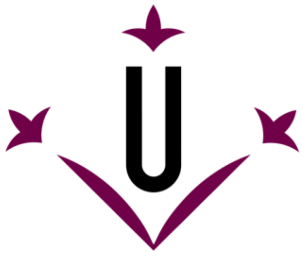

Universitat de Lleida

Particions d'un número natural

Serrano Ortega, Aniol

Boulhani, Hamza

Data: 06/11/2023

Laboratori 2

Estructures de Dades

Escola Politècnica Superior

Índex

Índex de Figures

1. Introducció

En aquest projecte hi ha tres objectius principals, implementar la classe *LinkedStack* (2.), implementar els tests d'aquesta *LinkedStack* (3.) i transformar un algorisme de recursiu a iteratiu amb la pila implementada prèviament (4.).

2. Definició i implementació de la pila

Per a implementar la *LinkedStack* prèviament és necessari conèixer el concepte d'una pila. Una pila (*stack* en anglès) és una estructura de dades del tipus LIFO[1]. Això implica que els elements s'afegeixen i es retiren des del mateix extrem de la pila anomenat *top*.

En la classe `LinkedStack<E>` s'implementa una llista enllaçada[2] per representar la pila. Els elements d'aquesta llista estan representats per nodes enllaçats.

2.1. La classe *Node*

Per a facilitar la implementació d'aquesta pila, s'ha creat una classe privada, estàtica i aniuada dins de la classe `LinkedStack<E>` anomenada `Node<E>`. Aquesta classe ha de ser privada, ja que no es té cap interès en què sigui accessible des de mètodes externs, ja que només és emprada per a representar cada element de la llista. D'aquesta forma no es pot modificar el comportament intern de la classe sinó, que només es pot veure el comportament extern de la classe.

D'altra banda, la classe ha de ser estàtica, ja que no necessita accedir als atributs ni mètodes de la classe envoltant (`LinkedStack<E>`), evitant així l'associació implícita amb una instància de *LinkedStack*. D'aquesta manera la classe `Node<E>` és privada per encapsular la seva funcionalitat dins de `LinkedStack<E>`, i estàtica perquè no requereix accés als membres no-estàtics de la classe envoltant.

[1] LIFO (*Last In, First Out*): L'últim element que entra és el primer en sortir.

[2] Llista enllaçada: És una seqüència d'elements que cada un conté una referència (enllaç) al següent element de la llista.

En la classe `Node<E>`, inicialment, es defineix un atribut de tipus genèric `E` que representa l'element que conté el node, així com un atribut anomenat `next` de tipus `Node<E>` que referencia el següent node de la pila. Aquests atributs permeten construir la llista enllaçada en forma d'una seqüència de nodes, on cada node té una referència al següent node en la seqüència.

El constructor de la classe `Node<E>` s'encarrega d'inicialitzar aquests dos atributs. D'aquesta forma, el constructor rep dos elements, l'element que ha de contenir el node, i una referència al següent node en la llista. D'aquesta manera, quan es crea un nou node, es pot especificar l'element que contindrà així com el node següent en la llista.

2.2. La classe *LinkedStack*

La classe `LinkedStack<E>` és la implementació de la pila basada en una llista enllaçada, on cada element està representat per un node. Per a implementar aquesta pila, es requereix d'una sèrie de mètodes bàsics per a la manipulació d'aquesta llista. Com és natural, aquests mètodes implementen el funcionament d'una pila del tipus LIFO.

Primerament, en aquesta classe conté un atribut que referència al node que està al cim de la pila, anomenat *top*. A més, un atribut que representa la llargada d'aquesta pila. Seguidament, es defineix el constructor que inicialitza la pila buida, és a dir, amb mida igual a zero i amb la referència al node *top* igual a *null*.

2.2.1. Mètode *push*

El primer mètode implementat s'anomena *push*, aquest mètode naturalment afegeix l'element passat com a paràmetre a la llista com un node, addicionalment s'ha d'actualitzar la referència al node *top* al nou node i s'ha d'augmentar la mida. D'aquesta manera, cada nou element que s'insereix a la pila resulta ser el node al cim de la pila.

