Datenbanken II

Wintersemester 2018/19

Praktische Aufgabe 1

abzugeben bis: Montag, 10. Dezember 2018, 23:55 Uhr

---

1. Allgemeines

Geben Sie bitte Ihr fertiges und *selbst geschriebenes* Programm bis spätestens **Montag, 10. Dezember 2018, 23:55 Uhr**, über das Abgabesystem [1] ab. Das System führt anschließend automatisierte Tests durch und Sie erhalten eine entsprechende Rückmeldung vom Abgabesystem.

Beachten Sie, dass Sie mindestens eine korrekte Programmieraufgabe abgeben müssen, damit Sie das Proseminar positiv absolvieren können. Eine Programmieraufgabe gilt als korrekt, sobald alle automatisch durchgeführten Tests korrekt abgearbeitet wurden. Bitte beachten Sie auch, dass nur die **letzte** Abgabe gewertet wird.

1.1. Betreuung

Bei Unklarheiten oder Verständnisproblemen gibt es für die Studierenden folgende Möglichkeiten diese abzuklären:

1. Slack Channel **#db2** [2] (bevorzugter Kommunikationskanal)
2. Tutorium (Christoph Siller, Sebastian Landl, mittwochs, 16:00 - 17:00 im SR T06)

Nehmen Sie *rechtzeitig* eine der Möglichkeiten wahr, um Fragen abzuklären. Je früher Sie sich mit der Programmieraufgabe auseinandersetzen, desto früher können Sie Fragen stellen und desto einfacher ist es für die LV-Leitung bzw. den Tutor diese zu beantworten.

---

[1] `https://abgaben.cosy.sbg.ac.at`
[2] `https://dbteaching.slack.com`

2. Aufgabenstellung

Um mit Datenmengen, die nicht vollständig in den Hauptspeicher passen, umgehen zu können, werden zur Minimierung der externen Speicherzugriffe spezielle Datenstrukturen verwendet, beispielsweise ein B^+-Baum.

In der Vorlesung werden die B^+-Baum-Operationen *Suchen*, *Einfügen* und *Löschen* besprochen. Die besprochene Einfüge-Operation bezieht sich dabei auf einzelne Schlüssel. In der Praxis wird ein B^+-Baum-Index allerdings oft auf einer bestehenden Datenmenge konstruiert. In diesem Fall kann der B^+-Baum auf eine spezielle Art konstruiert werden, die als *Bulk Loading* bezeichnet wird.

In dieser Programmieraufgabe implementieren Sie Bulk Loading für einen B^+-Baum. Um nicht mit Duplikaten umgehen zu müssen wird in dieser Programmieraufgabe angenommen, dass die Schlüssel eindeutig sind. Der Aufbau der B^+-Baum-Knoten ist genau wie in den Vorlesungsunterlagen beschrieben, d.h., jeder Knoten hat

- eine Knotengrad, m,
- eine Menge von Schlüsseln (maximal $m - 1$) und
- eine Menge von Zeigern (maximal m).

2.1. Bulk Loading

Bulk Loading hat mehrere Vorteile:

- Die Höhe des B^+-Baumes wird minimiert.
- Einzelne Einfüge-Operationen sind langsam.
- Speichern der Blattknoten auf hintereinanderliegenden Blöcken.

Damit ein minimaler B^+-Baum entsteht, wird der B^+-Baum von unten nach oben (*bottom-up*) konstruiert, d.h. zuerst wird die Blattknoten-Ebene konstruiert, dann wird die darüberliegende Ebene entsprechend konstruiert, usw. Im Folgenden ist eine mögliche Vorgehensweise für eine solche Bottom-up-Kontruktion beschrieben.

2.1.1. Blattknoten

1. Sortiere alle Schlüssel.
2. Erstelle Blattknoten und füge die ersten $(m - 1)$ Schlüssel aus der Datenmenge sortiert in den Blattknoten ein. Füge außerdem entsprechend viele `null`-Zeiger in den Blattknoten ein.
3. Erstelle weiteren Blattknoten und füge die nächsten $(m-1)$ Schlüssel aus der Datenmenge sortiert in den Blattknoten ein (mit den entsprechenden `null`-Zeigern). Dieser Schritt wird solange fortgesetzt bis alle Schlüssel der Datenmenge in Blattknoten vorhanden sind.

