Übungen zur Vorlesung

Algorithmen und Datenstrukturen

Übungsblatt 07

Cryptoplexity
Cryptography & Complexity Theory
Technische Universität Darmstadt
www.cryptoplexity.de

Arbeitsgruppe Kryptographie und Komplexitätstheorie
Prof. Dr. Marc Fischlin
Moritz Huppert
Tobias Schmalz

Sommersemester 2024
Veröffentlicht: 31.05.2024, 14:00 Uhr MESZ
Abgabe: 14.06.2024, 23:59 Uhr MESZ

String-Matching

Auf diesem Übungsblatt werden wir uns mit dem *String-Matching*-Problem auseinandersetzen: Es sollen Algorithmen zum Finden von Textsegmenten in einer Zeichenkette anhand eines vorgegebenen Suchmusters erarbeitet werden.

Wir formulieren das String-Matching-Problem wie folgt. Der zu durchsuchende Text entspricht einem Array T der Länge n, und das Textmuster einem Array P der Länge $m \leq n$. Wir nehmen an, dass die Elemente von P und T Zeichen aus einem endlichen Alphabet Σ sind. Gesucht sind nun alle gültigen „Verschiebungen“, mit denen P in T auftaucht. Mit anderen Worten: Es müssen alle $sft \in \mathbb{N}$ bestimmt werden, sodass $T[sft, \ldots, sft + m - 1] = P$ gilt, also $T[sft + j] = P[j]$ für alle $0 \leq j \leq m - 1$.

Wir beginnen in der nächsten Aufgabe mit dem einfachsten und intuitivsten Ansatz, um dieses Problem zu lösen: Dem „naiven“ String-Matching-Algorithmus.

G1 Gruppendiskussion

Nehmen Sie sich etwas Zeit, um die folgenden Fachbegriffe in einer Kleingruppe zu besprechen, sodass Sie anschließend in der Lage sind, die Begriffe dem Rest der Übungsgruppe zu erklären:

(a) String Matching;
(b) Heaps;
(c) Splay-Bäume.

G2 String-Matching I (Naiv)

Ziel dieser Aufgabe ist es, den naiven String-Matching-Algorithmus zu erarbeiten. Dieser Algorithmus bestimmt mithilfe einer Schleife alle gültigen Verschiebungen, indem für alle möglichen Werte $0 \leq sft \leq n - m$ überprüft wird, ob $T[sft, \ldots, sft + m - 1]$ und P übereinstimmen.

(a) Geben Sie den Pseudocode des naiven String-Matching-Algorithmus NaiveStringMatching an. Der Algorithmus soll den zu durchsuchenden Text T und das Textmusterarray P als Eingabe bekommen, und die Liste aller korrekten Verschiebungen ausgeben.
(b) Beweisen Sie die Korrektheit und analysieren Sie die Laufzeit von NaiveStringMatching.
(c) Betrachten Sie den Text $T = [h, e, h, e, h, h, h, e, y, h]$ und das Muster $P = [h, e, h]$. Benutzen Sie den naiven String-Matching-Algorithmus, um alle Vorkommen (in Form von Verschiebungen) von P in T zu bestimmen.

G3 String-Matching II (Rabin–Karp Algorithmus)

In der vorherigen Aufgabe haben den Algorithmus NaiveStringMatching das Problem des String-Matchings kennengelernt, der jedoch relativ ineffizient ist. Grund dafür ist, dass die Informationen über den Text T, die bei der Behandlung eines bestimmten Wertes von *sft* gewonnen werden, bei der Bearbeitung anderer Werte nicht miteinfließen. Solche Informationen können aber sehr wertvoll sein, und die Laufzeit des Algorithmus deutlich verkürzen. In dieser Übung werden Sie zwei alternative Ansätze untersuchen, mit denen versucht wird, diese Informationen einzusetzen. Beide Ansätze benötigen eine Vorverarbeitungsphase, um das Suchen des Textmusters danach zu beschleunigen. In dieser Aufgabe lernen Sie den *Rabin-Karp-Algorithmus* kennen.

Dieser Algorithmus basiert auf elementaren zahlentheoretischen Begriffen, wie z.B. der Gleichheit zweier Zahlen modulo einer dritten Zahl. Deshalb ist es notwendig, das Alphabet Σ (mit $|\Sigma| = d$) zunächst mit den Zahlen $\{0, \ldots, d-1\}$ zu identifizieren. Wir nehmen hier einfachheitshalber an, dass $\Sigma = \{0, \ldots, 9\}$, sodass wir wie gewohnt im Dezimalsystem ($d = 10$) arbeiten können. Wenn z.B. Σ das deutsche Alphabet wäre, dann müsste man in Basis $d = 26$ arbeiten.