Per a exemplificar visualment aquest mètode, s'ha plantejat el següent codi:

```
1    var elem1 = new LinkedStack<Integer>();
2    elem1.push(10);
```

Partint d'una pila buida, el mètode resulta així:

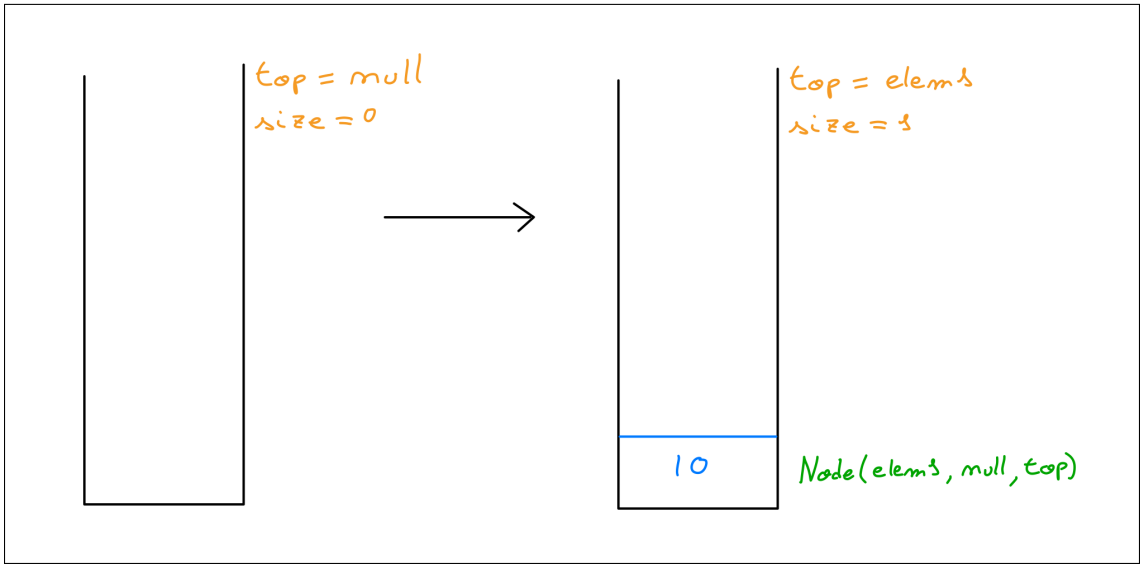

Figura 1: Funcionament del mètode *push* → 1

L'estructura d'un node és la següent:

```
Node(E element, Node<E> elementPrevi, Node<E> elementPosterior)
```

Seguidament, s'insereix un nou element:

```
1    var elem2 = new LinkedStack<Integer>();
2    elem2.push(20);
```

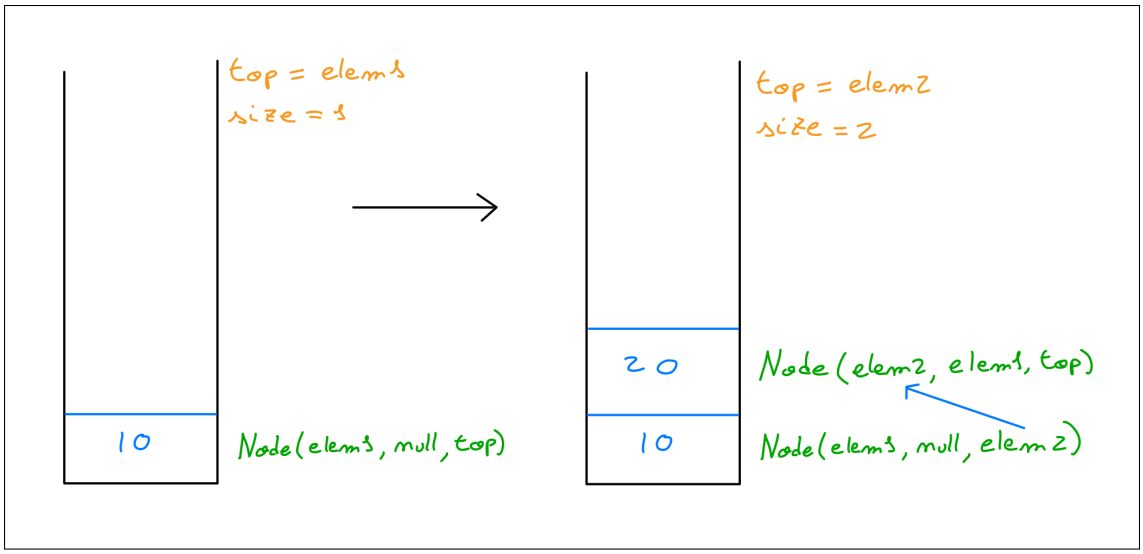

Figura 2: Funcionament del mètode *push* → 2

2.2.2. Mètode *top*

El mètode top retorna l'element que està al cim de la pila, per tant, l'atribut *top* de la classe. Altrament, és adequat considerar el cas en què top és igual a *null*, això significa que la pila és buida, en conseqüència, com que no hi ha cap element a la pila s'ha de llançar una excepció anomenada *NoSuchElementException*.