4. Sollte nun der letzte Blattknoten zu wenig *Schlüssel* besitzen, werden Schlüssel vom vorletzten Blattknoten umverteilt (es sind genug vorhanden, da der vorletzte Blattknoten voll ist).

5. Verbinde alle Blattknoten (für sequentiellen Scan).

2.1.2. Innere Knoten

1. Erstelle inneren Knoten und füge die ersten m Zeiger zu den ersten m darunterliegenden Knoten ein. Füge weiters die Schlüssel so ein, dass ein gültiger B^+-Baum-Knoten entsteht.

2. Erstelle weiteren inneren Knoten und füge die nächsten m Zeiger zu den nächsten m darunterliegenden Knoten ein. Füge weiters die Schlüssel so ein, dass ein gültiger B^+-Baum-Knoten entsteht. Dieser Schritt wird solange fortgesetzt bis alle darunterliegenden Knoten verbunden sind.

3. Sollte nun der letzte innere Knoten zu wenig *Zeiger* besitzen, werden Zeiger **und** Schlüssel vom vorletzten inneren Knoten umverteilt (es sind genug vorhanden, da der Vorgängerknoten voll ist).

4. Schritte $1-3$ werden solange wiederholt, bis der B^+-Baum vollständig erstellt ist, d.h. nur mehr ein innerer Knoten - der Wurzelknoten - erstellt wird.

Es gibt mehrere Varianten um einen *minimalen* **und** *gültigen* B^+-Baum für eine gegebene Schlüsselmenge zu konstruieren. Es steht den Studierenden frei eine andere Variante zu implementieren solange die folgenden Voraussetzungen erfüllt sind: Der B^+-Baum ist *gültig* **und** *minimal bzgl. der Höhe*. Diese Voraussetzungen werden auch bei den automatisierten Tests überprüft.

Voraussetzungen für einen gültigen B^+-Baum mit Grad m

- Alle Pfade von der Wurzel zu den Blattknoten haben die gleiche Länge. Ein B^+-Baum ist also ein balancierter Suchbaum.
- Jeder innere Knoten (mit Ausnahme der Wurzel) hat mindestens $\lceil \frac{m}{2} \rceil$ und maximal m Kinder.
- Hat ein Knoten $(i+1)$ Kinder, dann hat dieser Knoten i Suchschlüssel.
- Jeder Blattknoten hat mindestens $\lceil \frac{m-1}{2} \rceil$ und maximal $(m-1)$ Suchschlüssel.
- Ist die Wurzel der einzige Knoten im B^+-Baum (d.h. ein Blattknoten), dann hat die Wurzel 0 bis maximal $(m-1)$ Suchschlüssel. Gibt es mehrere Blattknoten (d.h. die Wurzel ist ein innerer Knoten), dann hat die Wurzel mindestens 2 Kinder.
- Die Suchschlüssel in einem Knoten sind aufsteigend sortiert.

Für K Suchschlüssel werden mindestens $\lceil \frac{K}{m-1} \rceil$ Blattknoten benötigt. Dadurch ergibt sich auch die minimale Höhe eines B^+-Baumes:

$$\left\lceil \log_m \left(\left\lceil \frac{K}{m-1} \right\rceil \right) \right\rceil \tag{1}$$

Konkret sollen Sie die folgenden Methoden in der Klasse `BPTree` implementieren (siehe `src/bptree/BPTree.java`):

- `List<BPNode<Key>> buildLeafLevel(final List<Key> keys)`
  - `keys`: Sortierte Liste der Schlüssel, die im B^+-Baum gespeichert werden.
  - Rückgabe: eine (sortierte) Liste aller erstellten Blattknoten.
- `List<BPNode<Key>> buildInnerLevel(final List<BPNode<Key>> children)`
  - `children`: Sortierte Liste mit den Kinder-Knoten der darunterliegenden B^+ Baum-Ebene. Für diese Kinder-Knoten soll die darüberliegende B^+ Baum Ebene konstruiert werden.
    Bsp.: Wird hier die Liste der Blattknoten übergeben, dann wird die erste innere B^+ Baum Ebene konstruiert, deren Zeiger auf die Blattknoten verweisen.
  - Rückgabe: eine (sortierte) Liste aller erstellten inneren Knoten.