Der Rabin-Karp-Algorithmus basiert auf folgender Überlegung: Für jedes $0 \leq sft \leq n - m$, sei t_{sft} der Dezimalwert des Teilstrings $T[sft, \ldots, sft + m - 1]$, und sei p der Dezimalwert von P. Offenbar ist *sft* eine gültige Verschiebung, also $T[sft, \ldots, sft + m - 1] = P$, genau dann wenn $t_{sft} = p$.

(a) Zeigen Sie, wie man p und t_0 in Zeit $\Theta(m)$ berechnen kann.
(b) Überlegen Sie sich, wie man t_{sft+1} aus t_{sft} in konstanter Zeit berechnen kann.
(c) Geben Sie einen Algorithmus RabinKarpMatchBasic an, der die oben erläuterte Vorgehensweise benutzt, um das String-Matching-Problem zu lösen. Der Algorithmus soll das zu durchsuchende Array T und das Musterarray P als Eingabe bekommen, und die Liste aller korrekten Verschiebungen ausgeben.

Wir haben bisher in unserer Vorgehensweise ein kleines Problem ignoriert: Mit zunehmender Länge des Suchmusters können die Zahlen p und t_{sft} sehr groß werden. Es könnte unter Umständen nicht angemessen sein, wenn wir voraussetzen, dass jede arithmetische Operation auf dem Wert p nur „konstante Zeit“ benötigt.

Dieses Problem kann wie folgt gelöst werden: Sei q eine Primzahl, sodass $10q$ in ein Computerwort passt. Der Trick ist nun, die Werte p und t_{sft} einfach modulo q zu berechnen und zu vergleichen; damit werden die betrachteten Zahlen um einiges kleiner. Man beachte allerdings, dass diese Lösung nicht perfekt ist: Aus $t_{sft} \equiv p \pmod{q}$ folgt nicht $t_{sft} = p$. Auf der anderen Seite gilt mit Sicherheit $t_{sft} \neq p$ wenn $t_{sft} \not\equiv p \pmod{q}$.

Man kann die Kongruenz $t_{sft} \equiv p \pmod{q}$ also als einen schnellen Test verwenden, um ungültige Verschiebungen auszuschließen. Wenn allerdings $t_{sft} \equiv p \pmod{q}$ für eine Verschiebung *sft* gilt, muss nochmal explizit nachgeprüft werden, ob $T[sft, \ldots, sft + m - 1] = P[0, \ldots, m - 1]$, also ob *sft* auch eine gültige Verschiebung ist, oder ob ein *unechter* Treffer vorliegt.

(d) Passen Sie Ihren Algorithmus RabinKarpMatchBasic an, sodass der eben beschriebene Test verwendet wird. Der resultierende Algorithmus RabinKarpMatch soll zusätzlich zu den vorherigen Parametern auch die Primzahl q als Eingabe bekommen.

(e) Finden Sie mithilfe des Rabin-Karp Algorithmus alle Vorkommen des Integer-Arrays $P = [7, 3, 4]$ in $T = [6, 9, 1, 7, 3, 4, 5, 0, 9, 4, 6, 2, 4, 8, 7, 3, 4]$. Verwenden Sie dabei $q = 13$. Sie können folgende Tabelle als Hilfsmittel benutzen:

sft	t_{sft}	$t_{sft} \pmod q$	$t_{sft} == p \pmod q$	$T[sft, \ldots, sft + m - 1] == P$	Treffer?
0					
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					

G4 Heaps

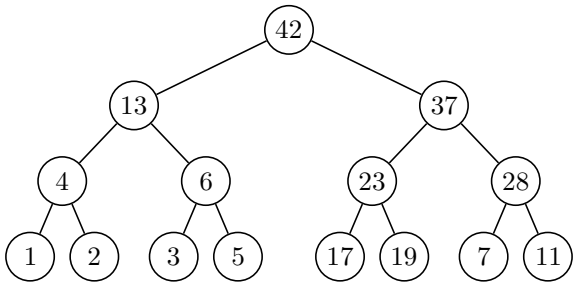
(a) Geben Sie an, welche der folgenden Bäume und Arrays die Heap-Eigenschaft erfüllen. Begründen Sie bei den Übrigen, was der Heap-Eigenschaft widerspricht.