2.2.3. Mètode *pop*

Aquest mètode li succeeix el mateix que el mètode anterior, és a dir, en el cas que la pila sigui buida s'ha de llançar una excepció. En cas contrari, ha d'eliminar l'element al cim de la pila. Per fer això, s'assigna el següent node al node *top* com al nou *top*. Finalment, a l'haver eliminat un node de la pila, s'ha de decrementar la mida d'aquesta.

2.2.4. Els mètodes *isEmpty* i getSize

El mètode *isEmpty* és el mètode més trivial de la classe, ja que aquest només retorna un booleà en funció de la mida de la pila. D'altra banda, s'ha creat un mètode addicional que actua de *getter* de la mida de la pila, ja que, aquesta variable és privada per evitar la manipulació externa d'aquesta.

3. Tests unitaris

Per a comprovar el correcte funcionament de les implementacions, és adient afegir una bateria de tests per a verificar que el comportament de la implementació és l'adequat. En aquest projecte s'han realitzat dues implementacions diferents, el *LinkedStack* i el *PartitionsIterative*, per tant, és adient comprovar el seu funcionament amb una bateria de tests per a cada implementació. Aquests tests s'ubiquen en els fitxers `LinkedStackTest.java` i `Partitions2Test.java` respectivament.

En el cas de la *LinkedStack*, s'han creat un conjunt de tests (específics per a cada mètode) per avaluar el comportament d'aquests. A continuació es detallen els tests elaborats:

- ***isEmpty()* (2)**: S'ha comprovat que en cas que la pila estigui buida i es faci aquesta crida, retorni verdader. També, s'ha comprovat el cas a l'inrevés, és a dir, el cas en què la pila no és buida i al fer aquesta crida retorni falç.
- ***push()* (1)**: S'ha creat un test per a verificar que aquest mètode efectivament inserit al munt de la pila l'element passat com a paràmetre. En aquest cas, s'insereix un element i es verifica amb el *getter* de la mida de la pila, es verifica que aquesta mida sigui igual al nombre d'elements inserits, és a dir, 1.
- ***pop()* (2)**: Aquest mètode elimina l'element de la cima de la pila, per tant, en el cas que la pila està buida ha de retornar una excepció. El segon test tracta de verificar que per a un nombre n d'elements inserits a la pila, si es fa un *pop* el nombre d'elements de la pila ha de ser n-1.
- ***top()* (5)**: Aquest mètode resulta el més difícil de verificar que el seu comportament és l'adequat i, per tant, s'ha hagut d'implementar un major nombre de tests i combinar els mètodes previs. Primerament, s'ha verificat que en cas d'estar la pila buida llanci una excepció de la mateixa forma que es fa en el cas del *pop*. Seguidament,

4. Transformació a iteratiu

La transformació a iteratiu es basa en transformar una algorisme recursiu en iteratiu fent us d'una pila. Per fer-ho, es fa servir la classe *Context* que emmagatzema els contextos de cada crida recursiva. Cada context conté l'estat necessari per poder seguir l'execució després de retornar d'una crida recursiva.

Per tant, els diferents punts d'entrada i sortida, que en aquest cas són `CALL`, `RESUME1` i `RESUME2`. A més, es requereixen les variables temporals `f1` i `f2` per emmagatzemar els resultats a mesura que es calculen les particions.