Beispiele für B^+-Bäume die per Bulk Loading generiert wurden, finden Sie in Anhang A dieses Dokumentes.

3. Codegerüst

Der Algorithmus ist in Java 8 zu implementieren. Um Ihnen das Lösen der Programmieraufgabe zu erleichtern (und für automatisiertes Testen), stellen wir Ihnen ein dokumentiertes Codegerüst zur Verfügung.
Das Codegerüst besteht aus 3 Dateien, wobei Sie Ihren Code in die Datei `src/bptree/BPTree.java` einfügen müssen. Dies ist auch die einzige Datei, die Sie abgeben müssen. Alle weiteren Klassen werden im Abgabesystem automatisch kopiert (in der Version, die Sie ebenfalls erhalten haben).
Weitere Details zum Codegerüst entnehmen Sie bitte der Dokumentation.

`doc/`	Dokumentation (Javadoc)
`src/`	Source-Dateien
`test/`	Testinstanzen

Hinweis

Für die automatisierten Tests wird nur die Datei `src/bptree/BPTree.java` berücksichtigt. Alle anderen hochgeladenen Dateien werden ignoriert bzw. mit den originalen Dateien überschrieben.

3.1. Kompilieren

Sie können den Code wie folgt per Kommandozeile kompilieren, wenn Sie sich im Ordner `pa1/` befinden:

```
javac src/*.java src/bptree/*.java
```

Alternativ kann auch das beiliegende `Makefile` zum kompilieren benutzt werden:

```
make
```

4. Abgabe

Die Abgabe Ihrer Implementierung erfolgt über unser Abgabesystem [3]. Beachten Sie, dass nur die jeweils **letzte** Abgabe gewertet wird.

Hinweis

Verwenden Sie in Ihrer Implementierung keinesfalls Sonderzeichen oder Umlaute. Dies kann zu unvorhergesehenen Fehlern führen und würde eine negative Bewertung nach sich ziehen.

4.1. Testinstanzen

Im Ordner `test/` werden Ihnen 10 Testinstanzen zur Verfügung gestellt. Diese Testinstanzen sollen es Ihnen erleichtern Ihre Implementierung zu testen. Ebenso sind die entsprechenden Resultate in den jeweiligen Unterordnern verfügbar. Verarbeitet Ihr Programm diese Testinstanzen korrekt, heißt das aber nicht zwangsläufig, dass Ihr Programm alle Eingaben korrekt behandlet. Wir empfehlen deshalb mit zusätzlichen Testinputs zu arbeiten.
Ausführen des Programms per Kommandozeile, wenn Sie sich im Ornder `pa1/` befinden.

```
java -cp src/ Main test/000x/input_000x
```

Über das `Makefile` ist es auch möglich alle Testinstanzen zu überprüfen. Dazu im Ordner `pa1/` den Befehl

```
make test
```

ausführen.

4.2. Eingabeformat

Ihr Programm nimmt einen Dateipfad als Kommandozeilenargument entgegen und liest diese Eingabedatei ein (Funktionalität bereits vorhanden siehe `src/Main.java`). Die erste Zeile einer Eingabedatei enthält den Grad m des B^+-Baumes als Integer. Alle darauffolgenden Zeilen enthalten Suchschlüssel als Integer (in beliebiger Reihenfolge). Sie können davon ausgehen, dass **alle** Testinstanzen dieser Spezifikation entsprechen.