11	4	23	2	6	17	37	1	3	5	7	13	19	28	42

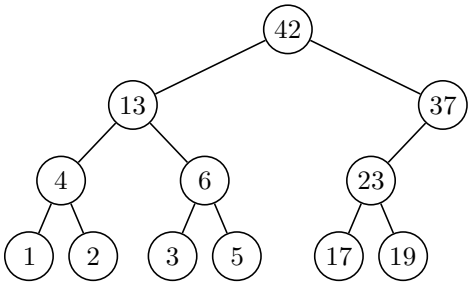
(i)

42	37	28	13	19	23	17	11	1	2	3	4	5	6	7

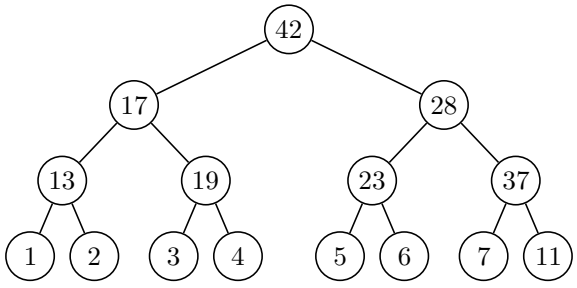
(ii)



(iii)



(iv)



(v)

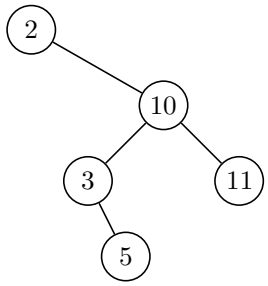
(b) Um einen Schlüssel in ein Heap einzufügen könnte man, anstelle von Insert(H, k), auch zuerst append(H, k) und danach BuildHeap$(H.A)$ aufrufen, wobei append(H, k) die Größe des Arrays inkrementiert und den neuen Eintrag an die letzte Position des Arrays einfügt. Welchen Nachteil hätte diese Implementierung?

(c) Erklären Sie, weshalb BuildHeap$(H.A)$ die Methode Heapify$(H.A, i)$ nur für die erste Hälfte (abgerundet) des Arrays durchführt, und weshalb dies nicht in aufsteigender Reihenfolge geschehen darf.

(d) Führen Sie HeapSort auf das Array $A = [6, 4, 1, 8, 3, 7]$ aus. Verwenden Sie dabei den Algorithmus aus der Vorlesung. Stellen Sie dazu das Heap *jeweils* in seiner anfänglichen Konfiguration, sowie nach jedem Aufruf der Methode Heapify und nach jeder Ausgabe des Maximums dar.

G5 Splay-Bäume

(a) Fügen Sie der Reihe nach die Knoten $4, 8, 7, 1, 6, 9$ in den abgebildeten Splay-Baum ein. Verwenden Sie dabei den Algorithmus aus der Vorlesung zum Einfügen von Knoten in einen Splay-Baum. Zeichnen Sie Ihr Zwischenergebnis *jeweils* nach dem Einfügen des Knotens, sowie nach jeder erfolgten Zig-Zig-, Zig-Zag-, und Zig-Operation.



(b) Betrachten Sie den resultierenden Splay-Baum aus G5(a), und suchen Sie den Knoten 5. Verwenden Sie dabei den Algorithmus aus der Vorlesung zum Suchen von Knoten in einem Splay-Baum. Zeichnen Sie Ihr Zwischenergebnis *jeweils* nach jeder erfolgten Zig-Zig-, Zig-Zag-, und Zig-Operation.

(c) Betrachten Sie den resultierenden Splay-Baum aus G5(b), und löschen Sie den Knoten 8. Verwenden Sie dabei den Algorithmus aus der Vorlesung zum Löschen von Knoten aus einem Splay-Baum. Zeichnen Sie Ihr Zwischenergebnis *jeweils* nach jeder erfolgten Zig-Zig-, Zig-Zag-, und Zig-Operation, sowie nach dem Löschen des Knotens und Aufteilen des Baumes, und nach dem Zusammenführen der zwei Teilbäume.

