Technische Universität Wien
Institut für Computergraphik und Algorithmen
Arbeitsbereich für Algorithmen und Datenstrukturen

186.813 Algorithmen und Datenstrukturen 1 VU 6.0

Sommersemester 2014

Programmieraufgabe

abzugeben bis: Montag, 12. Mai 2014, 15:00

Organisatorisches

Im Rahmen der Lehrveranstaltung **Algorithmen und Datenstrukturen 1** gilt es, eine Programmieraufgabe selbstständig zu lösen, die das Verständnis des im Vorlesungsteil vorgetragenen Stoffes vertiefen soll. Als Programmiersprache wird Java 6 verwendet.

Geben Sie bitte Ihr fertiges, gut getestetes und selbst geschriebenes Programm bis spätestens **Montag, 12. Mai 2014, 15:00** über das Abgabesystem in TUWEL ab. Der von Ihnen abgegebene Code wird vom System automatisch getestet, und Sie erhalten eine entsprechende Rückmeldung im Abgabesystem.

Um eine positive Note zu erhalten, müssen Sie bei der Programmieraufgabe **mindestens einen Punkt erreichen** und Ihren Termin zum Abgabegespräch einhalten.

> Gruppenabgaben bzw. mehrfache Abgaben desselben Programms unter verschiedenen Namen werden nicht akzeptiert. Wenn Sie das abgegebene Programm nicht selbst programmiert haben, erhalten Sie ein negatives Zeugnis. Auch der eigentliche Entwickler kann nur mehr maximal einen Punkt auf dieses Programmierbeispiel erhalten.

Repetitorium zur Programmieraufgabe

Am Montag, 5. Mai 2014, 17:00 – 19:00 findet ein Repetitorium zur Programmieraufgabe im EI 7 statt. Falls Sie allgemeine Fragen zur Aufgabe haben, können Sie diese dort stellen.

Abgabefrist

Sie haben bis Montag, 12. Mai 2014, 15:00 die Möglichkeit ein Programm abzugeben, das alle Testinstanzen korrekt abarbeitet. Danach werden keine weiteren Abgaben akzeptiert, d.h., sollten Sie diese Frist versäumen beziehungsweise Ihr Programm nicht alle verpflichtenden Testinstanzen korrekt abarbeiten, bekommen Sie keine Punkte auf die Programmieraufgabe.

Abgabegespräche

In der Zeit von Dienstag, 13.5., Donnerstag, 15.5., und Freitag, 16.5.2014, finden für alle LVA-Teilnehmer Abgabegespräche statt. Dazu vereinbaren Sie in TUWEL einen individuellen Gesprächstermin, bei dem Sie sich mit einer/m unserer TutorInnen im Informatiklabor treffen und den von Ihnen eingereichten Programmcode erklären können müssen. Sie können sich zu diesem Abgabegespräch von Freitag, 9.5., bis Montag, 12.5.2014, 23:59 in TUWEL bei einer/m TutorIn anmelden.

Falls Sie Systemroutinen in Ihrem Programm verwenden (z.B. das Auffinden eines Minimums), so sollten Sie auch über die Funktionsweise dieser Methoden **genau** Bescheid wissen. Je nach Funktionstüchtigkeit Ihres Programms, Innovation und Effizienz Ihrer Implementierung sowie der Qualität des Abgabegesprächs können Sie bis zu 10 Punkte für diese Programmieraufgabe erhalten. Voraussetzung für eine positive Absolvierung des Abgabegesprächs ist jedenfalls das Verständnis des zugrunde liegenden Stoffes.

Aufgabenstellung

In der Vorlesung haben Sie die Grundlagen für Hashverfahren und mögliche Erweiterungen (z.B. Double Hashing) kennengelernt. Im Zuge dieser Programmieraufgabe sollen Sie eine sogenannte *Distributed Hash Table* (DHT) in Form eines *Content Addressable Networks* (CAN) implementieren. Anwendung finden DHTs unter anderem in Peer-to-Peer (P2P) Overlay Netzwerken. In diesen sind alle Teilnehmer (Clients) üblicherweise gleichberechtigt und verfügen über dieselben Funktionen und Aufgaben. P2P Netzwerke werden oft verwendet, um Daten dezentral und ausfallsicher zu speichern, können jedoch auch andere Aufgaben wie anonymes Routing übernehmen. Durch das dezentrale Speichern der Daten ergibt sich jedoch das Problem diese im Netzwerk wiederzufinden, da jeder Client nur begrenzte Information über die Struktur des Netzwerkes besitzt. Jeder Client kennt nur seine direkten Nachbarn im Overlay Netzwerk, welches die Regeln für die Nachbarschaftsstruktur (d.h. den Graphen, der das Netzwerk repräsentiert) festlegt. CAN ist ein solches Overlay Netzwerk, das auf einer DHT basiert und ein gezieltes Wiederauffinden von Daten ermöglicht.

