Assignment 7 - Breath first

Jesper Hesselgren

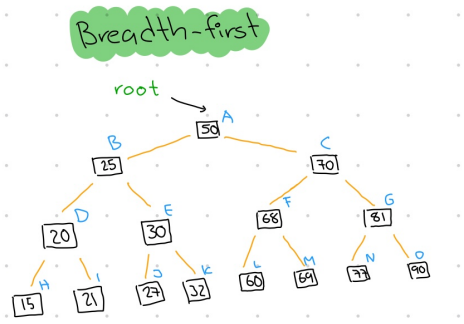
Oktober 10, 2024

Introduktion

I denna rapport fortsätter vi att utforska den komplexa datastrukturen träd, med särskilt fokus på binära träd. I vår tidigare rapport undersökte vi hur vi kunde traversera ett binärt träd i *in-order*, vilket innebär att vi går igenom noderna från den längst ner till vänster till den längst till höger. I denna rapport ska vi istället undersöka hur vi kan traversera ett binärt träd i *breadth-first* ordning, vilket innebär att vi traverserar trädet en nivå i taget med start vid roten. Vi ska sedan kolla på hur vi kan implementera en så kallad *lazy sequence* för vår breadth-first traversering. En lazy sequence gör det möjligt att hämta ett element i taget, istället för att traversera hela trädet på en gång.

Breadth-first

I figuren nedan ser vi en visuell representation av ett binärt träd som vi ska traversera från nod A (rot-noden) till nod O i alfabetisk ordning.



Figur 1: En skiss över ett binärt träd som ska traverseras från A till O

För att traversera trädet i *breadth-first* ordning använder vi den dynamiska kö som vi implementerade i Assignment 6. Vi justerar denna kö så att den kan hantera noderna i vårt binära träd istället för heltal, som den ursprungligen var designad för. Med hjälp av denna kö kan vi hålla reda på vilken node som ska besökas och skrivas ut härnäst.

Vi börjar med att kontrollera om trädet är tomt. Om så är fallet returnerar vi omedelbart, eftersom det inte finns något att traversera. Annars skapar vi en instans av vår anpassade kö-klass, som är konfigurerad för att hålla objekt av typen Node. Vi lägger sedan till en referens till roten i kön och ökar räknaren n, som håller reda på hur många objekt som för närvarande finns i kön. Denna räknare hjälper oss att hålla koll på när kön är tom.

Därefter går vi in i en while-loop som körs så länge kön inte är tom. Inuti loopen följer vi en procedur där vi först tar bort den nod som ligger längst fram i kön och skriver ut dess värde. Efter detta minskar vi räknaren n. Vi kontrollerar sedan om den aktuella noden har några barn. Om det finns ett vänsterbarn, läggs det till i kön, och om det finns ett högerbarn, läggs även detta till i kön. Loopen fortsätter med samma procedur tills alla noder i trädet har besökts och kön är tom.

```
public void breathFirst() {

    // Om trädet är tomt returnar vi bara
    if (root == null) {
        return;
    }

    QueueArray queue = new QueueArray(20);

    Node cur = root;
    queue.enqueue(cur);
    queue.n++;

    //Körs till kön är tom
    while (!queue.empty()) {
        cur = queue.dequeue();
        queue.n--;
        System.out.println(cur.value);

        if (cur.left != null) {
            queue.enqueue(cur.left);
            queue.n++;
        }
```

```
            if (cur.right != null) {
                queue.enqueue(cur.right);
                queue.n++;
            }
        }
    }
```

Lazy sequence

I vår `breadthFirst()`-metod kan vi traversera hela trädet i ett svep och i den ordning vi önskar. Problemet är dock att vi inte kan traversera exempelvis fyra noder, pausa för att utföra en annan uppgift, och sedan fortsätta traverseringen från där vi slutade.

Vi ska nu undersöka hur vi kan lösa detta genom att implementera en ny datastruktur, `Sequence`, som håller reda på tillståndet i vår traversering. På så sätt kan vi begära värdet i nästa node i ordningen när vi behöver den, utan att behöva traversera hela trädet på en gång. `Sequence` klassen består enbart av ett attribute som är våran justerad Node-kö samt en metod `next()` som möjligör att enbart returnera ett node-värde i taget från vårt träd.

Vi börjar med att implementera metoden `sequence()` i vår BinaryTree-klass. Denna metod returnerar enbart en instans av den nya datastrukturen Sequence, där vi skickar med trädet som argument. Genom att göra detta initierar vi Sequence-objektet med roten av trädet, vilket låter oss använda Sequence för att traversera trädet stegvis i breadth-first ordning.

```
public Sequence sequence() {
      return new Sequence(this);
  }
```

I koden här nedan ser vi konstruktorn i Sequence-klassen. När en ny instans av Sequence skapas skickas ett BinaryTree-objekt som argument. Konstruktorn kontrollerar först om trädet har en rot-nod. Om trädet inte är tomt initieras en kö och rot-noden läggs till som det första elementet i kön, vilket blir startpunkten för vår `breadth-first` traversering. Om trädet är tomt kastas ett exeception som påpekar att trädet är tomt.

```
    public Sequence(BinaryTree tree) {

            if (tree.root != null) {
                queue = new QueueArray(20);
```

```
            queue.enqueue(tree.root);
        } else {
            throw new NoSuchElementException("Trädet är tomt");
        }

    }
```

Här nedan ser vi implementation av `next()`-metoden i våran `Sequence`-klass som möjligör att bara kan hämta enstaka värden från vårat träd.`Next()`-metoden kontrollerar först så att vår kö som håller noderna i vårat träd inte är tom. Om kön är tom kastas ett undantag, annars hämtar next() den första noden i kön och returnerar dess värde. Därefter lägger den till nodens vänstra och högra barn i kön, om de existerar. Detta gör att vi kan fortsätta hämta ett nodvärde i taget från trädet.

```
    public int next() {

    if (queue.empty()) {
        throw new NoSuchElementException("Kön är tom");
    }

    Node firstInQueue = queue.dequeue();
    // Lägger till vänster barn
    if (firstInQueue.left != null) {
        queue.enqueue(firstInQueue.left);
    }
    // Lägger till höger barn
    if (firstInQueue.right != null) {
        queue.enqueue(firstInQueue.right);
    }

    return firstInQueue.value;
}
```

Slutsats/diskussion

I denna rapport har vi undersökt hur vi kan traversera ett träd med breadth-first-ordningen. Vi såg att det gick att implementera relativt smärtfritt med hjälp av vår modifierade kö, som hanterar våra noder. Genom att använda våran kö kunde vi effektivt traversera trädet nivå för nivå och säkerställa att noderna bearbetades i breadth-first-ordning. Vi implementerade också en Sequence-klass som möjlig gjorde en stegvis traversering, vilket gav oss flexibiliteten att pausa och återuppta traverseringen när så önskas. Detta kan

vara särskilt användbart i situationer där vi inte vill bearbeta hela trädet på en gång, utan istället begära och hantera ett nodvärde i taget