G6* String-Matching III (Endliche Automaten)

In den Aufgaben G2 und G3 haben Sie bislang zwei Algorithmen für das String-Matching Problem kennengelernt: Der naive Algorithms NaiveStringMatching löst das Problem zwar intuitiv, ist jedoch ineffizient. Der *Rabin-Karp-Algorithmus* benötigt eine Vorverarbeitungsphase, um das Suchen des Textmusters danach zu beschleunigen. Wir erarbeiten in dieser Aufgabe einen auf endlichen Automaten beruhenden Ansatz, bei dem es nach einem cleveren Preprocessing des Musters P ausreicht, das Textarray T genau einmal von links nach rechts durchzugehen, um das String-Matching-Problem zu lösen.

Dieser Ansatz basiert auf folgender Überlegung: Das String-Matching-Problem wäre einfach lösbar, wenn wir beim Durchgehen von T bei jedem Zeichen genau wüssten, wie groß die Überlappung zwischen den zuletzt gelesenen Zeichen von T und einem Anfangssegment von P ist (ob sich also ein 'Match' anbahnt oder nicht). Man hätte dann eine korrekte Verschiebung, sobald eine lückenlose Übereinstimmung zwischen P und einem gewissen Teilstring von T auftritt.

In der Vorverarbeitungsphase werden genau diese Informationen gesammelt. Angenommen, P ist ein Textmuster der Länge m, und beim Durchgehen von T sind wir gerade auf eine Übereinstimmung der letzten j gelesenen Symbole von T mit $P[0, \ldots, j-1]$ $(0 \leq j \leq m)$ gestoßen (die Variable j wird weiter unten in *st* umbenannt; der Grund wird gleich erklärt). Wie verändert sich die Länge dieser Überlappung, wenn wir zum nächsten Symbol weitergehen? Diese Informationen können effektiv durch einen endlichen Automaten dargestellt werden.

Informell ist ein endlicher Automat eine Maschine, die den gegebenen String T einmal von links nach rechts durchliest. Dabei befindet sich der Automat immer in einem von $m+1$ vielen internen Zuständen (einer pro Größe der Übereinstimmung zwischen den zuletzt gelesenen Zeichen von T und dem Anfang von P). Die Maschine geht nach dem Einlesen von einem neuen Symbol von T in einen neuen Zustand über, in Abhängigkeit *nur* vom aktuellen Zustand und dem gerade gelesenen Zeichen.

Allgemein gilt, dass sich der Automat im Zustand $0 \leq st \leq m$ befindet genau dann, wenn die letzten st gelesenen Zeichen von T mit den ersten st Symbolen von P übereinstimmen, und es kein $i > st$ mit derselben Eigenschaft gibt. Die Zustandsübergänge werden so gewählt, dass diese Eigenschaft beim Lesen jedes neuen Symbols immer erhalten bleibt: Für jeden Zustand $0 \leq st \leq m$ und jedes Zeichen $w \in \Sigma$ muss die Länge der größten Überlappung zwischen den letzten Symbolen von $[P[0], \ldots, P[st-1], w]$ und dem Anfang von P bestimmt werden – Dies ist dann der Zielzustand $\delta(st, w)$, ausgehend von Zustand st mit Eingabe w.

(a) Angenommen, für ein Textmuster P ist der oben beschriebene endliche Automat mit Übergangsfunktion δ gegeben. Entwerfen Sie einen Algorithmus FSMMatching, der den Automaten benutzt, um das String-Matching-Problem für einen Eingabetext T zu lösen. Beschreiben Sie Ihren Algorithmus und stellen Sie ihn in Pseudocode dar. Der Algorithmus bekommt als Eingabe das zu durchsuchende Array T, den endlichen Automaten für das Textmuster P (in Form der Übergangsfunktion δ), und die Länge m des Textmusters, und er soll die Liste aller korrekten Verschiebungen ausgeben.

(b) Betrachten Sie nun das Alphabet $\Sigma = \{a, g, n, o\}$ und das Muster $P = [n, a, n, o]$. Zeichnen Sie nach den oben erklärten Regeln den entsprechenden endlichen Automaten und benutzen Sie ihn, um den Text $T = [g, a, n, a, n, n, a, n, o, n, a, n, a, n, o, g]$ nach dem Muster P zu durchsuchen.

Hausübungen

In diesem Bereich finden Sie die praktische Hausübung von Blatt 07. Bitte beachten Sie die allgemeinen Hinweise zu den Hausübungen und deren Abgabe im Moodle-Kurs.

