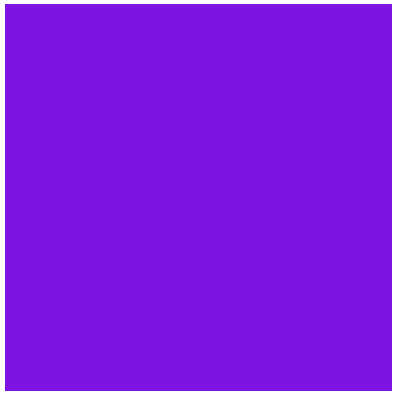
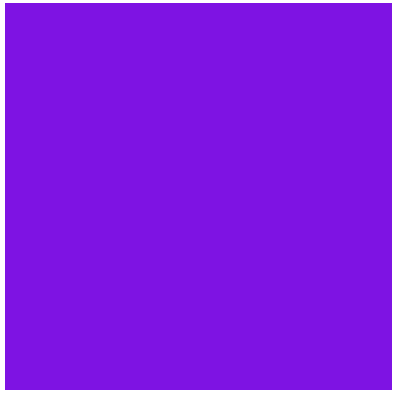


Bomen en grafen

Les 4



Binaire zoekbomen

Definitie

Een binaire zoekboom of gesorteerde binaire boom (binary search tree, vaak afgekort tot BST) is een speciaal geval van een binaire boom waarbij de data in de boom voldoen aan de eigenschap:

> *Voor elke knoop in de boom geldt dat zijn waarde strikt groter is dan alle waarden in zijn linkersubboom en strikt kleiner is dan alle waarden in zijn rechtersubboom.*

Dit impliceert uiteraard dat de data met elkaar kunnen vergeleken worden
→ ***E extends Comparable<E>*** *(zie OOP)*

Als E data1, data2 dan

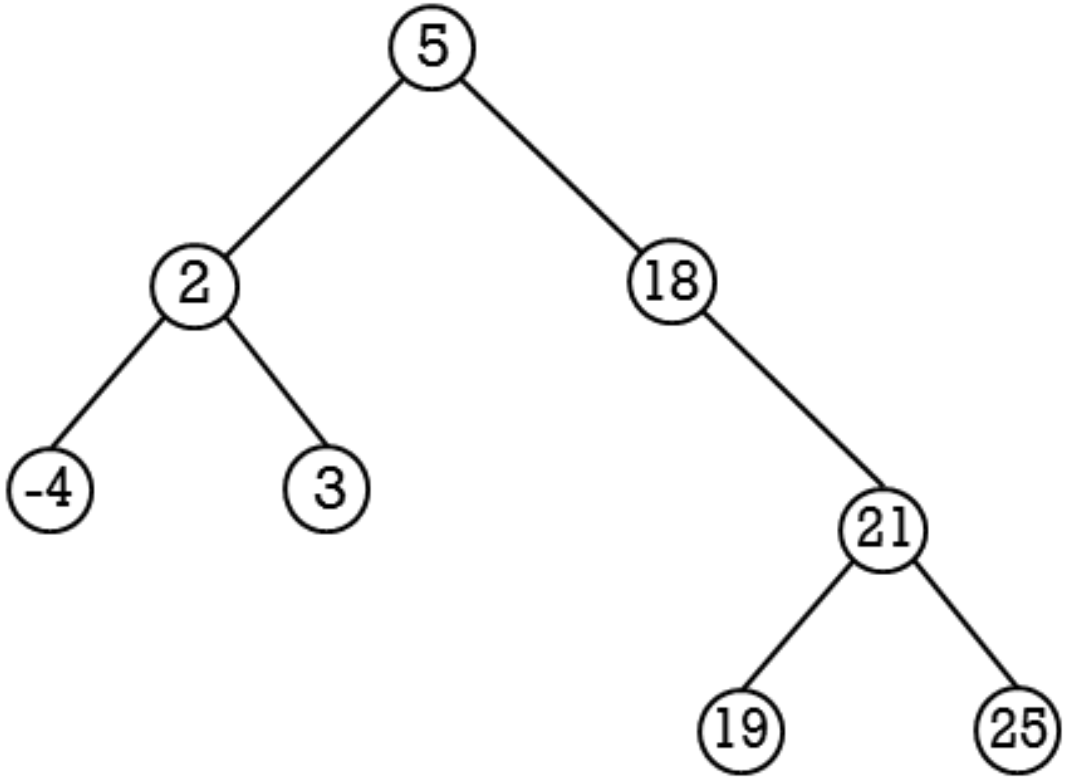
data1.compareTo(data2)