Funktionsweise

Das Overlay Netzwerk CAN kann im 2-dimensionalen Raum als Rechteck mit den Ausmaßen $([0, X[; [0, Y[) \in \mathbb{N}^2$ dargestellt werden. Jeder mit dem Netzwerk verbundene Client C ist für einen eindeutigen rechteckigen Bereich $C.area = ([x_1, x_2[; [y_1, y_2[) \in \mathbb{N}^2$ im Netz verantwortlich, besitzt eine Position $C.pos = (\frac{x_2-x_1}{2}, \frac{y_2-y_1}{2}) \in \mathbb{R}^{+2}$ und kann M Dokumente speichern. Client C ist mit allen Clients verbunden, die einen direkt angrenzenden Bereich mit C teilen (oberhalb, unterhalb, links oder rechts). Im Folgenden werden diese Clients daher als obere, untere, linke bzw. rechte Nachbarn und die Menge aller Nachbarn von C mit $N(C)$ bezeichnet.

Jedem Dokument D, das im Netzwerk gespeichert werden soll, wird eine eindeutige Position $D.pos = (x, y) \in \mathbb{N}^2$ im Bereich $([0, X[; [0, Y[)$ zugewiesen. *D.pos* wird durch Anwendung

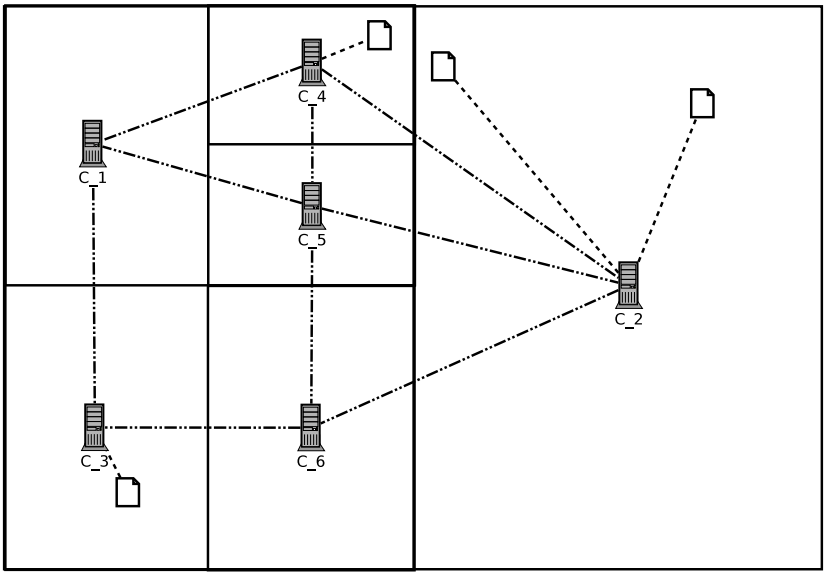

Abbildung 1: Ein beispielhaftes CAN mit mehreren Clients und Dokumenten

zweier unterschiedlicher Hashfunktionen $h_x(\textit{D.name}, i)$ und $h_y(\textit{D.name}, i)$ für offenes Hashing, basierend auf dem Namen des Dokuments und der aktuellen Iteration i für die Sondierung bei Kollisionen, berechnet. Der Client, in dessen Bereich *D.pos* fällt, speichert das Dokument.

Die Programmieraufgabe ist in drei Stufen gegliedert, wobei Sie für jede Stufe unterschiedlich viele Punkte bekommen können. *Beachten Sie, dass Sie Stufe 1 verpflichtend implementieren müssen, um positiv beurteilt werden zu können!*

Stufe 1: Ein Client C sucht nach einem Dokument D (3 Punkte)
C berechnet anhand von $h_x(\textit{D.name}, i)$, $h_y(\textit{D.name}, i)$ und $i = 0$ die vermeintliche Position $p = (x, y)$ des Dokumentes D. Danach sucht C den Client C_v, der für Position p zuständig ist. Ein Client C_v ist für p zuständig, falls für seinen verwalteten Bereich $C_v.\textit{area} = ([x_1, x_2[; [y_1, y_2[)$ gilt:

$$\begin{aligned} C_v.\textit{area}.x_1 &\leq p.x < C_v.\textit{area}.x_2 \quad \text{und} \\ C_v.\textit{area}.y_1 &\leq p.y < C_v.\textit{area}.y_2 \end{aligned}$$

