technische universität
dortmund

Fakultät für Informatik
Lehrstuhl 11

Jonas Ellert
Mart Hagedoorn
Guangping Li

Sommersemester 2022
17. Mai

DAP2 Praktikum – Blatt 7

Abgabe: ab 23. Mai

Studienleistung

- Zum Bestehen des Praktikums muss jeder Teilnehmer die folgenden Leistungen erbringen:
  - Es müssen mindestens 50 Prozent der Punkte in den Kurzaufgaben erreicht werden.
  - Es müssen mindestens 50 Prozent der Punkte in den Langaufgaben erreicht werden.
- Im Krankheitsfall kann ein Testat bei Vorlage eines Attests in der folgenden Woche nachgeholt werden.
- Wenn ein Praktikumstermin auf einen Feiertag fällt, müssen Sie sich an einem beliebigen anderen Praktikumstermin in der gleichen Woche testieren lassen.
- **Hinweis:** Notieren Sie sich Ihre Punkte nach jedem Testat! Dies dient der eigenen Kontrolle. (Ihr Punktestand kann Ihnen während des Semesters nicht genannt werden.)

Wichtige Information (im Moodle verfügbar)

- Beachten Sie die Erklärung des **Ablaufs (Blatt A)**.
- Beachten Sie die **Regeln und Hinweise (Blatt R)** in der aktuellsten Version!
- Beachten Sie die **Hilfestellungen (Blatt H)** in der aktuellsten Version!

Kurzaufgabe 7.1: Bitvektor-Datenstrukturen (8 Punkte)

(a) `BitVector` Klasse (3 Punkte)

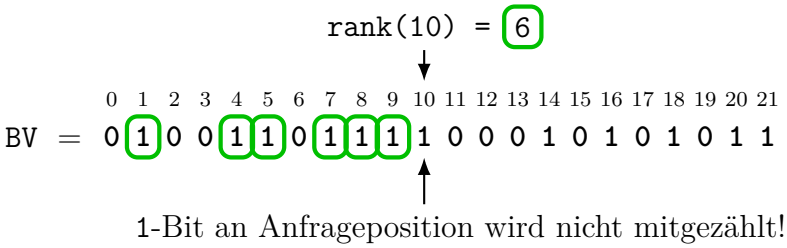
Ein Bitvektor ist eine Datenstruktur, welche zur Speicherung von n Bits (oder Wahrheitswerten) genutzt werden kann. Dabei soll das i-te Bit für jedes $i \in \{0, \ldots, n-1\}$ in konstanter Zeit sowohl ausgelesen als auch gesetzt werden können. In dieser Teilaufgabe sollen Sie eine Klasse `BitVector` implementieren, welche folgende Anforderungen erfüllt:

- Der Konstruktor erwartet einen nicht-negativen `int`-Wert n, welcher die Anzahl der zu speichernden Bits enthält. Initial sind alle Bits auf den Wert `0` gesetzt.
- Die Methode `public int size()` gibt die Länge n des Bitvektors zurück.
- Die Methode `public boolean get(int index)` gibt den Wert des Bits an Position `index` zurück.
- Die Methode `public void set(int index, boolean value)` setzt den Wert des Bits an Position `index` auf `value`.
- Abgesehen davon hat die Klasse **keine öffentlichen Methoden oder Member-Variablen**
- **Der gesamte Bitvektor sollte höchstens $n + \mathcal{O}(1)$ Bits an Speicher benötigen!** Sie können also nicht einfach ein Array `boolean [n]` verwenden, da ein solches Array pro Eintrag direkt ein ganzes Byte an Speicherplatz benötigt. In den Hinweisen am Ende des Blattes finden Sie Tipps zur effizienten Speicherung von Bitvektoren.

Den Speicherbedarf ihrer Implementierung können Sie mit dem Programm `BVMem.java` aus dem Archiv `7.1a.zip` verifizieren. Diesem übergeben Sie die Länge des Bitvektors als einziges Argument. Aufgrund von Messungenauigkeiten sollten Sie relativ große Längen ausprobieren, z.B. $n = 100000000$. Die Korrektheit können Sie mit dem Programm `BVCorrect.java` testen.

(b) `BitVectorRank` Klasse **(3.5 Punkte)**

Die Methoden `get` und `set` bieten in praktischen Anwendungen oftmals nicht genug Funktionalität. Häufig wird zusätzlich die Methode `int rank(int index)` benötigt. Diese gibt die Anzahl der `1`-Bits im Bitvektor-Interval $\{0, \ldots, \texttt{index} - 1\}$ zurück.



