

Artemis 6.4.0

Course Overview



ge64baw

Courses > Praktikum: Grundlagen der Programmierung WS22/23 > Exercises > W10H03 - Bis Zur Unendlichkeit

W10H03 - Bis Zur Unendlichkeit

Hausaufgabe

Hard

Submission due: 7 months ago

Complaint due: 7 months ago

Points: 4 of 6

Assessment: automatic ?

Resume practice in exercise

66.67% (7 months ago) GRADED

Recent results:



Show all results

Tasks:

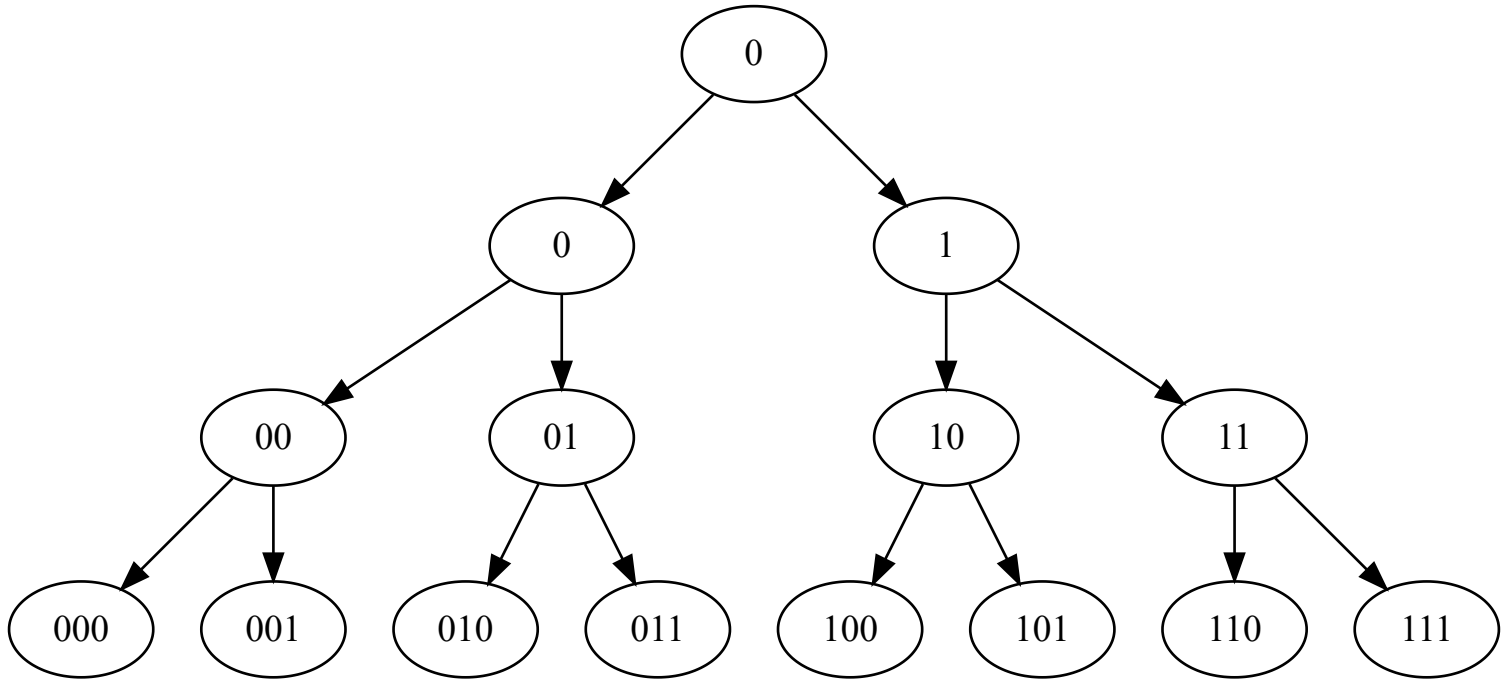
Bis zur Unendlichkeit

Die Pinguinküken haben zu Weihnachten das Spiel [Vier gewinnt](#) bekommen und sind sehr begeistert. Allerdings verlieren sie immer gegen die älteren Pinguine, die einfach vorausschauender spielen. Da sie ja schon ein bisschen programmieren können, haben sie daher angefangen, das Spiel selber zu programmieren, um damit einen guten Zug zu berechnen. Nachdem sie sich dann noch ein bisschen über Graphalgorithmen informiert haben, entscheiden sie sich für DFS (depth first search oder auch [Tiefensuche](#)), um ihren nächsten Zug zu finden. Aber gerade als sie einen Spielbaum mit allen möglichen Zugkombinationen aufbauen wollen, fällt ihnen auf: Schon nach 12 Zügen gibt es 13.793.336.634 mögliche Spielverläufe (inklusive solcher, wo weiter gespielt wird, obwohl gewonnen wurde). Wenn man von 12 Byte pro Zugkombination ausgeht, sind das 165.5 Gigabytes, die passen unmöglich in ihren Arbeitsspeicher. Kannst Du ihnen helfen?

Unendliche Bäume

Unendliche Bäume sind Datenstrukturen, welche ihre Knoten und Kanten nur berechnen und speichern, wenn dies wirklich nötig ist. Über die restlichen Knoten und Kanten wird nur gespeichert, wie man sie theoretisch erzeugen kann. Ein Beispiel hierfür wäre der Baum, welcher 0 in der Wurzel hat und in der n -ten Ebene alle möglichen Binärzahlen mit n -Bits speichert. Die Kinder eines Knotens mit dem Wert i sind nun immer $i * 2$ und $i * 2 + 1$. So müssen wir nur die Wurzel sowie die Berechnungsvorschrift abspeichern und können trotzdem jeden Wert im Baum bei Bedarf darstellen. Für diese Aufgabe sollt Ihr einen unendlichen Baum implementieren, wobei jeder Knoten des Baumes $[0, \infty)$ Kinder hat.

