Datastructures

Opdracht 1: Linked list

Naam: Hugo de Groot
Studentnummer: 500755313
Klas: iS-201
Docent: Gerke de Boer
Datum: 4-1-2019

Inhoudsopgave

Voorwoord

Voor het vak datastructures heb ik de opdracht gekregen om een linked list te implementeren in een bestand dat we vanuit school kregen. De opdracht was om alle methodes volledig te implementeren en om extra test cases toe te voegen. In dit verslag zal ik bespreken wat ik heb veranderd aan de methodes om ze werkend te krijgen en welke test cases ik toe heb gevoegd. Voor de opdracht moesten we ook JVM argumenten meegeven. Ik heb de opdracht in Intelijj gemaakt. Om de JVM argumenten mee te geven moet je het VM options field aanpassen bij de instellingen van je project.

1 Add methode

1.1 De opgeleverde code

De code die wij kregen om mee te werken heeft wat problemen. De code voert nog niet de taken uit die wij willen dat hij uitvoert en daarom moeten wij wat aanpassingen gaan maken. Dit eerste stukje code heeft als functie een element aan een lijst toevoegen. De code die ik gekregen heb voegt niks toe aan een lijst, maar hij verandert dezelfde variabel met een nieuwe waarde. De grote van de lijst wordt hierbij ook niet aangepast. Dit zijn de dingen die ik ga implementeren in de code.

```
public void add(T element) { head = new Node(element, head); }
```

1.2 Mijn implementatie

Om ervoor te zorgen dat de add methode goed functioneert heb ik wat code toegevoegd. Als eerste heb ik een variabel aangemaakt om de grootte van de lijst bij te houden. Dit doe ik met het variabel

```
public void add(T element) {
        Node currentLast = tail;
        Node temp = new Node(element, null);
        tail = temp;

        if (currentLast == null) {
            head = temp;
            size++;
        }
        else {
            currentLast.next = temp;
            currentLast.next.setPrevious(currentLast);
            size++;
```

```
        }
    }
```

2 Get methode

2.1 De Opgeleverde code

De huidige code loopt via deze methode door alle nodes heen. Hij zet de volgende node als de huidige node neer en zo gaat hij ze allemaal af. Het probleem hiermee is dat het niet stopt wanneer het een negatief getal leest of wanneer het getal te groot is.

```
public T get(int index) { Node node = head; while (index-- > 0) { node = node.next; } return node.value; }
```

2.2 Mijn implementatie

Ik heb de code aangepast waardoor hij nu doet wat hij zou moeten doen. De methode loopt alleen door de nodes heen wanneer de index tussen de 0 en de lengte van de lijst min 1 is. Hij zal net zo vaak door de lijst heen lopen als jij aangeeft. Als je bijvoorbeeld 5 opgeeft zal de methode 5 keer de volgende node pakken en de laatste die hij vindt teruggeven. Er wordt een exceptie gegooid wanneer de index niet voldoet aan de eisen.

```
public T get(int index) {
   Node current = head;

   if (index >= 0 && index < size) {
       for (int i = 0; i < index; i++) {
       current = current.next;
       }
   } else {
       throw new IllegalArgumentException("The index is out of bounds");
   }

   return current.value;
}
```

2.3 Optimalisatie

Als we de get methode willen optimaliseren dan moeten we ervoor zorgen dat de methode achterstevoren loopt wanneer hij een index over de helft wil krijgen. Wanneer je bijvoorbeeld de index 45 wilt hebben uit een lijst met 50 elementen dan zou de methode 45 stappen moeten nemen. In de optimalisatie zou de methode bij het getal 45 van achter naar voren gaan loopen omdat 45 meer dan de helft van 50 is. Nu hoeft de methode maar 5 stappen te nemen en deze is dus aanzienlijk sneller. Om dit voor elkaar te krijgen moet het programma voor iedere node niet alleen een next, maar ook een previous hebben. Zodra elke node een link naar het vorige element had kon de methode aangepast worden om ervoor te zorgen dat hij ook van achter naar voren kan loopen.