`1`-Bit an Anfrageposition wird nicht mitgezählt!

In dieser Teilaufgabe sollen Sie eine Klasse `BitVectorRank` erstellen, welche folgende Anforderungen erfüllt.

- Der Konstruktor erwartet ein Objekt der Klasse `BitVector` aus Aufgabe (a), sowie ein Argument c, welches später einen Trade-Off zwischen Anfragezeit und zusätzlichem Speicherbedarf kontrolliert. Der Konstruktoraufruf sollte nicht mehr als $\mathcal{O}(n)$ Zeit benötigen.

  Hinweis: Wenn Sie Aufgabe (a) nicht bearbeitet haben, können Sie einfach die naive Implementierung von `BitVector` aus dem Archiv `7.1b.zip` im Moodle verwenden.

- Die Methode `public int size()` gibt die Länge n des Bitvektors zurück.

- Die Methode `public int rank(int index)` gibt zurück, wie viele `1`-Bits im Bitvektor-Interval $\{0, \ldots, \texttt{index - 1}\}$ enthalten sind. Es gilt `rank(0) = 0`, und `rank(`n`)` gibt die Anzahl aller `1`-Bits im gesamten Bitvektor zurück. Die Beantwortung der Anfrage sollte nur $\mathcal{O}(c)$ viele Rechenschritte benötigen.

- Die Methode `public int count(int start, int end)` gibt zurück, wie viele `1`-Bits im Bitvektor-Interval $\{\texttt{start}, \ldots, \texttt{end - 1}\}$ enthalten sind. Im oben stehenden Beispiel gilt `count(2, 9) = 4`. Die Beantwortung der Anfrage sollte nur konstant viele Rechenschritte benötigen. Sie dürfen dabei die Methode `rank` zur Hilfe nehmen (auch wenn Sie diese nicht implementiert haben).

- **Das Objekt der Klasse `BitVectorRank` sollte höchstens $\left\lceil \frac{32n}{c} \right\rceil + \mathcal{O}(1)$ Bits benötigen**. Wenn z.B. $c = 3200$, dann benötigt das Objekt nur etwa 1% so viel Speicherplatz wie der Bitvektor. Es soll den eigentlichen Bitvektor also insbesondere nicht ersetzen, sondern lediglich als zusätzliche Hilfsdatenstruktur dienen. In den Hinweisen am Ende des Blattes finden Sie Tipps dazu, wie sie den geforderten Platz einhalten können.

Ihre Datenstruktur darf statisch sein: Wenn ein Bit im Bitvektor nach der Konstruktion der `rank`-Datenstruktur verändert wird, dann muss Ihre Datenstruktur diese Änderung nicht berücksichtigen. Den Speicherbedarf ihrer Implementierung können Sie mithilfe des Programms `RankMem.java` aus dem Archiv `7.1b.zip` verifizieren. Diesem übergeben Sie die Länge des Bitvektors als einziges Argument. Aufgrund von Messungenauigkeiten sollten Sie relativ große Längen ausprobieren, z.B. $n = 100000000$. Die Korrektheit können Sie mit dem Programm `RankCorrect.java` testen.

(c) `BitVectorSelect` Klasse **(1.5 Punkte)**

Implementieren Sie eine Klasse `BitVectorSelect`, welche folgende Anforderungen erfüllt.

- Der Konstruktor erwartet ein Objekt der Klasse `BitVectorRank` aus Aufgabe (b).

  Hinweis: Wenn Sie Aufgabe (b) nicht bearbeitet haben, verwenden Sie einfach die naive Implementierung von `BitVector` und `BitVectorRank` aus dem Archiv `7.1c.zip`.

- Die Methode `int size()` gibt die Länge n des Bitvektors zurück.

- Die Methode `int select(int k)` gibt die Position des `k`-ten `1`-Bits zurück. Im Beispiel aus Teilaufgabe (b) gilt `select(2) = 4` und `select(8) = 14`. Wenn `k` < 1, oder wenn es weniger als `k` `1`-Bits gibt, sollte die Methode `-1` zurückgeben. Die Beantwortung der Anfrage darf höchstens $\mathcal{O}(\log n)$ viele Rechenschritte benötigen. Das gelingt Ihnen vermutlich nur, wenn Sie die Methode `rank` des `BitVectorRank`-Objektes verwenden.

