Binärträd

Adalet Adiljan

Hösten 2023

Introduktion

När länkade listor har nu givit oss en grundläggande och fundamental förståelse kring hur datastrukturer förhåller sig, kan vi upptäcka den sannerliga mångsidigheten av att utforska mer inriktade konstruktioner. I denna uppgift har vi granskat oss i en specifik variant av träd som kallas binära träd. Begreppet 'träd' fångar dess natur på ett passande sätt, då den börjar från en enda rot och förgrenar sig i understrukturer, vilka i sin tur förgrenar sig vidare eller slutar i individuella 'löv' som ska representera de slutliga noderna.

Binära träd

Den mest simpla länkade listan består utav en linjär sekvens vars ena nod pekar på nästa nod i raden. I ett binärt träd expanderar vi detta och inkluderar grenar. I binära träd finns det också noder, men vrje nod kan förgrenas upp till två noder, 'barn'- en i vänster och en till höger. Detta bidrar till en hierarkisk struktur, där vi har en överordnad nod som är en rot och från den kan följa grenarna nedåt i lövnoderna. Denna datastruktur kan användas för att organisera och strukturera data i en hierarkisk form, där ett balanserat eller obalanserat träd kan bidra till olika konsekvenser. Man strävar i generellt efter ett balanserat träd för att upprätthålla effektiv sökning och insättning av nya noder i trädet.

Metod

Trädens struktur

I vårt binära träd vi definierat en inre klass kallad 'Node', som i generellt definierar strukturen för varje nod. I klassen har vi fyra attributer som för en nod har givit till fyra egenskaper. I klassen har vi även definierat en konstruktor som tar emot värdena på en nyckel och ett värde. Konstruktorn väcks till kraft när vi ska skapa nya noder med ett par av nyckeln och värdet som den blir associerat till. I början av den nya noden är den

högra respektive vänstra sidan satta till 'null' (indikerar att den saknar undernoder).

Till vår binära träd har vi skapat de två funktionerna kallade 'add' respektive 'lookup'. Innan funktionerna implementerades så var det värt att komma ihåg att noderna i trädet är ordnade i rekursiv storleksordning, där mindre nycklar sorteras till vänster och större nycklar åt höger. Det innebär att vi kan implementera både 'add' och 'lookup' funktionen rekursivt. Vår add-funktion tar emot två argumenter, en nyckel och ett värde värda ett heltal. Om värdet på nyckeln är unik, mappas denna nyckel till värdet oavsett om värdet återfinns. Om värdet på nyckeln finns, uppdateras enbart värdet. Den rekursiva anropet sker när nyckeln som vi försöker lägga till är mindre än nyckeln i den aktuella noden och det finns en vänster gren, går vi till vänster och utför add-funktionen reskursivt på denna gren. Medan i ett annat fall då nyckeln vi vill lägga till är större än nyckeln i den aktuella noden och det finns en höger gren, så går vi till höger och utför add-funktionen rekursivt på den högra delen av trädet. I de fall då det inte finns varken en vänster respektive höger gren och vi fortfarande inte har hittat nyckeln instanserar vi en ny ny nod och sätter den som högra grenen för den aktuella noden. Vår andra metod 'lookup' kan anropas rekursivt i liknande fall då den eftersökta nyckeln råkar vara mindre eller större än den nuvarande nyckeln, då den går vidare till den vänstra eller högra grenen av trädet och upprepar processen rekursivt där. Om rätt nod inte har hittats eller det inte finns fler noder att utforska som matchar den angivna nyckeln, returnerar metoden 'null'.

Iterator

För att möjliggöra itereringen på det binära trädet har vi implementerat gränssnittet iterable, som importerats från javas bibliotek. Till iterationen har vi även haft bra nytta utav javas stack-implementation, där vi med gränssnittets hjälpmetoder next() och hasNext() kan effektivt lagra, hämta eller ta bort en nod under en iteration.

För att använda iteratorn börjar vi med att skapa en ny instans av 'TreeIterator' klassen, varav klassens konstruktor väcks och träds till kraft. Konstruktorn tar ett argument 'root', vilket är en referens till rotnoden av det binära trädet vi vill iterera oss igenom. Sedan tilldelar vi denna rotnod till klassattributen 'next' i den nya TreeIterator-instansen, vilket denna används för att hålla reda på nästa nod som ska returneras när du anropar next()-metoden. Sedan skapar vi en ny tom stack för att hjälpa till med iterationen genom trädet, genom att spåra vilka noder som ska besökas i trädet under iterationen. När vi rör oss ner i en viss riktning i trädet, lägger vi till dess barnnoder från vänster respektive höger gren i stacken. När vi sedan behandlar noden (t.ex. hämtar dess värde med next()), tar vi bort noden från stacken och går sedan vidare till dess högra barn (om det

finns en). Denna typ av ordning som vi har valt att utforska (iterera) oss igenom trädet kallas för 'depth-first traversal'.

```
  public class TreeIterator implements Iterator<Integer>{
    private Node next;
    private Stack<Node>stack;

    public TreeIterator(Node root){
        this.next = root;
        this.stack = new Stack<>();
    }
  }
```