Nu, wanneer de index groter dan de helft van de lijst is veranderd de index zodat de loop sneller bij het juiste element aan kan komen. Als je bijvoorbeeld element 17 van de 20 wilt veranderen dan moet je dus 3 keer de vorige aanroepen en dan het huidige element aanpassen naar wat je er voor in de plaats wilt hebben.

```
public T get(int index) {
   Node current;

   if (index >= 0 && index < size) {
       current = tail;
       for (int i = 0; i < index; i++) {
           current = current.previous;
       }
   } else {
       current = head;
       for (int i = 0; i < index; i++) {
           current = current.next;
       }
   }
}
   else {
       throw new IllegalArgumentException("The index is out of bounds");
   }

   return node.value;
}
```

3 Delete methode

3.1 De opgeleverde code

De code die opgeleverd is zet het tweede element neer als het huidige element. Hierbij zijn er een aantal problemen. Ten eerste is er altijd een next en daardoor kan er geen nullPointerException gegooid worden. Ten tweede wordt er nooit wat met het element dat je wilt verwijderen gedaan. De methode wordt dus wel aangeroepen maar hij doet nu nog niks. Daar heb ik verandering in gebracht.

```
public void delete(T element) {
head = head.next;
}
```

3.2 Mijn implementatie

Ik heb een aantal dingen aangepast aan de delete methode om hem zo soepel mogelijk te laten werken. Ik zal hieronder ingaan op de methodes die ik toegepast heb en wat alles erin betekent. De delete methode werkt nu en kan ook meerdere elementen verwijderen.

```
public void delete(T element) {
   delete(head, element);
}

private void delete(Node head, T element) {
   if (head == null) {
       System.out.println("no Nodes to remove");
   }
       else{
       if (head.value == element && head.next == null) {
           head = null;
           size--;
       } else if (head.value == element && head.next != null) {
           head = head.next;
           size--;
       } else {
           head = head.next;
       }
       delete(head, element);
   }
}
```

3.2.1 public delete

Wanneer je een element uit de lijst wilt verwijderen dan roep je als eerste de public delete methode aan en geef je het element mee dat je wilt verwijderen. Deze public delete methode roept weer de private delete methode aan en geeft hiervoor het gegeven element en de head mee. De reden hiervoor is dat je anders niet meerdere elementen kan verwijderen. Dit komt omdat de head aan het einde van de lijst te vinden is en mocht je het dan nogmaals aan willen roepen dan zal de methode proberen aan het einde van de lijst te beginnen, waar niks zal zitten, en zal de methode niks uitvoeren en direct stoppen.

3.2.2 private delete

De private delete methode wil altijd het eerste element van de huidige lijst en het element dat je wilt verwijderen hebben. Voor deze opdracht moesten wij recursie gebruiken en omdat ik dit heb geïmplementeerd moet het ook een stop criterium hebben. Om dit aan te geven zeg ik dat head niet null mag zijn. Zodra head null is dan zal de methode niks uitvoeren. Het element zal verwijderd zijn en de methode kan dus eindigen. De delete methode heeft 3 checks om ervoor te zorgen dat alles goed verloopt en een element ook daadwerkelijk goed uit de lijst verwijderd wordt.

Check 1
In de eerste check kijkt de methode of de head het element is dat je wilt verwijderen en of het volgende element leeg is. Als dit het geval is dan zal het eerste element worden verwijderd, ofwel leeg gemaakt, en zal de lengte van de lijst verlaagd worden met 1.

check 2
In de tweede check kijkt de methode wederom of de head het element is dat je wilt verwijderen en of er een volgend element is. Als we erachter komen dat er een volgend element is dan zetten we het huidige element waar we alles mee checken naar het volgende element. Dat volgende element wordt dan de head. Hierna wordt de lengte van de lijst weer met 1 verlaagd.

check 3
Als allebei de check niet voldoen dan weten we dat het eerste en het tweede element niet null is of het element dat je wilt verwijderen. Mocht dit het geval zijn dan wil je net als bij check 2 het huidige element naar het volgende element verplaatsen.

recursie
Wanneer alle drie de checks uitgevoerd zijn roep je weer de delete methode aan met de huidige head. De head is nu 1 stap verder dan de vorige keer dat je de methode aangeroepen hebt. Vervolgens checkt de methode alles weer, maar nu met het tweede element in plaats van de eerste. Dit doet de methode net zo lang tot er geen volgend element is zoals aangegeven in check 1, of tot wanneer de lijst helemaal leeg is. Dit wordt aangegeven in de criteria.