Der unendliche Binärzahlbaum, wenn man die ersten drei Schichten berechnet.



Die Aufgabe

Hinweise

- Die bereits definierten Methodensignaturen und Attribute dürfen nicht verändert werden.
- Ihr dürft beliebig Hilfsmethoden und Attribute hinzufügen.
- Immer wenn die Kinder eines Knotens in einer Reihenfolge behandelt werden müssen, wird diese Reihenfolge durch den Kinder-`Iterator<T>` für den Wert des Knotens diktiert.
- Die Funktion `Function<T, Iterator<T>> children` im `InfiniteTree` ist deterministisch, daher wird für einen Wert immer genau der gleiche `Iterator` zurückgegeben, auch bei mehrmaligem Aufrufen.
- Es sollen nur dann Kinder eines Knotens berechnet werden, wenn dies wirklich nötig ist.
- `ConnectFour` und `GameComputer` ist das Vier gewinnt-Spiel der kleinen Pinguine. Hier könnt Ihr mit fertiger Implementierung das Spiel gegen den Computer spielen. **Dieser Teil ist optional und gibt keine Punkte.** Er ist nur dazu da, um zu demonstrieren, wieso man so eine unendliche

Datenstruktur vielleicht gebrauchen könnte.

InfiniteTree

- `InfiniteNode<T> withRoot(T t)`: soll die Wurzel (der höchste Knoten in einem Baum) eines Baumes erstellen, welche den Wert `t` hat. Der `parent` wird auf `null` gesetzt. Ansonsten sollen in der Methode keine weiteren Knoten des Baumes berechnet werden.

InfiniteNode

- `List<InfiniteNode<T>> getChildren()`: soll alle direkten, bisher berechneten Kinder des Knotens zurückgeben.
- `InfiniteNode<T> calculateNextChild()`: soll das nächste direkte Kind des Knotens berechnen und die entsprechende `InfiniteNode` zurückgeben.
- `void calculateAllChildren()`: soll alle direkten Kinder des Knotens berechnen.
- `boolean isFullyCalculated`: gibt `true` zurück, wenn bereits alle direkten Kinder des Knotens berechnet wurden. Sonst `false`.
- `void resetChildren()`: setzt die Berechnung aller Kinder zurück.