4.3. Ausgabeformat

- Liste von B^+-Baum-Knoten sowie die Werte der Suchschlüssel und Zeiger.
- Jeder B^+-Baum-Knoten wird in einer eigenen Zeile ausgegeben.
- Knoten-IDs, Suchschlüssel und Zeiger werden per Doppelpunkt (":") getrennt. Speichert ein Knoten i Suchschlüssel, müssen auch $(i-1)$ Zeiger in der Ausgabe vorhanden sein.

[3] `https://abgaben.cosy.sbg.ac.at`

- In der letzten Zeile wird die ID des Wurzelknotens mit einem vorangestellten "r" ausgegeben.
- `null`-Zeiger werden mit 0 dargestellt.

4.4. Automatisierte Tests

Die Überprüfung Ihrer Implementierung erfolgt automatisiert. Hierbei wird zuerst überprüft, ob Ihr Programmcode kompiliert. Danach werden Tests mit verschiedenen Inputdaten durchgeführt. Der Output Ihres Programmes wird dann validiert. Daher weisen wir darauf hin, dass der Output genau so dargestellt werden muss, wie vorgegeben. Im Codegerüst existieren bereits entsprechende Methoden, die auf `stdout` schreiben. Diese können von Ihnen verwendet werden. Natürlich können Sie temporär zusätzlichen Output hinzufügen, allerdings muss dieser vor der Abgabe entfernt werden.
Sie finden den jeweiligen Beispiel-Output auch in Anhang A.
Nachdem die Tests abgeschlossen sind, bekommen Sie eine Erfolgs- bzw. Fehlermeldung über das Abgabesystem [3]. Um Serverüberlastungen vorzubeugen, empfehlen wir, die Lösung nicht erst kurz vor der Abgabe-Deadline hochzuladen, sondern rechtzeitig abzugeben.
Während der Testphase wird Ihr Programm mit folgendem Kommandozeilenbefehl aus dem Ordner `pa1` aufgerufen:

```
java Main inputfile > outputfile
```

4.5. Aufwandsschätzung

Hier finden Sie eine Zusammenfassung unserer Referenzlösung, erzeugt durch den Befehl `git log --stat=80`. Die letzte Zeile zeigt die Anzahl an eingefügten und gelöschten Zeilen.
Die Referenzlösung stellt nur eine mögliche Lösung dar. Es sind auch andere Lösungen möglich, die sich entsprechend in der Anzahl der Änderungen unterscheiden. Die Zusammenfassung der Referenzlösung dient lediglich zur Orientierung.

```
src/bptree/BPTree.java | 81 ++++++++++++++++++++---
1 file changed, 72 insertions(+), 9 deletions(-)
```

5. Zusätzliche Informationen

- Vorlesungsunterlagen (siehe LV-Website [4]).
- Kapitel 11.4.4: *Bulk Loading of B^+-Tree Indices* in *Silberschatz et al. Database Systems Concepts, 6th Edition, McGraw-Hill, 2011.*
- *Lesson: Generics* aus der Java Dokumentation (siehe [5])
- Weiterführende Fragen:

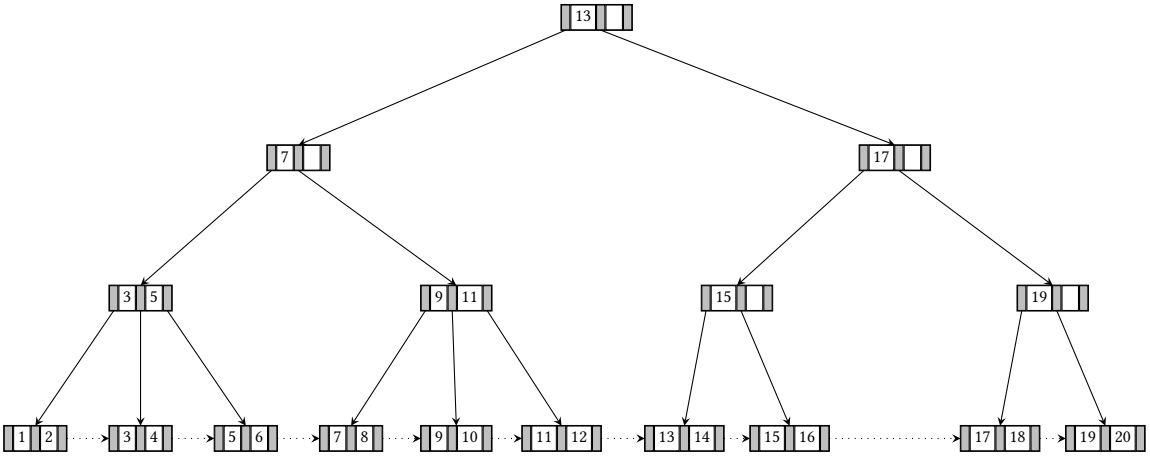
[4] `https://dbresearch.uni-salzburg.at/teaching/2017ws/db2/`
[5] `http://docs.oracle.com/javase/tutorial/extra/generics/index.html`