Bitte reichen Sie Ihre Abgabe bis spätestens *Freitag, 14.06.2024, 23:59 Uhr MESZ* ein. Verspätete Abgaben können *nicht* berücksichtigt werden.

Sie können davon ausgehen, dass immer nur für eine Aufgabe sinnvolle Werte an die Methoden übergeben werden, also z.B. dass ein Baum nicht `null`. Alle zu bearbeitenden Klassen befinden sich im Paket `p2.binarytree`.

Hinweis zum Testen

In den öffentlichen Tests werden die verwendeten Bäume in den Fehlermeldungen in einem kryptischen Format ausgegeben. Sie können mittels der `main`-Methode eine graphische Oberfläche starten, mit welcher Sie sich die Bäume visualisieren lassen können. Eine kurze Erklärung wie diese zu verwenden ist finden Sie im Kommentar über der `main`-Methode in der Klasse `Main`.

H1 Checker für RB Trees — 5 Punkte (1 + 1 + 1 + 2)

In dieser Hausübung werden Sie sich mit Rot-Schwarz-Bäumen beschäftigen, welche Sie bereits aus der Vorlesung kennen.

Um die Korrektheit eines Baumes zu verifizieren, soll zunächst ein Checker implementiert werden, welcher den übergebenen Rot-Schwarz-Baum auf die folgenden Eigenschaften überprüft:

(a) Regel 1: Jeder Knoten hat eine Farbe und das Attribute ist entweder rot oder schwarz.

(b) Regel 2: Die Wurzel ist schwarz.

(c) Regel 3: Rote Knoten haben keine roten Kinder (Es dürfen keine zwei roten Knoten aufeinander folgen).

(d) Regel 4: Jeder Pfad von einem Knoten zu seinen Blättern enthält die gleiche Anzahl schwarzer Knoten.

Vervollständigen Sie dafür die Klasse `RBTreeChecker`. Werfen Sie eine `RBTreeException`, sobald Sie einen Verstoß gegen eine Regel gefunden haben. Welche Nachricht Sie in der Exception zurückgeben, ist Ihnen überlassen.

Sie dürfen zum Überprüfen einer Regel nur einmal über den Baum iterieren: Mehrfaches Iterieren über den Baum, um z.B. die Schwarzhöhe für jeden Knoten neu zu berechnen, ist nicht erlaubt. Konkret bedeutet dies, dass im Durchlauf einer Methode auf jedem Knoten nur einmal die Methode `getLeft` bzw. `getRight` aufgerufen werden darf[1]. Beachten Sie bitte, dass jede Regel einzeln betrachtet werden soll: Wird eine andere Regel verletzt, aber die aktuell geprüfte Regel ist korrekt, dann soll keine Exception geworfen werden.

Hinweis: Wie Sie die Aufgabe lösen, ist Ihnen überlassen. Das Verändern der Methodensignaturen ist allerdings nicht erlaubt.

H2 RB-Tree Insert — 8 Punkte (2 + 5 + 1)

In dieser Aufgabe beschäftigen Sie sich mit dem Einfügen von neuen Knoten in Rot-Schwarz-Bäume.

(a) Beginnen Sie zunächst damit, die Methode `insert` in der Klasse `AbstractBinarySearchTree` zu implementieren. Diese Methode bekommt einen neuen Knoten und einen initialen Wert für den Verweis zu dem Elternknoten `px` übergeben. Anschließend fügt die Methode den Knoten an der richtigen Stelle ein.

Implementieren Sie dazu den Pseudocode zum Einfügen aus der Vorlesung (Vorlesung 03, Folie 71). Beachten Sie dabei, dass wir nicht von einer konkreten Implementierung ausgehen und daher statt des Sentinelknotens den Parameter `initialPX` verwenden. Für normale binäre Suchbäume hat dieser den Wert `null`, und für Rot-Schwarz-Bäume entspricht er dem Sentinelknoten.

[1] Dies wird auch von den öffentlichen Tests abgefragt.

(b) Darüber hinaus implementieren Sie die Methoden `rotateLeft`, `rotateRight` und `fixColorsAfterInsertion` in der Klasse `RBTree`. Diese Methoden bekommen ebenfalls einen Startknoten übergeben und rotieren oder modifizieren die entsprechenden Knoten so, dass die Bedingungen für einen Rot-Schwarz-Baum wieder erfüllt sind. Sie können sich für die Implementation an den Pseudocode aus der Vorlesung orientieren[2].