- Die Methode `int select(int k, int start, int end)` gibt die Position des `k`-ten `1`-Bits im Intervall $\{$`start`$, \ldots,$ `end - 1`$\}$ zurück (oder `-1`, falls das Intervall weniger als `k` `1`-Bits enthält). Im Beispiel aus Teilaufgabe (b) gilt `select(3, 5, 12) = 8`. Die Beantwortung der Anfrage darf höchstens $\mathcal{O}(c \log n)$ viele Rechenschritte benötigen (wobei c der Trade-Off-Parameter der Rank-Datenstruktur ist).

- **Tipp:** Verwenden Sie eine binäre Suche!

- **Das Objekt der Klasse `BitVectorSelect` sollte nur konstant wenig Speicher benötigen!** Es sollten keine weiteren Arrays oder Ähnliches angelegt werden.

Die Korrektheit können Sie mit dem Programm `SelectCorrect.java` aus dem Archiv `7.1c.zip` überprüfen.

Hinweise und Tipps

Platzeffiziente Speicherung von Bitvektoren

Ein Array `boolean [`n`]` wird von Java üblicherweise in n Bytes (also $8n$ Bits) gespeichert. Damit werden also acht mal mehr Bits verwendet, als eigentlich nötig wären. Eine kompaktere Darstellung verwendet stattdessen ein Array `int [k]` mit `k` $= \left\lceil \frac{n}{32} \right\rceil$ Einträgen.

Ein einzelner Integer wird von Java in 32 Bits gespeichert. Damit können in einem Integer 32 Stellen des Bitvektors gespeichert werden! Die j-te Position des Bitvektors entspricht also dem (j mod 32)-ten Bit im ($j/32$)-ten Integer. Um nur das i-te Bit eines Integers **auszulesen** (mit $i \in \{0, \ldots, 31\}$), können Sie Bit-Operatoren verwenden. Hier ein Beispiel mit $i = 7$:

```
int word = ...;        // Binärdarstellung: 01101010111101100011100 1 0011110
shifted = word >> 7;       // Rechts-Shift: 0000000 01101010111101100011100 1
int mask = 1;          // Rechtestes 1-Bit: 0000000 00000000000000000000000 1
boolean result = (shifted & mask) > 0; // true wenn grün=1; false wenn grün=0.
```

Wenn Sie das i-te Bit **verändern** wollen, hängt die Strategie vom neuen Wert ab:

```
// Fall 1: Das Bit soll auf 1 gesetzt werden.
int word = ...;        // Binärdarstellung: 01101010111101100011100 0 0011110
int mask = 1 << 7;    // Maske mit 7tem Bit: 00000000000000000000000 1 0000000
word = word | mask;         // OR-Operator: 01101010111101100011100 1 0011110
```

```
// Fall 2: Das Bit soll auf 0 gesetzt werden.
int word = ...;        // Binärdarstellung: 01101010111101100011100 1 0011110
int mask = 1 << 7;    // Maske mit 7tem Bit: 00000000000000000000000 1 0000000
mask = ~mask;                  // Bit-Flip: 11111111111111111111111 0 1111111
word = word & mask;        // AND-Operator: 01101010111101100011100 0 0011110
```

Weitere Informationen zu Bit-Operationen finden Sie zum Beispiel hier:

`https://www.geeksforgeeks.org/bitwise-operators-in-java/`

Platzeffiziente `rank`-Datenstrukturen

Eine simple Implementierung einer `rank`-Datenstruktur sieht wie folgt aus: Sie berechnen während des Konstruktoraufrufs einfach die Ergebnisse aller Anfragen, und speichern diese in einem `int`-Array mit n Einträgen. Das geht in $\mathcal{O}(n)$ Zeit mithilfe folgender Formel:

$$\texttt{rank}(0) = 0 \qquad \forall i \in \{1, \ldots, n-1\} : \texttt{rank(i)} = \begin{cases} \texttt{rank}(i-1) + 1 & \text{, if } B[i] = \texttt{true} \\ \texttt{rank}(i-1) & \text{, if } B[i] = \texttt{false} \end{cases}$$

Beim Aufruf der Methode `rank(i)` können Sie das Ergebnis dann einfach in konstanter Zeit aus dem i-ten Arrayeintrag auslesen. Sie benötigen dann natürlich $32n$ Bits zusätzlichen Speicherplatz für das Array!