Falls Position p in den Zuständigkeitsbereich von C fällt, ist die Suche beendet, da D direkt im Speicher von C liegt. Ansonsten fragt C alle Nachbarn $C' \in N(C)$, ob einer davon für p zuständig ist. Sollte niemand aus $N(C)$ für p zuständig sein, fragt C den Nachbarn C' – mit der kürzesten euklidischen Distanz $Q(C'.\textit{area}, p)$ zwischen dem verwalteten Bereich des Nachbarn und p – nach dem für die Position p verantwortlichen Client C_v.

$Q(C'.\textit{area}, p)$ wird wie folgt berechnet:

$$\begin{aligned} r.x &= \max(\min(C'.\textit{area}.x_2, p.x), C'.\textit{area}.x_1) \\ r.y &= \max(\min(C'.\textit{area}.y_2, p.y), C'.\textit{area}.y_1) \\ Q(C'.\textit{area}, p) &= \sqrt{(r.x - p.x)^2 + (r.y - p.y)^2} \end{aligned}$$

Sollten mehrere Clients mit der gleichen Distanz zu p existieren, wird unter ihnen jener mit der kleinsten lexikographischen ID gewählt (verwenden Sie `String.compareTo`). C' setzt diesen Prozess rekursiv fort, bis der verantwortliche Client C_v gefunden wurde. Das Ergebnis der Anfrage wird an C zurückgeliefert.

Im nächsten Schritt fragt C bei C_v nach D. Falls C_v das Dokument D gespeichert hat, übergibt C_v das Dokument an C und die Suche ist abgeschlossen. Gibt C_v aber bekannt, dass D nicht in dessen Speicher liegt, wird i um eins erhöht, d.h. $i = i + 1$, und der Suchvorgang von C erneut mit $p = (h_x(\textit{D.name}, i), h_y(\textit{D.name}, i))$ gestartet. Dieser Vorgang wird solange wiederholt, bis C das Dokument D gefunden hat oder garantiert werden kann, dass D nicht im Netzwerk gespeichert ist.

Stufe 2: Ein Client C will Dokument D im Netzwerk hinzufügen / löschen (2 Punkte)
Um ein Dokument D einzufügen berechnet C mithilfe von $h_x(\textit{D.name}, i)$, $h_y(\textit{D.name}, i)$ und $i = 0$ die (vermeintliche) Einfügeposition $p = (x, y)$ des Dokuments D und sucht nach dem für p verantwortlichen Client C_v (siehe oben). C übergibt C_v das Dokument D um dieses zu speichern. Falls C_v weniger als M Dokumente verwaltet, speichert C_v Dokument D und das Einfügen ist abgeschlossen. Sonst wird C informiert, dass C_v über zu wenig Speicher verfügt, i wird um eins erhöht, d.h. $i = i + 1$, und C startet die Suche nach einer neuen Einfügeposition mit $p = (h_x(k, i), h_y(k, i))$ (siehe oben).

Um ein Dokument D zu löschen sucht C nach dem für Dokument D verantwortlichen Client C_v (siehe oben). Entweder findet C das Dokument, dann befiehlt C dem Besitzer C_v es aus dem Speicher zu löschen, oder das Dokument befindet sich nicht im P2P Netzwerk und der Vorgang wird abgebrochen.

Stufe 3: Ein Client C betritt das Netzwerk (4 Punkte)
Hier sind grundsätzlich zwei Fälle zu unterscheiden. Falls C der erste Client im Netzwerk ist, dann übernimmt C automatisch die Verantwortung für den gesamten Bereich $([0, X[; [0, Y[)$.

Falls C ein weiterer Client im Netzwerk ist, wird C ein vorgegebener Client C_e übergeben, der sich bereits im Netzwerk befindet und als Einstiegspunkt für C dient. Vor Betreten des Netzwerks hat sich C einen zufälligen Punkt $p = (x, y) \in ([0, X[; [0, Y[)$ ausgesucht, um den eigenen Verantwortlichkeitsbereich bestimmen zu können. C sucht mit Hilfe von C_e nach dem Client C_p, der für den Bereich, in den p fällt, zuständig ist (siehe oben). C fordert C_p auf den Bereich in zwei gleich große Hälften zu teilen, wobei drei Fälle zu unterscheiden sind.

Sei $([x_1, x_2[; [y_1, y_2[)$ der Bereich von C_p. Falls $|x_1 - x_2| \geq |y_1 - y_2|$ wird der Bereich normal zur X-Achse aufgeteilt. Als neuen Bereich wählt C_p darauf die linke Hälfte, d.h. $C_p.\textit{area} = ([x_1, x_1 + \frac{|x_1 - x_2|}{2}[; [y_1, y_2[)$, und C die rechte Hälfte, d.h. $C.\textit{area} = ([x_1 + \frac{|x_1 - x_2|}{2}, x_2[; [y_1, y_2[)$. C übernimmt alle rechten Nachbarn von C_p. C_p entfernt alle rechten Nachbarn und wählt C als einzigen neuen rechten Nachbarn. C übernimmt von C_p alle oberen und unteren Nachbarn $C' \in N(C_p)$ für die mindestens eine der folgenden Bedingungen gilt:

$$\begin{array}{rcccl} C'.\textit{area}.x_1 & \leq & C.area.x_1 & < & C'.\textit{area}.x_2 \\ C'.\textit{area}.x_1 & < & C.area.x_2 & \leq & C'.\textit{area}.x_2 \end{array}$$

C_p entfernt alle oberen und unteren Nachbarn, für die keine der Bedingungen mehr gilt.

Bedenken Sie, dass bei der Übernahme von neuen Nachbarn die Verbindungen seitens der Nachbarn selbst auch angepasst werden müssen! Ein Client C_a besitzt dann und nur dann eine Verbindung zu einem anderen Client C_b, wenn C_b auch eine Verbindung zu C_a besitzt (z.B. muss C auch C_p als einzigen linken Nachbarn wählen).

Falls $|x_1 - x_2| < |y_1 - y_2|$ wird normal zur Y-Achse geteilt. Der Teilungsprozess erfolgt analog zum Teilungsprozess, der normal zur X-Achse durchgeführt wird. C_p wählt in diesem Fall immer die obere Hälfte.

Falls für einen der geteilten Räume $x_2 - x_1 < 1$ oder $y_2 - y_1 < 1$ gelten würde, wird die Teilung nicht durchgeführt und C kann das Netzwerk nicht betreten.

Nachdem alle Nachbarschaften aktualisiert wurden, übergibt C_p an C alle Dokumente, die nicht mehr in den Bereich von C_p fallen. **WICHTIG!** Beim Aufteilen der Bereiche muss die Speicherplatzbeschränkung der Clients ignoriert werden! C und C_p setzen ihre neuen Positionen *C.pos* bzw. C_p*.pos* in die Mitte des jeweils verwalteten Bereichs. C hat damit das Betreten des Netzwerkes erfolgreich abgeschlossen.

Codegerüst

Der Algorithmus ist in Java 6 zu implementieren. Um Ihnen einerseits das Lösen der Aufgabenstellung zu erleichtern und andererseits automatisches Testen mit entsprechender Rückmeldung an Sie zu ermöglichen, stellen wir Ihnen ein Codegerüst zur Verfügung[1].

Das Codegerüst besteht aus mehreren Klassen, wobei Sie Ihren Code in die Datei `Client.java` einfügen müssen. Das fertige Programm führt vorgegebene Befehlssequenzen aus (siehe Abschnitt Testdaten), die Clients im Netzwerk anweisen Dokumente zu suchen, einzufügen, zu löschen oder das Netzwerk zu betreten. Um ein funktionsfähiges Programm zu erhalten, müssen Sie die unten angegebenen Funktionen der Interfaces `ClientInterface` und `ClientCommandInterface` in der Klasse `Client` implementieren. Beachten Sie, dass Sie die Funktionssignaturen (Rückgabewert, Parameter) nicht ändern dürfen. Im Speziellen gilt das für Funktionen, die Objekte des Typs `ClientInterface` entgegennehmen. Ebenso ist es nicht erlaubt einen Downcast auf Interface-Typen oder Abstract-Typen durchzuführen! Weiters beachten Sie, dass alle Änderungen im Code außerhalb von `Client.java` nicht von unserem Abgabesystem berücksichtigt werden.