? **Einfache Traversierung des Baumes** No results

? **Hidden Tests** No results

Tiefensuche

Tiefensuche wird verwendet, um Graphen oder Bäume nach Werten zu durchsuchen.

Prinzipiell wird zuerst der Wert des aktuellen Knotens überprüft und dann Tiefensuche auf allen Kindern ausgeführt. Für einen endlichen Baum funktioniert Tiefensuche also wie folgt:

```
void dfs(Node node) {
    check(node.value);
    for(Node child : node.children) {
        dfs(child);
    }
}
```

Da unsere Bäume aber potentiell unendlich sind, müssen einige Besonderheiten beachtet werden.

- Da wir sonst unendlich lang suchen würden, müssen wir die maximale Tiefe der Suche beschränken. `maxDepth` gibt daher an, wie viele Ebenen unterhalb des Startknotens (ausschließlich der Ebene des Startknotens selbst) durchsucht werden sollen. Ihr könnt davon ausgehen, dass `maxDepth` immer größer gleich 0 ist.
- Da die Suche beschränkt ist, kann es sein, dass wir den gesuchten Wert nicht finden, aber trotzdem gerne einen Anhaltspunkt hätten in welche Richtung wir weiter suchen müssen. Daher verwenden wir ein `Optimizable`. Diesem kann mit der Methode `boolean process(T t)` ein Wert übergeben werden und er gibt `true` wieder, wenn dies der gesuchte Wert ist, sonst `false`. Mit `T getOptimum()` bekommt man den zu diesem Zeitpunkt der Suche optimalen Wert.
- Wenn `process(t)` auf einen Wert `true` zurückgibt, soll die Suche abgebrochen und der gesuchte Wert zurückgegeben werden. Ansonsten soll am Ende der Suche das Optimum zurückgegeben werden. (Wenn `process` einmal `true` zurückgegeben hat, dann ist `getOptimum() == gesuchter Wert`).
- Teilbäume, welche bereits durchsucht wurden, nehmen nur unnötig Speicherplatz weg und sollten wieder entfernt werden. Genauso sollen nur Teilbäume berechnet werden, wenn dies für die Suche notwendig ist.

Wie genau Ihr die Suche implementiert, ist Euch überlassen, solange die Einschränkungen beachtet werden und Ihr Tiefensuche verwendet.

? **Einfache Suche funktioniert** No results

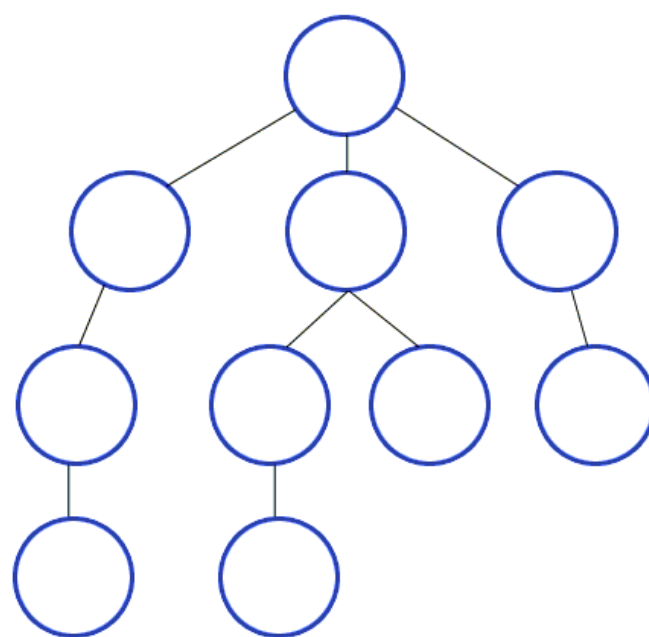
? **Hidden Tests** No results

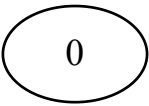
Beispiele

Um die Funktionsweise des Baumes und seiner Methoden nochmal ein bisschen anschaulicher zu erklären, folgt hier ein kleines Beispiel. Nehmen wir den Binärzahl-Baum aus der Einführung:

```
InfiniteTree<Long> tree = new InfiniteTree<Long>(i -> List.of(i * 2, i * 2 + 1).iterator());
InfiniteNode<Long> root = tree.withRoot(0L);
```

