En el cas de les copies, que han de ser independents, ja ho tenim considerat en l'apartat 6.1.

7. Observacions/Conclusions

Per acabar aquest informe, voldríem justificar d'alguna forma la llargada d'aquest. En alguns apartats es mostra el desenvolupament i raonaments de mètodes els quals en un principi eren incorrectes i encara així els hem mantingut. Això era per mostrar com ha evolucionat el plantejament a mesura que es feien tests i implementacions d'altres mètodes i ens pareixia una mica frustrant tenir que esborrar per complet aquell raonament. Per això hem optat en analitzar el problema que teníem i en base a les conclusions presentar la solució final.

Els diagrames han jugat una part molt fonamental en el transcurs de tota la pràctica no només a nivell elaborar un bon informe, sinó com ajuda cap a la programació.

Finalment comentar el desenvolupament de les pràctiques durant l'assignatura i en concret aquesta, on a comparació amb les primeres que tenien un enunciat molt més pautat amb tests ja inclosos, aquesta en particular ens ha deixat més al nostre propi raonament i investigació sobre els problemes a tractar en comptes de mirar contínuament el enunciat per copiar les indicacions del professor. En general una bona pràctica i estem satisfets amb el resultat.