(c) Abschließend implementieren Sie die Methode `insert` in der Klasse `RBTree`. Diese Methode bekommt einen Wert übergeben und fügt mithilfe der vorherigen Methoden den Knoten in den Baum ein. Es wird zunächst die `insert`-Methode aufgerufen und danach die `fixColorsAfterInsertion`-Methoden, um die Bedingungen des Rot-Schwarz-Baums wieder zu erfüllen. Benutzen Sie zum Erstellen eines Knotens die Methode `createNode`.

H3 Autovervollständigung — 7 Punkte (2 + 3 + 2)

Aufgrund der Gruppierungsstruktur der RB-Bäume können diese auch dazu verwendet werden, alle Strings zurückzugeben, die ein bestimmtes Präfix haben. Wir nutzen dies in dieser Aufgabe, um eine einfache Autovervollständigung zu implementieren.

(a) Um dies umzusetzen, müssen wir uns mit verschiedenen Arten der Traversierung über einen binären Baum beschäftigen. Implementieren Sie dafür zunächst die Methode `inOrder` in der Klasse `AbstractBinarySearchTree`. Diese Methode führt eine In-Order-Traversierung über den Baum, angefangen von dem übergebenen Knoten, durch und speichert die Werte der besuchten Knoten in der übergebenen Ergebnisliste. Darüber hinaus gibt es zwei weitere Anforderungen an die Methode:

- Die Methode soll höchstens `max` viele Werte in die Liste hinzufügen. Falls eigentlich mehr Werte in der Traversierung enthalten sind, sollen diese nicht weiter betrachtet werden.
- Die Traversierung soll abbrechen, sobald das übergebene `Predicate` `false` für einen betrachteten Wert zurückgibt. Somit soll der erste Wert, für den `false` zurückgegeben wurde, und die folgenden Werte nicht mehr Teil der Ergebnisliste sein. Sie können davon ausgehen, dass das `Predicate` auch für alle nachfolgenden Knoten `false` zurückgibt.

(b) Implementieren Sie zusätzlich die Methode `findNext` in derselben Klasse. Die Parameter der Methode haben dieselbe Bedeutung wie zuvor. Der Unterschied zu `inOrder` ist, dass die Methode die nächsten Werte im zugehörigen Baum in aufsteigender Reihenfolge zurückgibt, beginnend mit dem Wert des übergebenen Knotens. Beachten Sie, dass Sie dafür u.U. auch den Elternknoten betrachten müssen. Es ist dabei nicht erlaubt, die Methode mittels einer In-Order-Traversierung über den Wurzelknoten umzusetzen. Es soll direkt bei dem übergebenen Knoten gestartet werden.

(c) Implementieren Sie zum Schluss die Klasse `AutoComplete`, welche die eigentliche Funktionalität umsetzt. Die Methode `autoComplete` ist bereits für Sie implementiert. Sie müssen nun noch die Methode `prefixSearch` implementieren. Diese bekommt einen String übergeben und gibt anschließend den kleinsten Knoten zurück, für den gilt, dass der übergebene String ein Präfix seines Schlüssels ist[3]. Falls kein solcher Knoten vorhanden ist, gibt die Methode `null` zurück. Der verwendete binäre Suchbaum ist in dem Feld `searchTree` gespeichert.

(d) (*Unbewertet*) In der Klasse `Main` finden Sie ein Beispiel, wie die Autovervollständigung verwendet wird um 10 Ergebnisse für den Präfix `z` zu finden. Dabei wird einmal ein normaler, binärer Suchbaum verwendet und einmal ein Rot-Schwarz-Baum. Für beide wird jeweils die Zeit ausgegeben, die benötigt wurde um den Baum zu initialisieren (die Worte einzufügen) und die Anfrage zu beantworten. Führen Sie die Methode aus und vergleichen Sie die benötigten Zeiten. Können Sie sich die Ergebnisse erklären? Schauen Sie sich dafür auch eventuell den Aufbau der Eingabedaten in `main/resources/p2` an.

[2] `rotateRight` funktioniert symmetrisch zu `rotateLeft`, d.h. Sie müssen überall links und rechts tauschen.
[3] Um zu prüfen, ob der übergebene String ein Präfix des Schlüssels ist, können Sie die Methode `String#startsWith` verwenden.