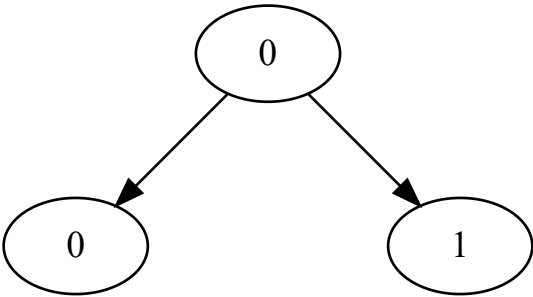
▼ `root` am Anfang, vor jeder weiteren Berechnung:





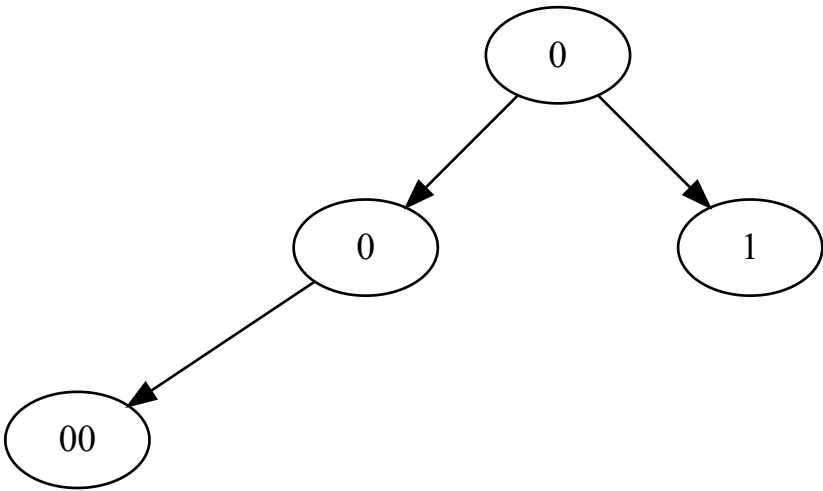
Es gilt `root.isFullyCalculated() == false`.

▼ Nach dem Aufruf `root.calculateAllChildren()`:



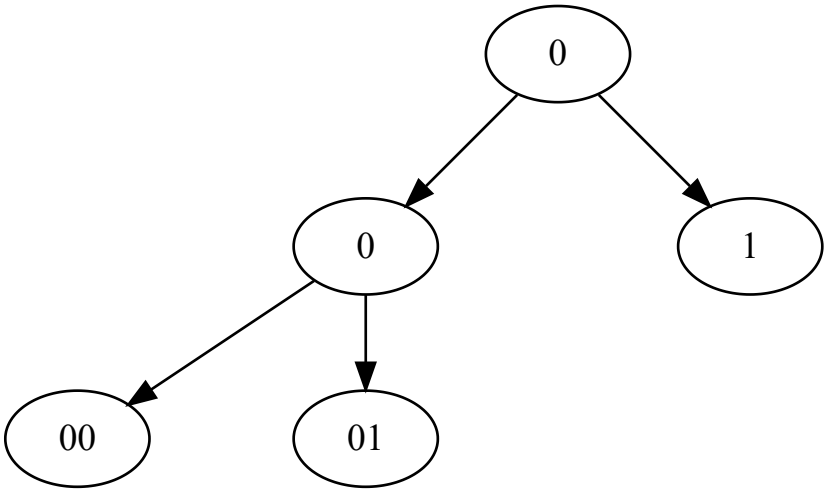
Es gilt `root.isFullyCalculated() == true`.