== 0 als ***zelfde***

> 0 als data1 ***is groter*** *dan data2*

< 0 als data1 ***is kleiner*** *dan data2*

BST voorbeeld

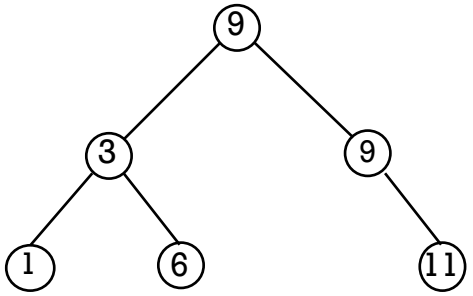


https://visualgo.net/bst

Oefening

Zijn onderstaande bomen een binaire zoekboom?

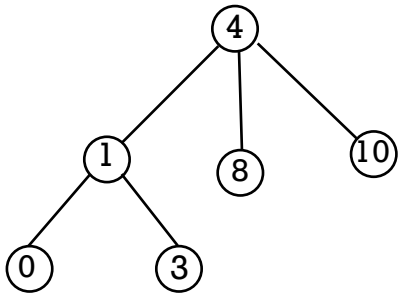
a)



Nee, 9 is niet strikt groter dan 9.

Dit impliceert dat er geen duplicaten mogen voorkomen in een BST!

b)



Nee, een binaire zoekboom moet in de eerste plaats een binaire boom zijn.
Dit is niet het geval omdat de wortel meer dan 2 kinderen heeft.

BST

- We vertrekken bij de creatie van een BST van een lege BST (het dataveld is hier dan leeg) en voegen vervolgens elementen toe

- we wensen volgende operaties te voorzien voor een BST:
  - BST() → constructie van een "lege" BST
  - printInOrder() → schrijft de data-velden gesorteerd uit
  - addNode(data): boolean → voegt een nieuwe knoop met data toe aan de BST en geeft terug of dit al dan niet gelukt is
  - lookup(data) : boolean → geeft terug of er een knoop met gegeven dataveld aanwezig is in de BST
  - removeNode(data): boolean → verwijdert de knoop met gegeven dataveld indien deze voorkomt en geeft terug of dit gelukt is

BST implementatie tot nu toe


```
public class BinarySearchTree<E extends Comparable<E>> {
        private E data;
        private BinarySearchTree<E> leftTree, rightTree;

        public BinarySearchTree(){
        }

        public void printInOrder() {
                if (this.data!=null){
                        if (leftTree!=null) leftTree.printInOrder();
                        System.out.print(this.data + " ");
                        if(rightTree!=null) rightTree.printInOrder();
                }
        }
}
```

BST implementatie tot nu toe

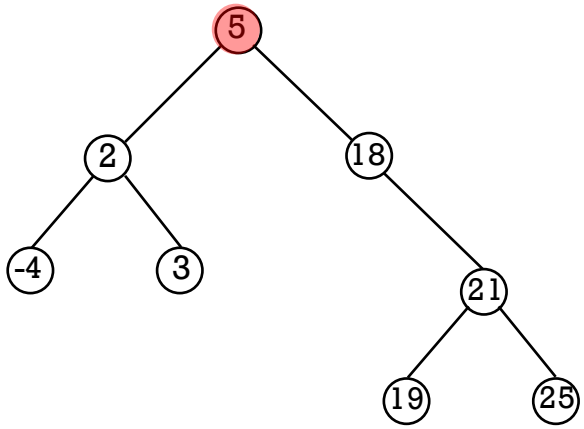

```
public class BinarySearchTree<E extends Comparable<E>> {
        private E data;
        private BinarySearchTree<E> leftTree, rightTree;

        public BinarySearchTree() {
        }

        public void printInOrder() {
                if (this.data != null) {
                        if (leftTree != null)
                                leftTree.printInOrder();
                        System.out.print(this.data + " ");
                        if (rightTree != null)
                                rightTree.printInOrder();
                }
        }
  }
```