4 Size methode

De methode die wij mee kregen in dit programma om de size te bepalen gaf altijd 0 terug wanneer de methode aangeroepen werd. Dit heb ik veranderd naar een methode die een variabel teruggeeft dat constant verandert wanneer er iets toegevoegd wordt aan de lijst of verwijderd.

```
public int size() {
   return size;
}
```

5 Test methodes

5.1 getLastElement

```
@Test
public void getLastElement() {
   testList.add("one");
   assertEquals("one", testList.get(1));
}
```

De testmethode die meegeleverd is is fout. De methode voegt altijd 1 element toe en wil dat element vervolgens vergelijken met het volgende element, maar dat element bestaat helemaal niet. Hierdoor zal deze test altijd een fout teruggeven en is de test dus verkeerd. Om dit op te lossen heb ik de size van de lijst gepakt en hier 1 er van af getrokken zodat je altijd de lengte van de lijst terugkrijgt.

```
@Test
public void getLastElement() {
   testList.add("one");

   assertEquals("one", testList.get(testList.size() - 1));
}
```

5.2 deleteFromMiddleOfList

```
@Test
public void deleteFromMiddleOfList() {
   testList.add("one");
   testList.add("two");
   testList.add("three");
   testList.add("four");
   testList.add("five");
   testList.add("six");
   testList.delete("three");
   testList.delete("four");

   assertEquals(4, testList.size());
}
```

In de test die ik hier geschreven heb is mijn doel om te testen of de delete methode op de juiste manier elementen kan verwijderen uit het midden van een lijst. De test voegt eerst 6 elementen toe aan de lijst, waarna hij er ook weer twee uit het midden verwijdert. Hierna houden we er 4 over. De test wordt met succes afgerond.

5.3 deleteFromMiddleOfListWithMultipleOccurrences

```
@Test
public void deleteFromMiddleOfListWithMultipleOccurrences() {
   testList.add("one");
   testList.add("two");
   testList.add("one");
   testList.add("four");
   testList.add("one");
   testList.add("six");
   testList.delete("one");

   assertEquals(3, testList.size());
}
```

In de test die ik hier geschreven heb is mijn doel om te testen of de delete methode op de juiste manier meerdere elementen kan verwijderen met dezelfde naam. Eerst worden er 6 elementen toegevoegd waarvan er 3 dezelfde naam hebben. Vervolgens worden deze drie verwijdert met 1 delete methode en checkt de test case of er nog 3 elementen in de lijst zitten. De test wordt met succes afgerond.

5.4 deleteFromAndAddToListWithMultipleOccurrences

```
@Test
public void deleteFromAndAddToListWithMultipleOccurrences() {
   testList.add("one");
   testList.add("two");
   testList.add("one");
   testList.delete("two");
   testList.add("four");
   testList.add("one");
   testList.add("six");
   testList.delete("one");
   testList.add("five");
   testList.add("seven");
   testList.add("eight");
   testList.delete("five");

   assertEquals(4, testList.size());
}
```

In de test die ik hier geschreven heb is mijn doel om te testen of de delete methode op de juiste manier opeenvolgend meerdere elementen kan verwijderen en toevoegen. Er worden hier steeds elementen toegevoegd, sommige met dezelfde naam, en verwijdert. Aan het einde van de test zouden er nog 4 elementen over moeten zijn. De test wordt met succes afgerond.

5.5 GetFromMiddleElement

```
@Test
public void getFromMiddleElement(){
   testList.add("one");
   testList.add("two");
   testList.add("three");
   testList.add("four");
   testList.add("five");
   testList.add("six");
   testList.add("seven");
   testList.add("eight");
   testList.add("nine");
   testList.add("ten");
   assertEquals("four", testList.get(5));
}
```

In de test die ik hier geschreven heb is mijn doel om te testen of de get methode het goede element uit de lijst pakt. De test case vult hier de lijst met getallen van 1 t/m 10. Vervolgens kijkt hij of 4 op de juiste plek in de lijst staat. De test wordt met succes afgerond.

5.6 BiggerIndexIsNotAllowed

```
@Test(expected = IllegalArgumentException.class)
public void biggerIndexIsNotAllowed() {
   testList.add("one");
   testList.add("two");
   testList.get(testList.size(+10));
}
```

In de laatste test die ik hier geschreven heb is mijn doel om te testen of het programma een exceptie gooit wanneer er een index gevraagd wordt die veel groter is dan de lijst lang is. Hij voegt twee elementen aan de lijst toe en voert vervolgens de get methode uit met de lengte van de lijst +10. Dit geeft een error en de test wordt met succes afgerond omdat we aan het begin aangeven dat we een error verwachten.