Weitere Details, die das Codegerüst betreffen, entnehmen Sie bitte der LVA-Webseite[2]. Dort befindet sich auch der Link zur Dokumentation (Javadoc) des Codegerüstes.

Grundlegendes

- Dokumentnamen bestehen nur aus Kleinbuchstaben und sind vom Typ `String`.

[1] `http://www.ads.tuwien.ac.at/teaching/lva/186813.html#Programmieraufgabe`
[2] `http://www.ads.tuwien.ac.at/teaching/lva/186813.html`

- Client IDs werden aufsteigend beim Betreten des Netzwerkes vergeben: client0, client1, client2, ...
- Sowohl Client IDs als auch Dokumentnamen sind **eindeutig**.
- Sie dürfen keine `static` Membervariablen verwenden um Informationen auszutauschen!
- Es ist Ihnen erlaubt (und auch erwünscht) Java 6 SE Datenstrukturen (Collections) zur Verwaltung von Dokumenten bzw. Clients zu verwenden. Beachten Sie dabei jedoch das Laufzeitverhalten der unterschiedlichen Datenstrukturen!

Hashfunktionen
Sei $\texttt{ord}(\alpha)$ die Ordinalzahl des Zeichens α im Alphabet (beginnend bei 0), d.h. $\texttt{ord}(a) = 0$, $\texttt{ord}(b) = 1, \ldots, \texttt{ord}(z) = 25$ (keine Umlaute). Die zu implementierenden Hashfunktionen $h_x(\textit{D.name}, i)$ und $h_y(\textit{D.name}, i)$ berechnen sich daraus wie folgt:

Berechnen Sie zuerst die Summe S der Ordinalzahlen aller Zeichen im Dokumentnamen. Die Hashwerte ergeben sich dann aus:

$$\begin{aligned} M &= X \cdot Y \\ r_h(S,i) &= i \cdot ((2 \cdot (S \bmod (M-2))) + 1) \\ H(S,i) &= ((S \bmod M) + r_h(S,i)) \bmod M \\ h_x(\textit{D.name}, i) &= H(S,i) \bmod X \\ h_y(\textit{D.name}, i) &= \left\lfloor \frac{H(S,i)}{X} \right\rfloor \end{aligned}$$

Wobei X bzw. Y die Größe des Netzwerks entlang der horizontalen bzw. vertikalen Achse angeben.

Hilfsklassen
Das Framework verwendet einige Hilfsklassen, die Sie auch verwenden können bzw. müssen:

`Position` repräsentiert einen Punkt im Netzwerk. Mit `getX/Y` können Sie auf die Komponenten der Position zugreifen. Weiters wird eine `equals` Methode zur Verfügung gestellt, um zwei Positionen auf Gleichheit zu überprüfen.

`Area` repräsentiert einen Bereich des Netzwerks. Auf die Grenzen des Bereichs können Sie mit `getLowerX/Y`, entspricht x_1 und y_1, bzw. `getUpperX/Y`, entspricht x_2 und y_2, zugreifen. Die Klasse stellt die Methode `contains` zur Verfügung um zu überprüfen, ob eine Position innerhalb des Bereichs liegt. Weiters kann mit den Methoden `splitVertically` und `splitHorizontally` der Bereich in zwei neue Bereiche geteilt werden. Einmal erstellte Bereiche können nie verändert werden! Sie müssen daher immer ein neues Objekt erstellen.

`Document` repräsentiert ein Dokument im Netzwerk. Mit `getName` können Sie auf den Namen des Dokuments zugreifen.

`Pair<F,S>` ist ein Generic, das zwei Objekte mit Typen `F` und `S` speichern kann. Sie können über die `public` Membervariablen `first` und `second` auf die gespeicherten Werte zugreifen.

Methoden

Folgende Methoden des Interface `ClientInterface` müssen Sie in der Klasse `Client` implementieren. Falls Sie nicht alle Stufen der Programmieraufgabe implementieren, müssen Sie für jene Methoden die Sie nicht benötigen eine leere Implementierung angeben. Andernfalls wird Java Ihre Klasse nicht (in Bytecode) kompilieren! Falls Sie eine Methode, die einen Rückgabewert vorgibt, nicht benötigen, geben Sie `null` zurück. Einige Methoden müssen korrekt implementiert werden damit das Framework funktionsfähig ist. Falls das der Fall ist, ist es bei den folgenden Methodenbeschreibungen angegeben.