– Warum könnte es sinnvoll sein, die Knoten nicht ganz voll zu machen, sondern diese nur bis zu einem gewissen Füllgrad (bspw. 65%) zu befüllen?

Anhang A Beispiele

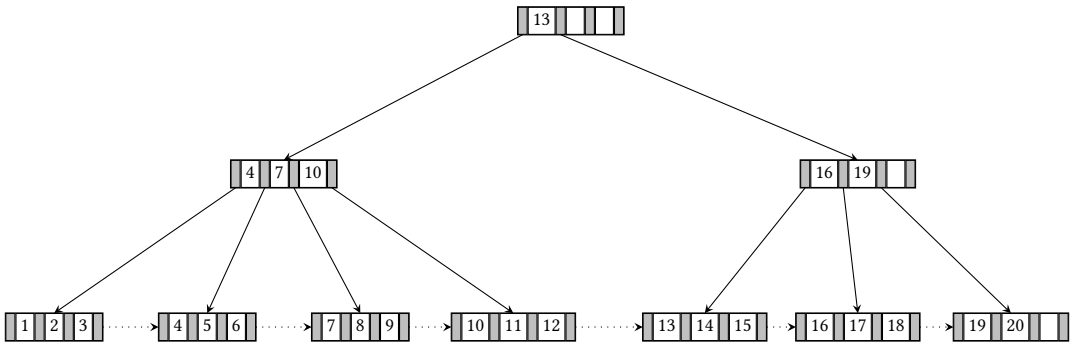
Im Folgenden finden Sie Beispiele für B^+-Bäume, die per Bulk Loading konstruiert wurden. Für alle Beispiele wurde die folgende Schlüsselmenge verwendet:

$$K = \{10, 3, 20, 1, 16, 18, 8, 9, 14, 11, 2, 15, 4, 5, 7, 13, 6, 17, 12, 19\}$$



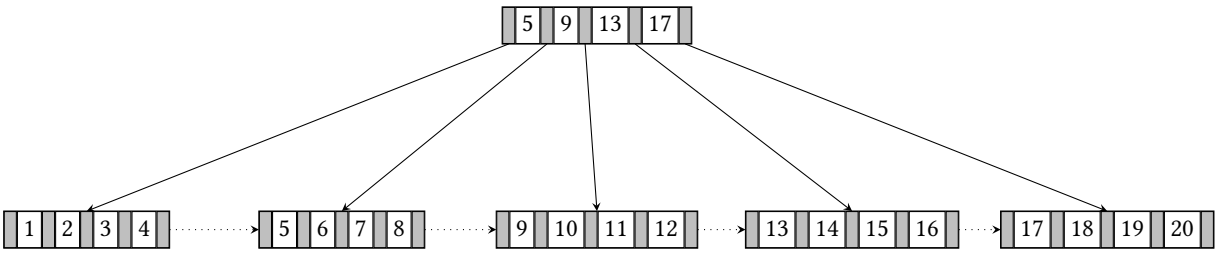

```
1:0:1:0:2:2
2:0:3:0:4:3
3:0:5:0:6:4
4:0:7:0:8:5
5:0:9:0:10:6
6:0:11:0:12:7
7:0:13:0:14:8
8:0:15:0:16:9
9:0:17:0:18:10
10:0:19:0:20:0
11:1:3:2:5:3
12:4:9:5:11:6
13:7:15:8
14:9:19:10
15:11:7:12
16:13:17:14
17:15:13:16
r17
```