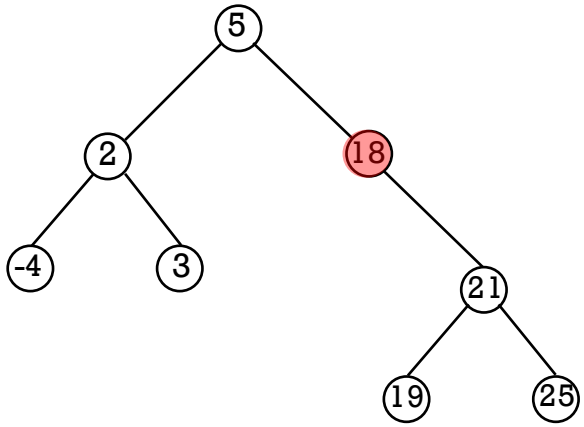
Zoeken in een binaire zoekboom

- Voorbeeld: bevindt 20 zich in onderstaande zoekboom?
  - 20 is niet gelijk aan 5. Omdat 20 > 5 onderzoeken we de rechtersubboom.



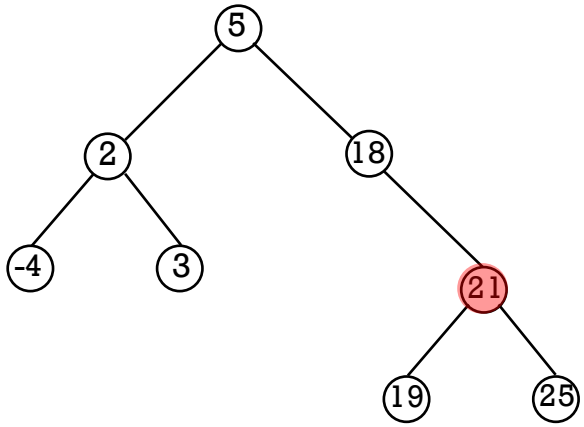
Zoeken in een binaire zoekboom

- Voorbeeld: bevindt 20 zich in onderstaande zoekboom?
  - 20 is niet gelijk aan 5. Omdat 20 > 5 onderzoeken we de rechtersubboom.
  - 20 is niet gelijk aan 18. Omdat 20 > 18 onderzoeken we de rechtersubboom.



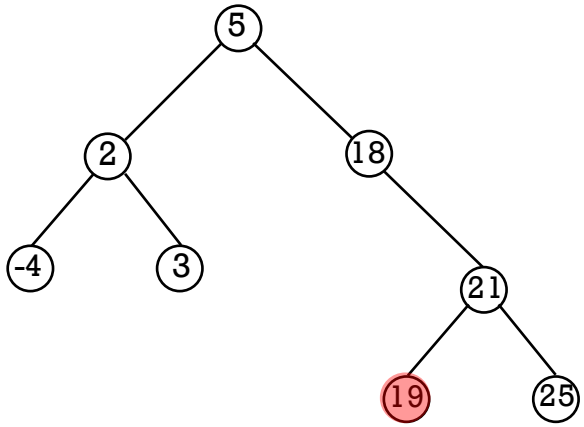
Zoeken in een binaire zoekboom

- Voorbeeld: bevindt 20 zich in onderstaande zoekboom?
  - 20 is niet gelijk aan 5. Omdat 20 > 5 onderzoeken we de rechtersubboom.
  - 20 is niet gelijk aan 18. Omdat 20 > 18 onderzoeken we de rechtersubboom.
  - 20 is niet gelijk aan 21. Omdat 20 < 21 onderzoeken we de linkersubboom.



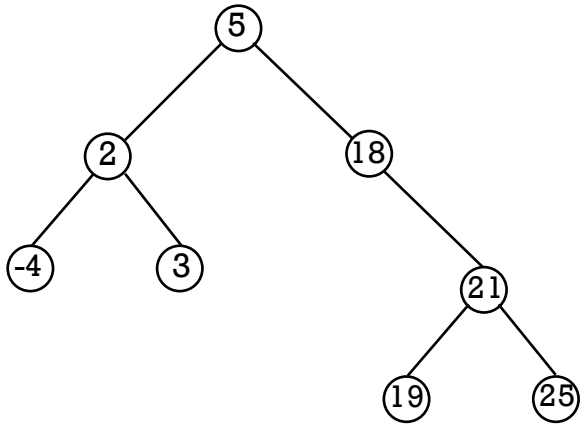
Zoeken in een binaire zoekboom

- Voorbeeld: bevindt 20 zich in onderstaande zoekboom?
  - 20 is niet gelijk aan 5. Omdat 20 > 5 onderzoeken we de rechtersubboom.
  - 20 is niet gelijk aan 18. Omdat 20 > 18 onderzoeken we de rechtersubboom.
  - 20 is niet gelijk aan 21. Omdat 20 < 21 onderzoeken we de linkersubboom.
  - 20 is niet gelijk aan 19. Omdat 20 > 19 onderzoeken we de rechtersubboom.