Per representar els punts d'entrada, s'utilitza el següent tipus enumerat `EntryPoint`:

Enumerat *EntryPoint*

```
1 private enum EntryPoint {
2     CALL, RESUME1, RESUME2
3 }
```

També es defineix la classe estàtica privada `Context`, on es declaren les variables i tots els paràmetres que seran necessaris per a l'algorisme iteratiu:

Classe *Context*

```
1 private static class Context {
2     final int n;
3     int minAddend;
4     int f1;
5     int f2;
6     EntryPoint entryPoint;
7
```

```
8     public Context(int n, int minAddend) {
9         this.n = n;
10        this.f1 = 0;
11        this.f2 = 0;
12        this.minAddend = minAddend;
13        this.entryPoint = EntryPoint.CALL;
14    }
15 }
```

A continuació, es descriu el procés iteratiu:

1. **Inicialització de la pila**: S'afegeix un context inicial a la pila amb els valors de n i el mínim addend *minAddend*, i amb el punt d'entrada `CALL`.

```
1     public static int partitionsIter(int n) {
2         assert n > 0;
3         return partitionsIter(n, 1);
4     }
```

2. **Algorisme Iteratiu**: Mentre la pila no estigui buida:
   - Es treu el context del cim de la pila.
   - Segons el valor de l'*entryPoint* dins del context, s'executa la següent lògica:
     - **CALL**: Si n és igual a *minAddend*, això implica que només hi ha una partició possible, que és n mateix. Si n és menor que *minAddend*, no hi ha particions possibles. Si no es compleix cap d'aquests casos, es crea un nou context amb els valors actualitzats i es canvia l'*entryPoint* a `RESUME1`.
     - **RESUME1**: Es guarda el resultat parcial en $f1$ i es prepara per calcular la següent part de la fórmula de partició afegint un nou context a la pila amb l'`entryPoint RESUME2`.

- **RESUME2**: Es guarda el resultat parcial en $f2$ i es calcula el valor final sumant $f1$ i $f2$, que representa el total de particions per aquests valors de n i $minAddend$.

Per tant, de forma conceptual l'estructura dels casos dels punts d'entrada és és la següent:

Switch amb *EntryPoints*

```
var context = stack.top();
switch (context.entryPoint) {
    case CALL -> {...}
    case RESUME1 -> {...}
    case RESUME2 -> {...}
}
```

Aquesta estratègia iterativa evita el problema de l'ús excessiu de memòria i el potencial *stackoverflow* que pot ocórrer amb l'algorisme recursiu, especialment per valors grans de n.

Primerament, en el punt d'entrada `CALL` es verifica si es compleix algun dels dos casos simples: si `context.n == context.minAddend`, aleshores `return_ = 1`, ja que només hi ha una única partició en aquest cas. Si `context.n < context.minAddend`, aleshores `return_ = 0`, ja que no es pot fer cap partició. En ambdós casos, s'elimina el `context` de la pila. Si no es compleix cap dels casos simples, el programa canvia el `EntryPoint` a `RESUME1` i s'afegeix un nou `context` a la pila amb `(context.n - context.minAddend, context.minAddend)`.

En `RESUME1`, el valor de `return_` s'emmagatzema en `context.f1`, i el `EntryPoint` canvia a `RESUME2`. A continuació, s'afegeix un nou `context` a la pila amb `(context.n, context.minAddend + 1)` per seguir verificant que encara queden particions.

En `RESUME2`, el valor anterior de `return_` s'emmagatzema en `context.f2`, i després s'actualitza `return_` amb la suma de `context.f1` i `context.f2`, eliminant finalment el `context` de la pila.

Seguidament, s'exemplifica el cas de l'algorisme per a $n = 2$ i $minAddend = 1$:

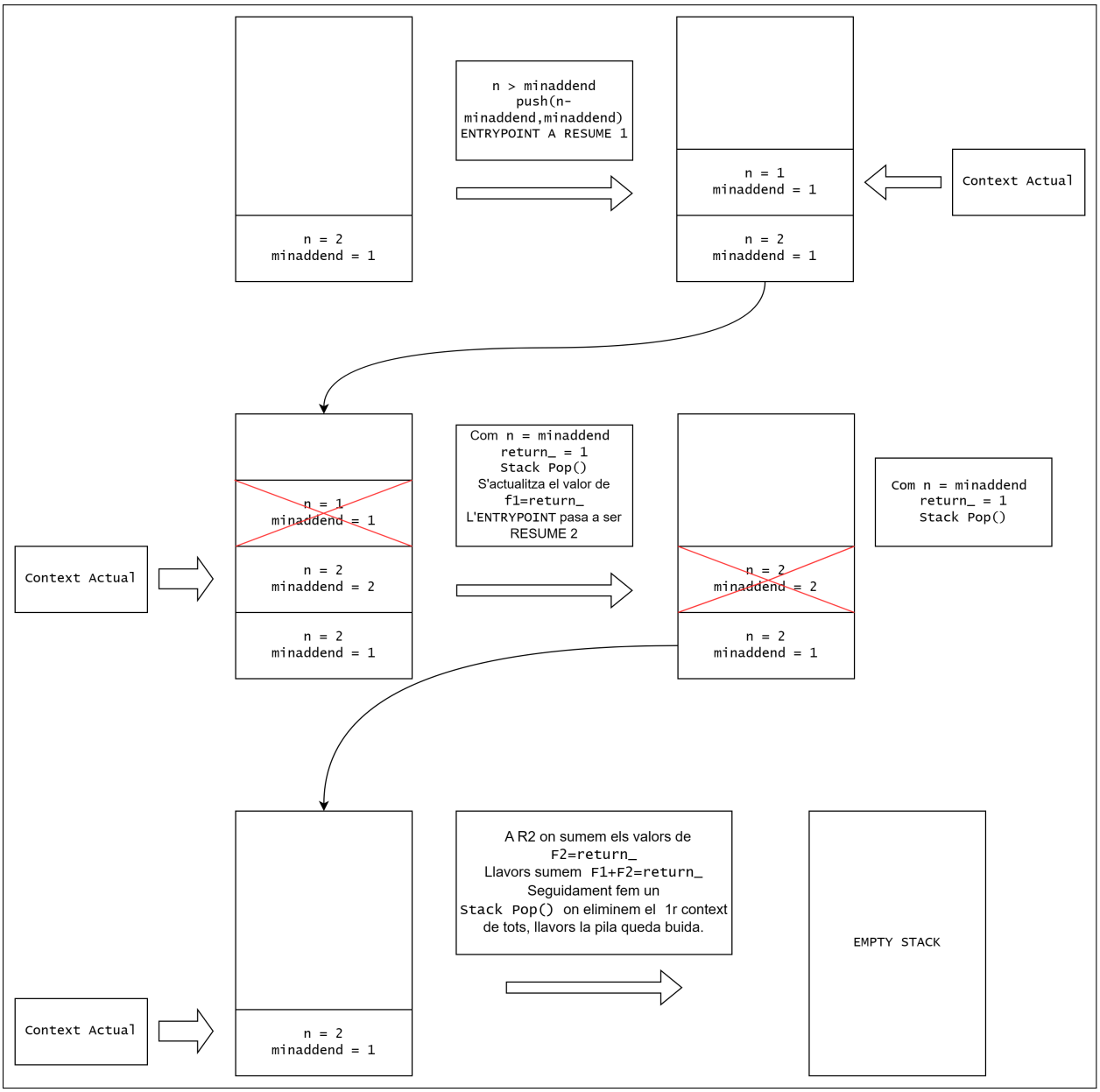

Figura 3: Funcionament del *Partitions2*

En el diagrama anterior, es pot observar que per a $n = 2$ i minAddend $= 1$, es comença marcant com a context actual el valor de $n = 2$ i minAddend $= 1$, ja que aquest és el primer element que es col·loca al cim de la pila. Seguidament, s'entra al bucle `While`; mentre la pila no estigui buida, el bucle continuarà executant-se.

Dins del bucle, es pot veure que pel fet que $n >$ minAddend, es crea un nou context amb $(n - \text{minAddend}, \text{minAddend})$ i el `EntryPoint` es modifica per "apuntar" a `RESUME1`, on els nous valors a la part superior de la pila són $n = 1$ i minAddend $= 1$. Pel fet que es detecta que es compleix el cas simple $n =$ minAddend, s'assigna return_ $= 1$ i es retira de la pila mitjançant `pop()`. Al entrar a `RESUME1`, s'actualitza el valor de $f1$ amb el de return_ i es canvia el `EntryPoint` a `RESUME2`, afegint un nou context a la pila amb $n = 2$ i minAddend $= 2$. Aquest context segueix els mateixos passos que el segon context, on minAddend $= n$. Així, s'iguala return_ a 1 i s'elimina el context de la pila.

Tal com s'ha esmentat prèviament, el `EntryPoint` es troba a `RESUME2`, on es registra $f2$ amb el valor de return_. Seguidament, return_ s'actualitza per ser la suma de $f1$ i $f2$, resultant en $1 + 1$ en aquest escenari. Finalment, s'elimina l'últim context restant a la pila, que correspon a $n = 2$ i minAddend $= 1$, arribant així al final del procés, amb la pila buida.