H4 Joinen von RB-Trees — 10 Punkte (1 + 3 + 6)

Die Aufgabe besteht darin, zwei Rot-Schwarz-Bäume, $T1$ und $T2$, zu einem einzigen Rot-Schwarz-Baum zusammenzuführen. Dabei können Sie davon ausgehen, dass alle Werte im rechten Baum ($T2$) größer sind als alle Werte im linken Baum ($T1$). Zudem wird beim Zusammenführen ein neuer, mittlerer Wert k eingefügt, sodass die Bedingung $T1.\max < k < T2.\min$ erfüllt ist.

Um die beiden Rot-Schwarz-Bäume T1 und T2 zusammenzuführen, gehen Sie wie folgt vor:

(a) Implementierung Sie Methode `blackHeight` in der Klasse `RBTree`, welche die Schwarzhöhe des Baumes zurückgibt.

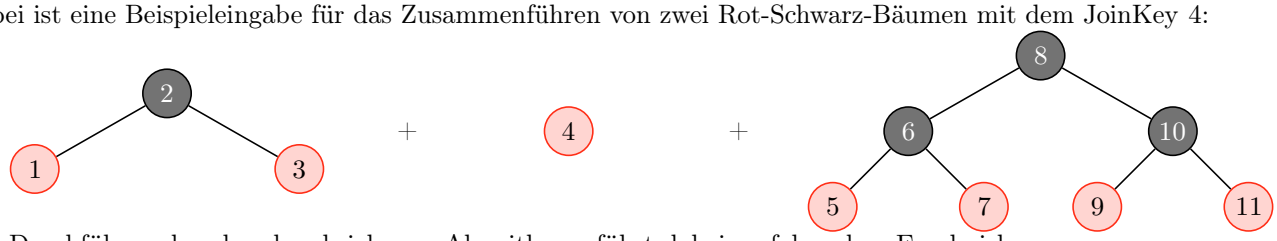
(b) Implementieren Sie anschließend die Methode `findBlackNodeWithBlackHeight` in der Klasse `RBTree`. Diese Methode erhält die Zielhöhe des zu findenden Knotens und gibt den kleinsten oder größten schwarzen Knoten mit der entsprechenden Schwarzhöhe zurück. Zudem werden die Gesamthöhe des Baumes und eine Flag übergeben, die angibt, ob der größte oder kleinste Knoten gefunden werden soll. Sie können davon ausgehen, dass der Baum nicht leer ist.

(c) Implementierung Sie final die Methode `join` in der Klasse `RBTree`. Diese Methode erhält einen RB-Tree und einen JoinKey k. Ziel ist es, beide Bäume unter Zuhilfenahme des JoinKeys zu einem einzigen Baum zusammenzuführen. Der übergebene Baum wird zu dem aktuellen Baum hinzugefügt. Die Methode soll dabei nach folgenden Schritten vorgehen:

- Zunächst wird die Schwarzhöhe der beiden Bäume verglichen.
- Dann wird nach zwei Knoten in beiden Bäumen gesucht, die dieselbe Schwarzhöhe besitzen: Vom Baum mit der kleineren Schwarzhöhe wird die Wurzel verwendet und vom Baum mit der größeren Schwarzhöhe der kleinste bzw. größte Knoten mit derselben Schwarzhöhe wie die Wurzel im kleineren Baum.
- Anschließend wird mit der Methode `createNode` ein neuer, roter Knoten mit dem JoinKey als Schlüssel erstellt, welcher als Parent der beiden zuvor gefundenen Knoten eingefügt wird, sodass ein zusammengeführter Baum entsteht. Denken Sie daran, alle zugehörigen Kinder- und Elternverweise zu aktualisieren und, wenn notwendig, den Wurzelknoten des momentanen Baumes zu aktualisieren. Beachten Sie auch den Fall, dass beide Bäume dieselbe Schwarzhöhe haben.
- Zum Abschluss müssen die Eigenschaften des Rot-Schwarz-Baumes wiederhergestellt werden, da nun eventuell zwei rote Knoten übereinander stehen. Rufen Sie dafür die Methode `fixColorsAfterInsertion` mit dem neu erstellten Knoten auf.

Sie können davon ausgehen, dass beide Bäume nicht leer sind.

Anbei ist eine Beispieleingabe für das Zusammenführen von zwei Rot-Schwarz-Bäumen mit dem JoinKey 4:



Das Durchführen des oben beschriebenen Algorithmus führt dabei zu folgendem Ergebnisbaum: