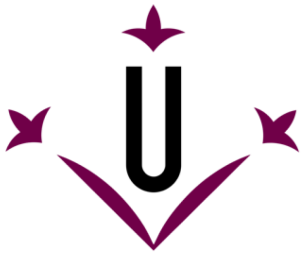

Universitat de Lleida

Arbres Binaris i Recorreguts

Boulhani Zanzan, Hamza

Serrano Ortega, Aniol

Data: 22/12/2023

Laboratori 4

Estructura de Dades

Escola Politècnica Superior

Índex

Índex de Figures

1. Introducció

Aquest projecte es centra en la implementació i exploració de quatre característiques clau en l'àmbit de les estructures de dades d'arbres binaris. Primerament, es desenvoluparà un constructor de còpia per a arbres binaris, una tasca fonamental per garantir una gestió eficaç de les còpies d'aquestes estructures. Seguidament, es tractarà la implementació de l'enhebrat simple en inordre, una tècnica que optimitza el recorregut dels arbres.

A més, el projecte inclou la creació d'iteradors en inordre, permetent un recorregut seqüencial i ordenat dels elements de l'arbre. Finalment, es durà a terme la implementació d'iteradors per nivells, utilitzant cues de nodes per a una navegació eficient a través de les diferents capes de l'arbre.

El desenvolupament d'aquest projecte es realitzarà sobre una base ja proporcionada, on es requerirà completar les funcions pendents i afegir els tests necessaris per assegurar el correcte funcionament de les implementacions. Aquest enfocament metodològic garantirà una comprensió profunda i pràctica dels mètodes per a la manipulació i gestió d'arbres binaris.

2. Descripció i desenvolupament

Primerament, cal definir què és una enhebració simple en inordre, aquesta és un tipus d'estratègia que s'empra de forma iterativa sense utilitzar una pila o la forma recursiva.

Es caracteritza pel fet que, partint des del node més a l'esquerra possible, si aquest no té fill dret, el seu apuntador passarà a ser el següent node inordre que vindria a ser el seu pare. Quan estem en el node del pare i passa el cas en què no té fill esquerre, l'apuntador passarà a ser el node que el precedeix; si no té fill dret, l'apuntador passarà a apuntar al node que el segueix. Així successivament. Això ho podem apreciar millor amb el següent diagrama.

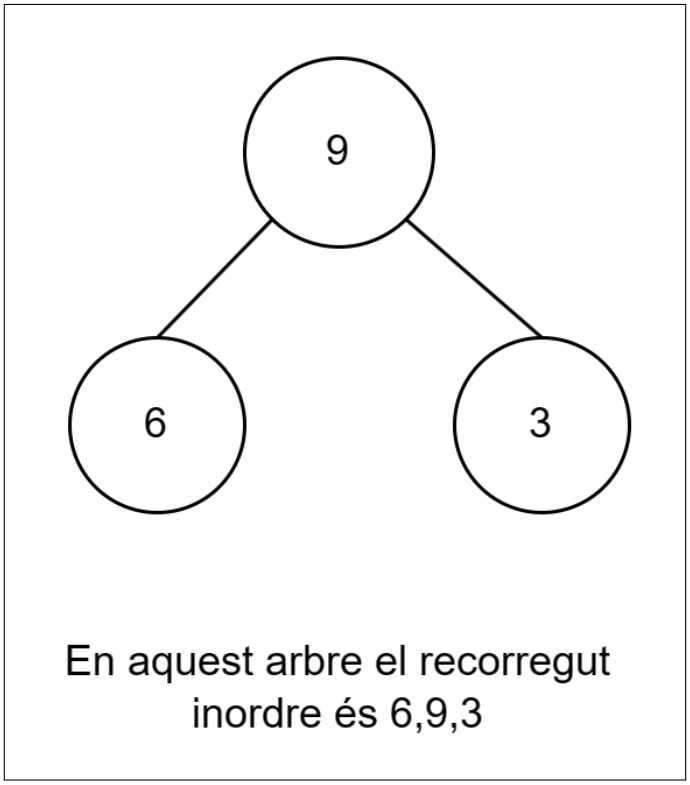

Figura 1: Exemple de recorregut en inordre

Seguim amb la descripció de la còpia d'arbres binaris, que té com a objectiu evitar problemes relacionats amb la compartició de referències. Finalment, abordarem la implementació d'iteradors per nivells. Aquesta funcionalitat permet recórrer l'arbre binari pels seus diferents nivells de manera iterativa, utilitzant cues de nodes. Això assegurarà que tots els nodes hagin estat visitats abans de canviar al següent nivell.

2.1. La classe *LinkedBinaryTree*

A la classe *LinkedBinaryTree* representa ser la classe principal d'aquest laboratori, la qual és en la que s'han de fer les diferents implementacions. Les següents classes es troben aniuades dins d'aquesta de forma privada, amb alguns mètodes públics.

Seguidament, trobem el que seria la implementació de l'arbre binari de la nostra estructura de dades, que estaria composada per nodes, on cada node podria tenir un màxim de 2 fills, un anomenat dret i l'altre esquerre. Les parts més fonamentals serien:

Root és el node arrel de l'arbre binari, és a dir, tots són fills d'ell. Seguidament, trobem el constructor `LinkedBinaryTree`, que crea arbres a partir d'altres subarbres. L'altre constructor amb el mateix nom genera un arbre buit, sense cap node.

2.2. La classe *Node*

Aquesta classe representa un node dins de l'arbre binari, cada node que conté una referència als seus fills dret i esquerre. Dins de la classe interna node, trobem diverses funcions essencials. Per exemple, *size* s'utilitza per determinar quants descendents té el node, *isRightChild* serveix per identificar si el node és fill dret, establint així la seva relació amb el pare. Finalment, per a crear aquest constructor s'ha afegit una referencia al node pare en cas

Constructor de la classe *Node*

```
1 Node(Node<E> left, E element, Node<E> right) {
2     this.left = left;
3     this.element = element;
4     this.right = right;
5     if (left != null) {
6         left.parent = this;
7     }
8     if (right != null) {
```

```
9           right.parent = this;
10      }
11      this.size = 1 + Node.size(left) + Node.size(right);
12  }
```

A més a més, la funció *copy* és un dels pilars fonamentals de la classe interna node, ja que copia directament un node, facilitant la creació d'una còpia d'un arbre binari.

2.3. La Classe *InOrderIterator*

La classe `InOrderIterator` implementa la interfície `BinaryTreeIterator`, proporcionant la capacitat de recórrer un arbre binari. Aquesta classe inclou diverses funcions i mètodes clau.

- **Constructor *InOrderIterator()***: Inicialitza l'iterador posicionant `next` al node més esquerre de l'arbre, que és el primer node a ser visitat en aquest recorregut.
- **Mètode `leftMost(Node<E> node)`:** És un mètode auxiliar que serveix per a retornar el node més a l'esquerra del subarbre.
- **Mètode `set(E e)`:** Serveix per a substituir l'element de l'últim node retornat per `next()` amb un nou element `e`. En cas que `next()` no hagi estat cridat abans, es llança una `IllegalStateException`.
- **Mètode `hasNext()`:** Si encara hi ha elements a recórrer, retorna `true`. Això es determina comprovant si `next` és no nul.
- **Mètode `next()`:** Retorna l'element del següent node en el recorregut inordre i actualitza `next` per apuntar al següent node a visitar. En cas que no hi hagi més elements, es llança una `NoSuchElementException`.
- **Mètode `getNext(Node<E> node)`:** És un mètode auxiliar per a determinar el següent node a visitar després del node actual, seguint la lògica del recorregut inordre. Si el node actual té un subarbre dret, retorna el node més esquerre d'aquest subarbre. Altrament, puja per l'arbre fins trobar un node que no sigui el fill dret del seu pare.

2.4. La classe *LevelOrderIterator*

La classe *LevelOrderIterator* dóna la capacitat de recórrer l'arbre pels seus diferents nivells de l'esquerra a la dreta. Primer de tot, tindrem una cua (*queue*) que és la cua de nodes de tipus `ArrayDeque` que s'utilitzarà per recórrer els nivells. Es parteix de la base que no hi nodes nuls en la cua.

- **Constructor** `LevelOrderIterator(Node <E> root):` Aquest constructor inicia el recorregut per nivells afegint el node arrel a la cua en cas que l'arbre estigui buit.
- **Mètode** `set(E e):` Substitueix l'element de l'últim node retornat per `next()`. En cas que `next()` no hagi estat cridat abans es llança una excepció `IllegalStateException`. Aquest mètode permet modificar els elements de l'arbre durant el recorregut.
- **Mètode** `hasNext():` Retorna `true` si queden més elements per recórrer a la cua.
- **Mètode** `next():` Retorna l'element del node següent en l'ordre de recorregut per nivells. Si no hi ha més elements, llança una `NoSuchElementException`. Aquest mètode també actualitza la cua afegint els fills esquerra i dret del node retornat, si existeixen, mantenint així l'ordre de recorregut per nivells.

3. Tests

3.1. *AbstractLinkedBinaryTreeTest*

Aquesta classe defineix un arbre que servirà per a realitzar els següents tests. D'aquesta classe es defineix un arbre anomenat `tree` que serveix per a comprovar la implementació dels iteradors i de la copia. De forma visual aquest arbre és així:

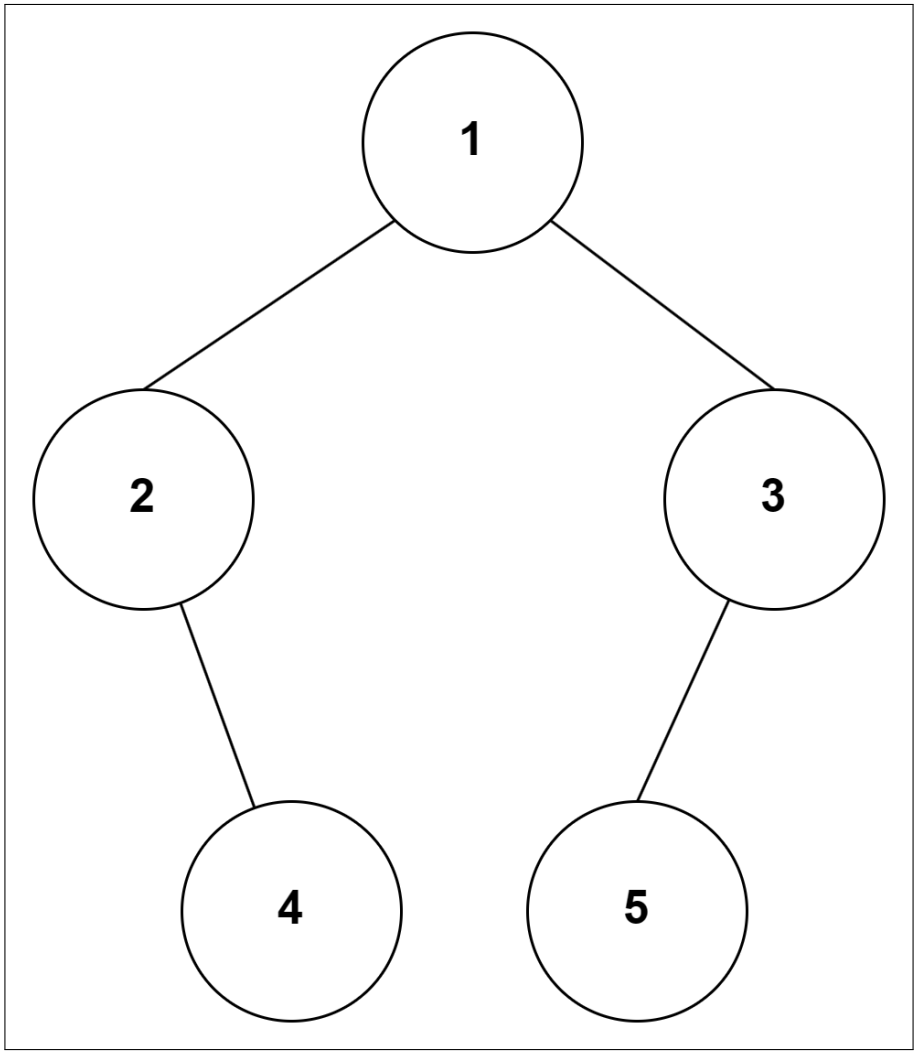

Figura 2: Diagrama del arbre `tree`

3.2. *CopyTest*

El test *CopyTest* el que faria és comprovar si l'arbre es copia de la manera correcta. Ho fa de la següent forma: Primer de tot, crea una còpia de l'arbre original, anomenat *copytree*, on hauria de ser exactament el mateix arbre que l'original.

```
1 @Test
2 void testCopy() {
3     LinkedBinaryTree<Integer> copyTree = new
          LinkedBinaryTree<>(tree);
4     ...
5 }
```

Seguidament, comprovem que la còpia sigui idèntica a l'original, en cas de fallar es mostrarà el següent missatge:

```
3     assertEquals(tree, copyTree, "The copy should be equal
          to the original");
```

Després, el que es fa és canviar el valor de l'arrel de l'arbre copiat a 50. Per comprovar si el valor que hem canviat ha afectat l'arbre original, fem el següent:

```
4     copyTree.setRoot(50);
5     assertNotEquals(tree.root(), copyTree.root(), "
          Modifying the copy should not affect the original");
```

En el cas que hagi afectat a l'arbre original, es llançarà el missatge *Modifying the copy shouldn't affect the original.*

3.3. *InOrderIteratorTest*

Aquests testos serveixen per comprovar si s'ha implementat la capacitat de fer el recorregut inordre de la manera correcta. Per aixo, verifiquem si el recorregut sobre l'arbre `tree` en inordre és el correcte, és a dir, `[2, 4, 1, 5, 3]`.

```
1 @Test
2 void testInOrderNonEmpty() {
3     var list = List.of(2, 4, 1, 5, 3);
4     assertEquals(list, iterate(tree.inOrderIterator()));
5 }
```

El següent test comprova que el recorregut d'un arbre buit és nul:

```
1 @Test
2 void testInOrderEmpty() {
3     assertTrue(iterate(empty.inOrderIterator()).isEmpty(),
        "Inorder of an empty tree should be empty");
4 }
```

El següent test comprova el comportament dels mètodes *hasNext* i *next*. Primer de tot, comprova que el mètode *hasNext* retorna `true` si l'arbre no és buit i també comprova si *next* retorna el primer element recorregut en inordre, que hauria de ser 2. Per últim, tenim el `testNextException`, que el que fa és comprovar que el mètode *next* llança una excepció si es truca a un iterador d'un arbre buit. El codi és el següent:

```java
@Test
void testNextException() {
    var iterator = empty.inOrderIterator();
    assertFalse(iterator.hasNext(), "hasNext should return
       false for an empty tree");
    assertThrows(NoSuchElementException.class, iterator::
       next, "Calling next on an empty iterator should
       throw NoSuchElementException");
}
```

3.4. *LevelOrderIteratorTest*

Aquests tests serveixen per comprovar si s'ha implementat de manera correcta la classe *LevelOrderIterator*. Per comprovar aixo es fan 2 testos diferents.

- **Inorder level of a non-empty tree**: El primer test verifica que el recorregut per nivells d'un arbre no buit coincideix amb el valor retornat. Podem comprovar en la figura 2 que aquest recorregut ha de retornar la llista [1, 2, 3 4, 5].
- **Inorder level of an empty tree**: El segon test comprova que el recorregut en ordre d'un arbre buit sigui també buit.

El primer test té la següent aparença:

```java
@Test
void testInOrderLevel() {
    var expected = List.of(1, 2, 3, 4, 5);
    assertEquals(expected, iterate(tree.levelOrderIterator
       ()));
}
```

4. Conclusions

En aquest projecte, hem aprofundit en els conceptes i les implementacions relacionades amb els arbres binaris. Específicament, centrant-nos en la creació de constructors de còpia, l'enhebrat inordre, i la implementació d'iteradors inordre i per nivells.

Finalment, en aquest projecte hem après la importància de fer tests per a cada fase de la implementació, ja que un error una fase inicial de la implementació, es pot propagar fins al final de la implementació. Aquest fet s'ha donat i ens ha dut problemes, mitjançant tests i anàlisi amb el *debugger* s'han pogut arreglar aquests errors.