Sie können den benötigten Speicher um einen beliebigen Faktor c reduzieren, indem Sie nicht das Ergebnis von `rank(i)` für jedes $i \in \{0, \ldots, n-1\}$ vorberechnen, sondern nur für jede c-te mögliche Anfrage (oder formaler für alle $i = c \cdot j$ mit $j \in \{0, \ldots, \left\lfloor \frac{n}{c} \right\rfloor\}$). Die Ergebnisse können Sie in einem `int`-Array abspeichern, welches $32 \cdot \left\lfloor 1 + \frac{n}{c} \right\rfloor$ Bits benötigt.

Das Ergebnis der Anfrage $\texttt{rank}(i)$ lässt sich nun leicht berechnen. Sei $i' = c \cdot \left\lfloor \frac{i}{c} \right\rfloor$, dann ist das Ergebnis von $\texttt{rank}(i)$ die Summe aus $\texttt{rank}(i')$ und der Anzahl der 1-Bits im Bitvektor-Intervall $[i', i-1]$. Da i' ein vielfaches von c ist, haben Sie das Ergebnis von $\texttt{rank}(i')$ bereits vorberechnet, und können es einfach in konstanter Zeit nachschlagen. Die Anzahl der 1-Bits im Bitvektor-Intervall $[i', i-1]$ können Sie einfach ganz naiv in $\mathcal{O}(c)$ Zeit zählen, da das Interval maximal Länge $c-1$ hat.

Beispiel-Programm

```
import java.util.*;

class Example {

  private static BitVector BVfromString(String s) {
    BitVector bv = new BitVector(s.length());
    for (int i = 0; i < s.length(); i++)
      bv.set(i, (s.charAt(i) != '0'));
    return bv;
  }

  private static String BVtoString(BitVector bv) {
    StringBuilder result = new StringBuilder();
    for (int i = 0; i < bv.size(); i++)
      result.append(bv.get(i) ? '1' : '0');
    return result.toString();
  }

  public static void main ( String[] args ) {
    // Aufgabe (a)
    String input = "0100110111100010101011";
    BitVector bvDs = BVfromString(input);
    System.out.println("Bitvektor: " + BVtoString(bvDs) + "\n");

    // Aufgabe (b)
    BitVectorRank rankDs = new BitVectorRank(bvDs, 3);
    for (int i = 0; i < rankDs.size() + 1; i++)
      System.out.println("rank(" + i + ") = " + rankDs.rank(i));

    System.out.println("count(2, 9) = " + rankDs.count(2, 9) + "\n");

    // Aufgabe (c)
    BitVectorSelect selectDs = new BitVectorSelect(rankDs);
    for (int k = 0; k <= 12; k++)
      System.out.println("select(" + k + ") = " + selectDs.select(k));

    System.out.println("select(" + 13 + ") = " + selectDs.select(13) +
                       " (BV enthaelt nur 12 1-Bits)" + "\n");

    for (int k = 0; k <= 5; k++)
      System.out.println("select(" + k + ", 9, 20) = " +
                         selectDs.select(k, 9, 20));

    System.out.println("select(" + 6 + ", 9, 20) = " +
                       selectDs.select(6, 9, 20) +
                       " (BV-Intervall enthaelt nur 5 1-Bits)");

  }
}
```

```
/* Erzeugte Ausgabe:
Bitvektor: 0100110111100010101011

rank(0) = 0
rank(1) = 0
rank(2) = 1
rank(3) = 1
rank(4) = 1
rank(5) = 2
rank(6) = 3
rank(7) = 3
rank(8) = 4
rank(9) = 5
rank(10) = 6
rank(11) = 7
rank(12) = 7
rank(13) = 7
rank(14) = 7
rank(15) = 8
rank(16) = 8
rank(17) = 9
rank(18) = 9
rank(19) = 10
rank(20) = 10
rank(21) = 11
rank(22) = 12
count(2, 9) = 4

select(0) = -1
select(1) = 1
select(2) = 4
select(3) = 5
select(4) = 7
select(5) = 8
select(6) = 9
select(7) = 10
select(8) = 14
select(9) = 16
select(10) = 18
select(11) = 20
select(12) = 21
select(13) = -1 (BV enthaelt nur 12 1-Bits)

select(0, 9, 20) = -1
select(1, 9, 20) = 9
select(2, 9, 20) = 10
select(3, 9, 20) = 14
select(4, 9, 20) = 16
select(5, 9, 20) = 18
select(6, 9, 20) = -1 (BV-Intervall enthaelt nur 5 1-Bits)
/*
```