Zoeken in een binaire zoekboom

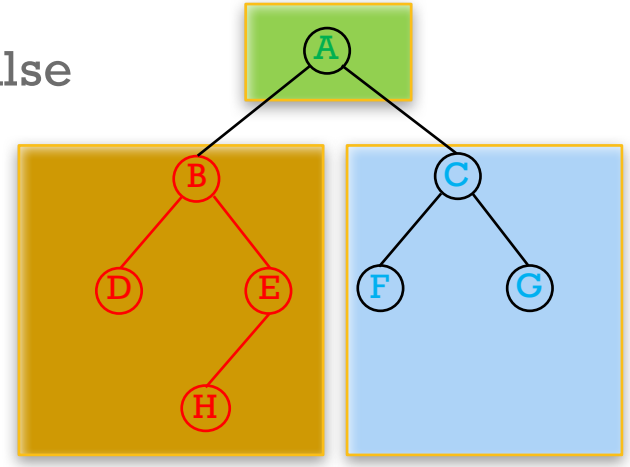
- Voorbeeld: bevindt 20 zich in onderstaande zoekboom?
  - 20 is niet gelijk aan 5. Omdat 20 > 5 onderzoeken we de rechtersubboom.
  - 20 is niet gelijk aan 18. Omdat 20 > 18 onderzoeken we de rechtersubboom.
  - 20 is niet gelijk aan 21. Omdat 20 < 21 onderzoeken we de linkersubboom.
  - 20 is niet gelijk aan 19. Omdat 20 > 19 onderzoeken we de rechtersubboom.
  - Omdat deze rechtersubboom leeg is, bevindt 20 zich niet in deze boom.



Zoeken in een binaire zoekboom

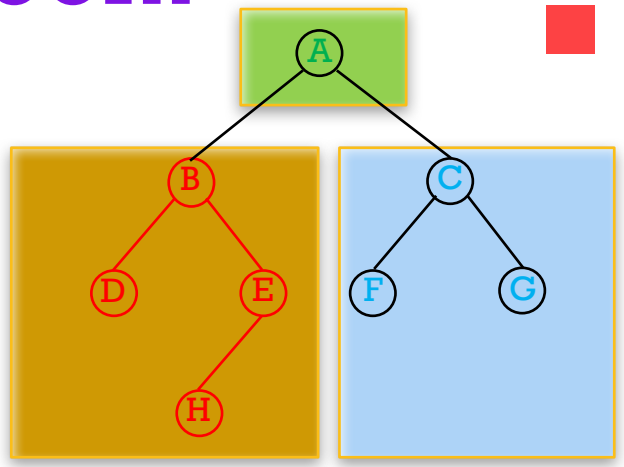
Recursieve implementatie *lookup(data): boolean*

- Uitzondering: data == null
- Basisgeval:
  - boom is leeg : return false
- Algemene regel:
  - wortel == data: return true
  - als data < wortel: zoek in linkerBoom
  - als data > wortel: zoek in rechterBoom
  - als geen linkerBoom of geen rechterBoom: return false



Zoeken in een binaire zoekboom

Recursieve implementatie *lookup(data): boolean*




```
als (data is leeg || this.data is leeg) :
        return false // geen lege data
anders:
        als (data gelijk is aan this.data): return true // gevonden
        anders:
          als (data < this.data):
                als linkerdeelboom is leeg:
                 return false;

                 anders : return linkerdeelboom.lookup(data) // kijk in linkerdeelboo

          anders:
                als rechterdeelboom is leeg:
                 return false;
                 anders: return rechterdeelboom.lookup(data) // kijk in rechterdeeli
```

Zoeken in een binaire zoekboom

Recursieve implementatie:

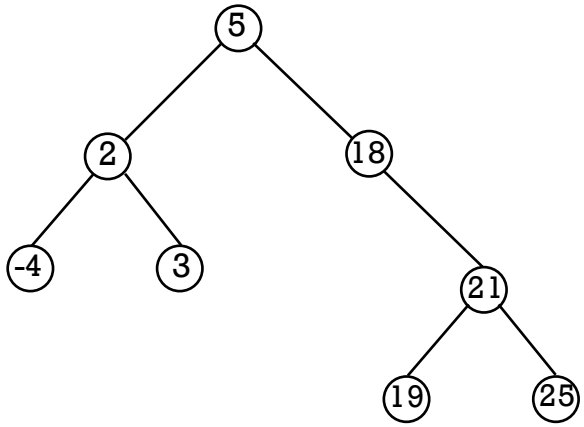
```
public boolean lookup(E data) {
  if (data == null || this.data == null)
        return false;
  else {
     if (data.compareTo(this.data) == 0)
        return true;
     else if (data.compareTo(this.data) < 0) {
        return (this.leftTree == null ? false : leftTree.lookup(data));
     }
     else {
        return (this.rightTree == null ? false:rightTree.lookup(data));
                }
        }
}
```

Zoeken in een binaire zoekboom

- Voorbeeld: bevindt 20 zich in onderstaande zoekboom?
  - 20 is niet gelijk aan 5. Omdat 20 > 5 onderzoeken we de rechtersubboom.
  - 20 is niet gelijk aan 18. Omdat 20 > 18 onderzoeken we de rechtersubboom.
  - 20 is niet gelijk aan 21. Omdat 20 < 21 onderzoeken we de linkersubboom.
  - 20 is niet gelijk aan 19. Omdat 20 > 19 onderzoeken we de rechtersubboom.
  - Omdat deze rechtersubboom leeg is, bevindt 20 zich niet in deze boom.

We moeten maximaal (in dit geval exact) 4 knopen bezoeken om de lookup methode uit te voeren.

Dit komt overeen met de diepte van de boom.

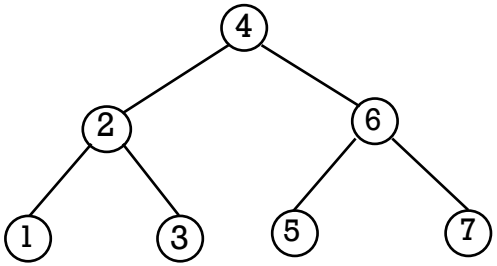


De meeste operaties op een BST vergen een tijd die rechtstreeks afhangt van de diepte van de boom.

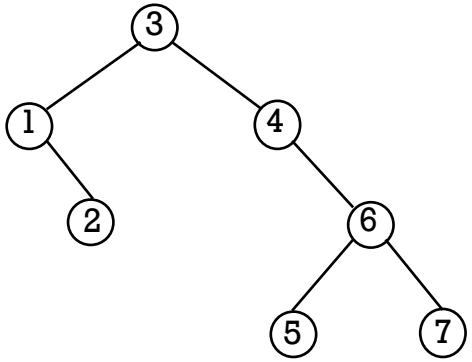
Het is dus van belang om deze diepte zo klein mogelijk te houden!

Voorbeeld

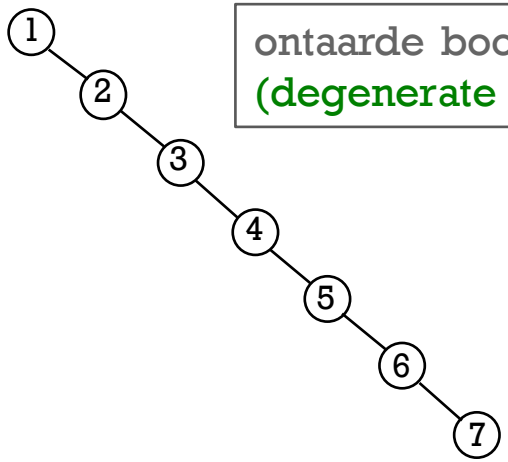
Veronderstel dat we 7 integers willen opslaan in een BST.



diepte = 3



diepte = 4



ontaarde boom
(degenerate tree)

diepte = 7

meer gebalanceerd

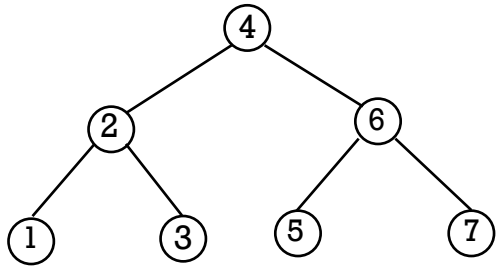
We willen gebalanceerde BSTs zodanig dat hun diepte klein is
en bijgevolg de operaties op deze BST efficient worden uitgevoerd.