Abbildung 1: $m = 3$



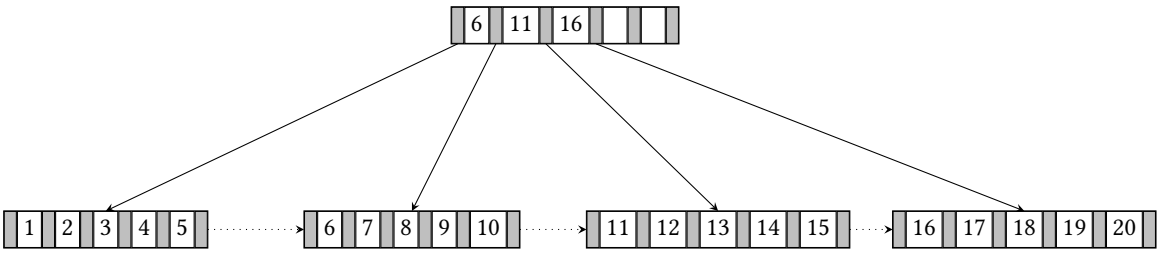

```
1:0:1:0:2:0:3:2
2:0:4:0:5:0:6:3
3:0:7:0:8:0:9:4
4:0:10:0:11:0:12:5
5:0:13:0:14:0:15:6
6:0:16:0:17:0:18:7
7:0:19:0:20:0
8:1:4:2:7:3:10:4
9:5:16:6:19:7
10:8:13:9
r10
```

Abbildung 2: $m = 4$



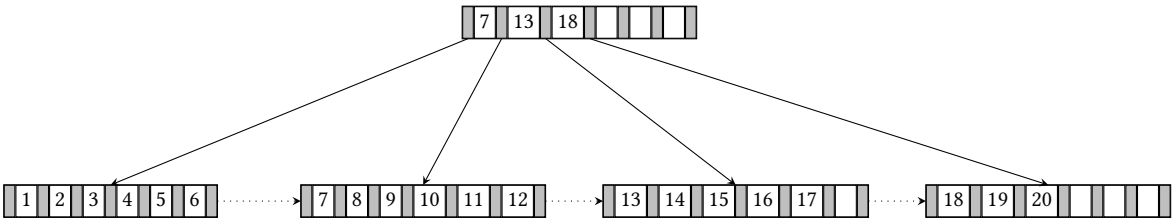

```
1:0:1:0:2:0:3:0:4:2
2:0:5:0:6:0:7:0:8:3
3:0:9:0:10:0:11:0:12:4
4:0:13:0:14:0:15:0:16:5
5:0:17:0:18:0:19:0:20:0
6:1:5:2:9:3:13:4:17:5
r6
```

Abbildung 3: $m = 5$



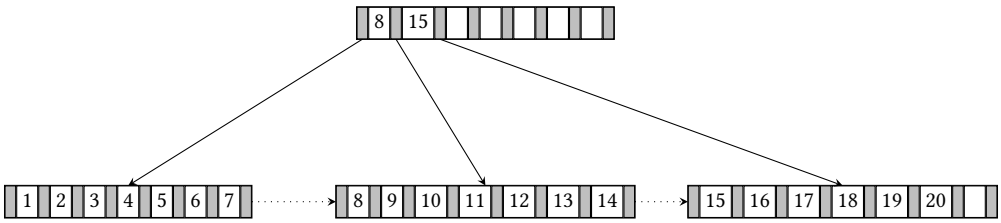

```
1:0:1:0:2:0:3:0:4:0:5:2
2:0:6:0:7:0:8:0:9:0:10:3
3:0:11:0:12:0:13:0:14:0:15:4
4:0:16:0:17:0:18:0:19:0:20:0
5:1:6:2:11:3:16:4
r5
```