Efter konstruktorn exekveras de implementerade metoderna 'hasNext()' och 'next()' som vi överskrider med vår anpassade version enligt följande:

```
@Override
public boolean hasNext() {
    return next != null || !stack.isEmpty();
}
```

Metoden returnerar ett booleskt värde, där det returneras 'true om det finns fler element att iterera igenom, om inte returneras det 'false'.

```
@Override
public Integer next() {
    if(!hasNext()) {
    throw new NoSuchElementException
    }

    while(next != null) {
    stack.push(next);
    next = next.left;
    }
    Node current = stack.pop();
    next = current.right;
    return current.value;
}
```

Först kontrollerar vi om det finns ett nästa element i det binära trädet genom att anropa 'hasNext()'.Om det inte finns något nästa element, kastas ett undantag 'NoSuchElementException' eftersom att vi har nått slutet av trädet. Vi vill röra oss så långt till vänster i trädet som möjligt, vilket görs med while-loop som följer. Loopen körs så länge 'next' inte är null. När while-loopen har avslutats, sparar vi dess värde i 'current', och uppdaterar sedan vår next-pekare till att peka åt höger för att se om det finns en

förgrening, dvs. en barnnod. Om det finns innebär det att det finns större värden i trädet som ska besökas. Tillsist returneras värdet av den nod som behandlades, vilket är det minsta värdet i trädet i sekvensen genom att skriva 'current.value;'. Sammantaget kommer dessa metoder för stackimplementationen till användning för att hålla reda på noderna längs vägen och returnera nodernas värden enligt nycklarnas ordning i sekvensen.

Resultat

Resultaten utav exekveringstiden medges i tabell 1 för metoden lookup för olika ingångsstorlekar.

storlek	**lookup**
100	0.18
200	1.0
400	0.1
800	0.5
1600	0.3
3200	0.9
6400	0.6
12800	1.1
25600	1.5
51200	2.3

Table 1: Första kolumnen visar olika storlekar för det binära trädet i stigande ordning, medan andra kolumnen visar exekveringstiden för varje storlek givet i millisekunder.

Förväntade tidskomplexiteten för lookup metoden i vår binära implementation ligger i genomsnitt på

$$O(logn)$$

, där n är antalet noder i det binära trädet. Beroende på om trädet är balanserad eller obalanserad kan orsaka en märkbar skillnad i prestandan hos search-funktionen. Obalanserad träd innebär att elementen har blivit infogade i slumpmässig eller specifik ordning som leder till obalans, vilket kan ske om vi skulle ha ordnat nycklarna i t.ex. en ökande eller fallande ordning, vilket skulle ge oss en tidskomplexitet

$$O(n)$$

Olog(n) gäller om vi har balanserad träd, vilket inte garanterat motsvarar vårt fall då vi inte såg till att trädet var balanserad innan vi utförde search-funktionen. Trots detta fick vi inte alltför avvikande värden som kunde

indikera en felaktig implementation av 'lookup'- respektive 'add' metoden, eller en alltför obalanserad fall i olyckligt fall. Detta är förväntat då det indikerar att vi inte har en helt obalanserad träd, eftersom både bästa och värsta fall för ett balanserad träd ligger på O(log n) i båda fall. En närmare analys av tabellen då vi dubblar ingångsstorleken (t.ex. från 100 till 200 eller från 200 till 400) kan vi se att tiden ökar respektive minskar inte bara lite utan avsevärt. Detta motsvarar inte en linjär ökning i exekveringstiden med ökad ingångsstorlek, vilket skulle vara karakteristisk för O(n) komplexitet. Tar vi hänsyn till att trädet garanteras att inte vara helt balanserad (då vi har tillämpat slump-generatorn vid infogningen av nya noder och värdet på nycklarna med det konstanta räckvidden [0, size-1]), kan vi se att den gradvisa ökningen är indicerande för en logaritmisk ökning, vilket är karakteristiskt för O(log n) komplexitet. Tidskomplexiteten för ett obalanserat träd liknar bästa fallet för en länkad lista. I detta scenario är varje nod kopplad direkt till den föregående noden (antingen till vänster eller höger) och saknar strukturell balans. Tidskomplexiteten för lookup-metoden lutar sig till att likna en linjär sökning då vi i värsta fall måste gå igenom noderna en efter en för den sökta elementet. Fördelen med ett binärt träd när det är just att det kan reducera söktiden avsevärt genom att eliminera delar av trädet som inte innehåller det sökta elementet, vilket gör den effektiv för stora träd. Däremot kan stora träd leda till hög minnesanvändning, då varje nod i trädet kräver extra minnesuttrymme för att lagra dess nyckel, värde och referenser till barnnoder. Vi har lärt oss att binära träd kan vara ett utmärkt val för en effektiv sökoperation samt som sorteringsalgoritm när det kommer till stora data, förutsatt att trädet förutsatt att det binära trädet är balanserat.