Voorbeeld

Blijven BSTs niet van nature min of meer gebalanceerd?

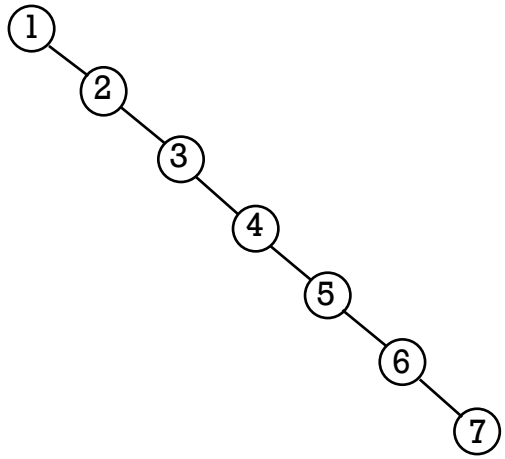
Neen, dit hangt heel sterk af van de volgorde waarin waarden worden toegevoegd of verwijderd.

Toevoegen van 4 2 6 1 3 5 7
levert



diepte = 3

Toevoegen van 1 2 3 4 5 6 7
levert



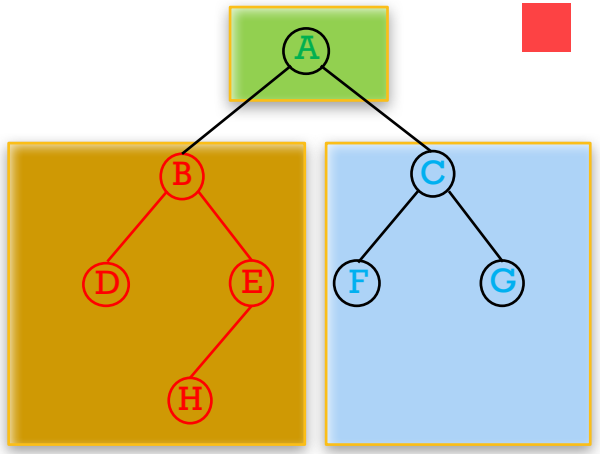
diepte = 7

Zelf-balancerende BST

Datastructuren die een BST implementeren zodanig dat zijn diepte automatisch klein blijft.

- 2-3 tree
- AA tree
- AVL tree
- Red-black tree
- Scapegoat tree
- Splay tree
- Treap

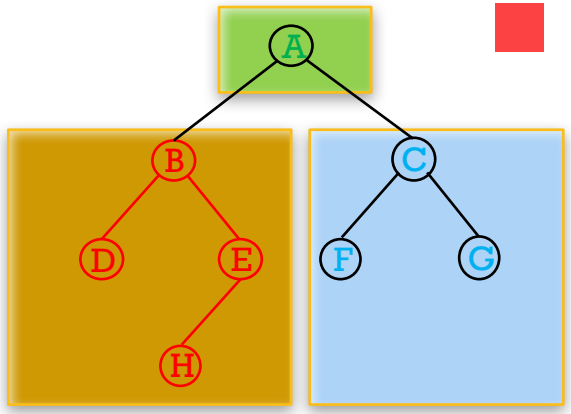
Toevoegen aan een binaire zoekboom



Recursieve implementatie: addNode(data) : boolean

- Uitzondering: data == null
- Basisgeval
  - Boom is leeg: maak boom met één blaadje (nl data)
- Algemene regel
  - Wortel == data : return false (want data mag niet dubbel voorkomen)
  - Als data < wortel:
    - Als linkerBoom is leeg: maak linkerBoom met wortel = data
    - Anders: linkerBoom .addNode(data)
  - Als data > wortel:
    - Als rechterBoom is leeg: maak rechterBoom met wortel = data
    - Anders: rechterBoom .addNode(data)