Abbildung 4: $m = 6$



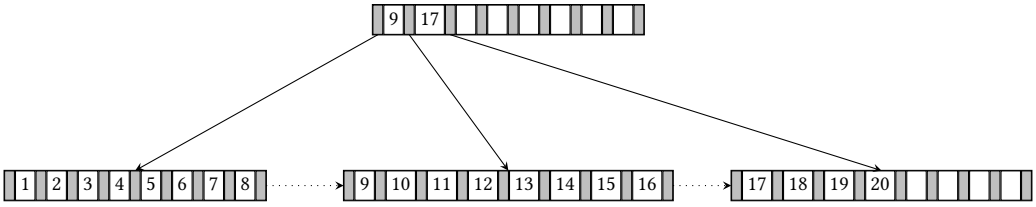

```
1:0:1:0:2:0:3:0:4:0:5:0:6:2
2:0:7:0:8:0:9:0:10:0:11:0:12:3
3:0:13:0:14:0:15:0:16:0:17:4
4:0:18:0:19:0:20:0
5:1:7:2:13:3:18:4
r5
```

Abbildung 5: $m = 7$



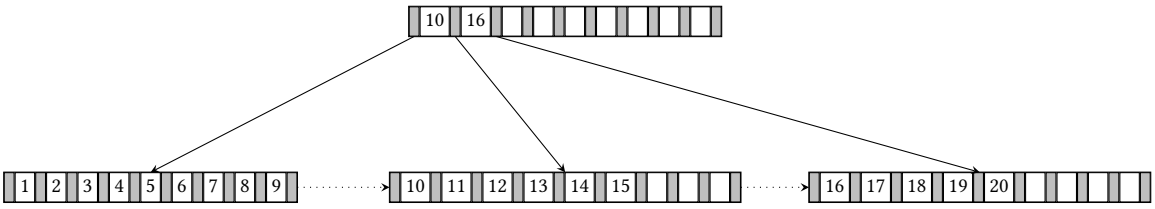

```
1:0:1:0:2:0:3:0:4:0:5:0:6:0:7:2
2:0:8:0:9:0:10:0:11:0:12:0:13:0:14:3
3:0:15:0:16:0:17:0:18:0:19:0:20:0
4:1:8:2:15:3
r4
```

Abbildung 6: $m = 8$



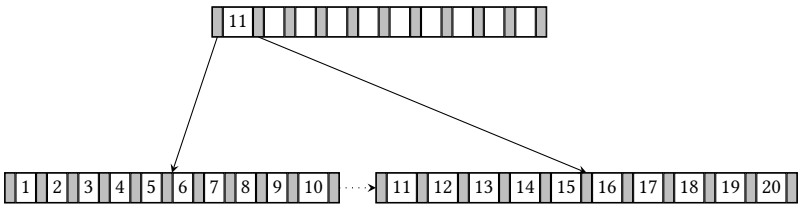

```
1:0:1:0:2:0:3:0:4:0:5:0:6:0:7:0:8:2
2:0:9:0:10:0:11:0:12:0:13:0:14:0:15:0:16:3
3:0:17:0:18:0:19:0:20:0
4:1:9:2:17:3
r4
```

Abbildung 7: $m = 9$




```
1:0:1:0:2:0:3:0:4:0:5:0:6:0:7:0:8:0:9:2
2:0:10:0:11:0:12:0:13:0:14:0:15:3
3:0:16:0:17:0:18:0:19:0:20:0
4:1:10:2:16:3
r4
```

Abbildung 8: $m = 10$




```
1:0:1:0:2:0:3:0:4:0:5:0:6:0:7:0:8:0:9:0:10:2
2:0:11:0:12:0:13:0:14:0:15:0:16:0:17:0:18:0:19:0:20:0
3:1:11:2
r3
```

Abbildung 9: $m = 11$