▼ Nach dem Aufruf `root.getChildren().get(0).calculateNextChild()`:



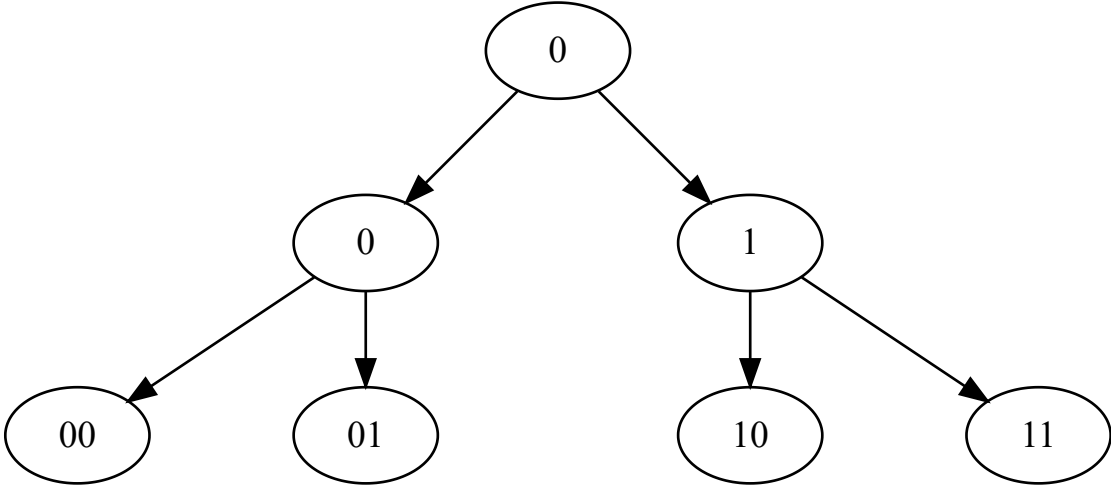
Es gilt `root.getChildren().get(0).isFullyCalculated() == false`.

▼ Nach dem Aufruf `root.getChildren().get(0).calculateNextChild()`:



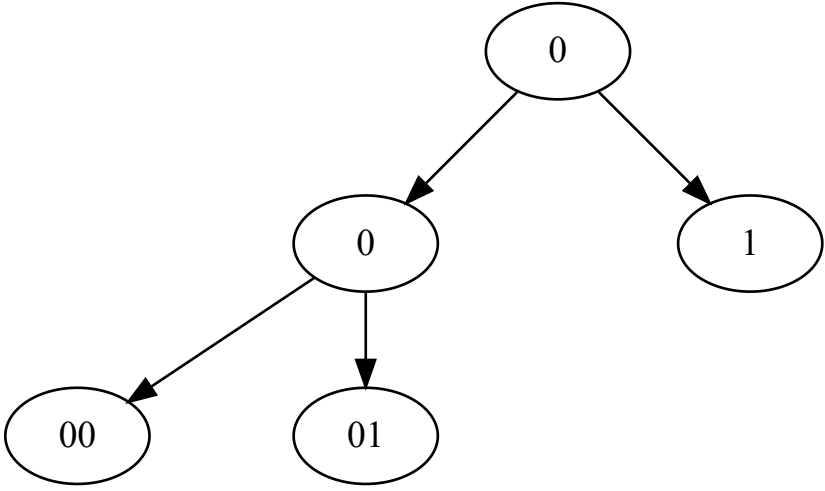
Es gilt `root.getChildren().get(0).isFullyCalculated() == true` und `root.getChildren().get(1).isFullyCalculated() == false`.

▼ Nach dem Aufruf `root.getChildren().get(1).calculateAllChildren()`:



Es gilt `root.getChildren().get(1).isFullyCalculated() == true`.

▼ Nach dem Aufruf `root.getChildren().get(1).resetChildren()`:



Exercise details

Release date:	Jan 5, 2023 18:30
Start date:	Jan 5, 2023 18:30
Submission due:	Jan 22, 2023 18:00
Complaint due:	Jan 29, 2023 18:00

Every student is allowed to complain once per exercise. In total 1000 complaints are possible in this course. You still have **998** complaints left.