Toevoegen aan een binaire zoekboom



Recursieve implementatie: addNode(data) : boolean

- als data is leeg return false;
- als this.data is leeg dan :
  - this.data = data // in lege boom toevoegen wordt boom met 1 blad
  - return true;
- anders:
  - als data gelijk aan this.data:
    - return false *// data komt al voor*

    anders:
    - als data < this.data:

      als linkerboom is leeg :

      linkerboom = boom met 1 blad nl data

      return true

      anders: return voeg data toe aan linkerboom
    - anders:

      als rechterboom is leeg:

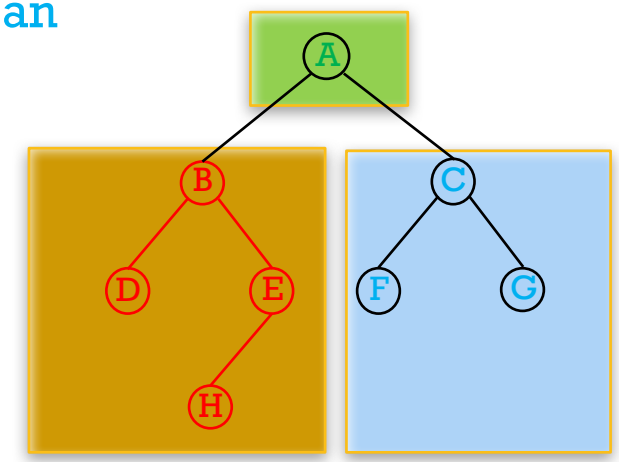
      rechterboom = boom met 1 blad nl data

      return true

      anders: return voeg data toe aan rechterboom

Verwijderen uit een binaire zoekboom

Recursieve implementatie: removeNode(data) : boolean

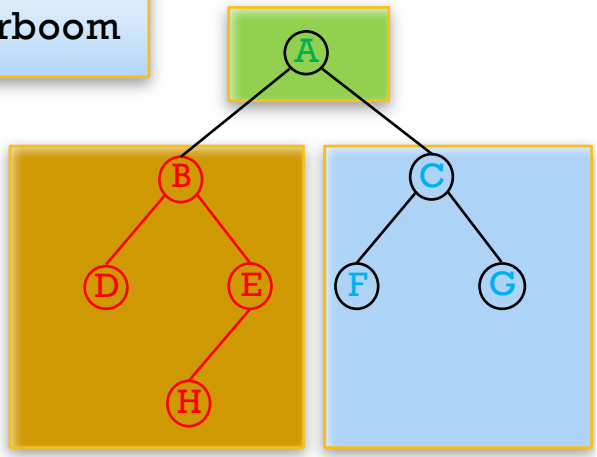


- Uitzondering: data == null
- Basisgeval
  - Boom is leeg: return false
- Algemene regel
  - Wortel == data : verwijder knoop
  - Als data < wortel:
    - Als linkerBoom is leeg: return false
    - Anders: linkerBoom .removeNode(data)
  - Als data > wortel:
    - Als rechterBoom is leeg:return false
    - Anders: rechterBoom .removeNode(data)

Verwijderen uit een binaire zoekboom

Recursieve implementatie: removeNode(data) : boolean

```
als (data is leeg of this.data is leeg) : // lege boom
        return false // data komt niet voor
anders:
        als data gelijk aan this.data:
                gevonden, zie volgende slide *
         anders:
                als data < this.data :
                        als linkerboom is leeg return false;
                        anders return verwijder data uit linkerboom
                anders:
                        als rechterboom is leeg return false
                        anders return verwijder data uit rechterboom
```



Verwijderen uit een binaire zoekboom

Recursieve implementatie: removeNode(data) : boolean

* gevonden dus this.data is gelijk aan data

als (this.isLeaf()) :*// boom bestaande uit 1 knoop met gevonden data*
- this.data= null
- return true

anders:*// of linker- of rechterdeelboom is niet leeg*
als (leftTree != null):*//linkerdeelboom is niet leeg*
- zoekGrootsteDataLinks → gl
- vervang this.data door gl
- verwijder gl uit linkerdeelboom → blaadje moet nog opgeruimd worden

return true

anders:*// rechterdeelboom is niet leeg*
- zoekKleinsteDataRechts → kr
- vervang this.data door kr
- verwijder kr uit rechterdeelboom → blaadje moet nog opgeruimd worden
- return true

Opm: door deze operatie kunnen er bladeren ontstaan met datavelden leeg,
Deze moeten dan nog opgeruimd worden. Zie oefeningen.


Vragen?