`Client(String uniqueID, int networkXSize, int networkYSize)`
Der Constructor um einen neuen Client zu erzeugen. `uniqueID` ist die eindeutige ID des neuen Clients. `networkXSize` ist die Größe des CANs entlang der horizontalen Achse. `networkYSize` ist die Größe des CANs entlang der vertikalen Achse.
Der Constructor muss implementiert werden!

`String getUniqueID()`
Gibt die – bei der Erzeugung übergebene – eindeutige ID des Clients zurück.
Diese Methode muss implementiert werden!

`Position getPosition()`
Gibt die Position des Clients im Netzwerk zurück (die Mitte des verwalteten Bereichs).

`Area getArea()`
Gibt den verwalteten Bereich zurück.
Diese Methode muss implementiert werden!

`Area setArea(Area newArea)`
Setzt den verwalteten Bereich des Clients. **ACHTUNG!** Bedenken Sie, dass mit dem Ändern des Bereichs sich auch die Position des Clients ändert!
Diese Methode muss implementiert werden!

`void setMaxNumberOfDocuments(int M)`
Setzt die Anzahl an speicherbaren Dokumenten für diesen Client auf `M`.
Diese Methode muss implementiert werden!

`int getMaxNumberOfDocuments()`
Gibt die Anzahl `M` an maximal speicherbaren Dokumenten zurück oder -1, falls diese Anzahl zuvor nicht festgelegt wurde.

`Document getDocument(String documentName) throws NoSuchDocument`
Retourniert das Dokument mit Namen `documentName`, falls sich

das Dokument im Speicher des Clients befindet. Sonst wird eine `ads1.ss14.can.exceptions.NoSuchDocument` Exception geworfen.
ACHTUNG! Diese Funktion darf nur für Clients aufgerufen werden, die das gesuchte Dokument auch tatsächlich speichern könnten. Der Aufruf dieser Funktion bei anderen Clients kann vom Abgabesystem als Fehler bewertet werden! Sie können daher nicht einfach mit Brute Force alle Clients nach einem Dokument durchsuchen. Sie müssen den Algorithmus wie in Abschnitt Funktionsweise beschrieben implementieren, ansonsten kann Ihre Abgabe beim Abgabegespräch negativ beurteilt werden!
Diese Methode muss implementiert werden!

`void storeDocument(Document d, Position p) throws NoAdditionalStorageAvailable`
Speichert das übergebene Dokument `d` mit der berechneten Position `p` im aufrufenden Client. Falls die Anzahl an speicherbaren Dokumenten für diesen Client überschritten wird, muss eine `ads1.ss14.can.exceptions.NoAdditionalStorageAvailable` Exception geworfen werden.
Diese Methode muss implementiert werden!

`void deleteDocument(String documentName) throws NoSuchDocument`
Löscht das Dokument mit Namen `documentName` aus dem Speicher des aufrufenden Clients. Falls der Client das Dokument nicht gespeichert hat, wird eine `ads1.ss14.can.exceptions.NoSuchDocument` Exception geworfen.

`ClientInterface searchForResponsibleClient(Position p)`
Retourniert den Client, in dessen Bereich die Position `p` fällt. Es werden keine ungültigen Positionen (außerhalb der Netzgrenzen) vom Framework übergeben.

`Iterable<ClientInterface> getNeighbours()`
Gibt alle Nachbarn des Clients zurück. **ACHTUNG!** Falls Sie Java 6 Collections verwenden, bedenken Sie, dass gleichzeitiges Iterieren und Modifizieren einer Datenstruktur zu unerwartetem Verhalten führen kann!
Diese Methode muss implementiert werden!

`void addNeighbour(ClientInterface newNeighbour)`
Fügt `newNeighbour` als neuen Nachbarn des Clients hinzu.
Diese Methode muss implementiert werden!

`void removeNeighbour(String clientID)`
Entfernt den Nachbarn mit der ID `clientID` aus der Menge der Nachbarn. Falls kein Nachbar mit dieser ID existiert, wird der Löschbefehl ignoriert.

`Iterable<Pair<Document, Position>> removeUnmangedDocuments()`
Diese Methode ist optional, d.h., Sie können sie leer lassen, falls Sie die Dokumente anders entfernen. Der aufrufende Client entfernt all jene Dokumente aus dem eigenen Speicher, für die er nicht zuständig ist, d.h. deren Speicherposition (wie in `storeDocument` übergeben) nicht mehr in seinen Bereich fallen. Der Client gibt die entsprechenden `<Document, Position>` Pairs (eine Klasse des Frameworks) als iterierbares Objekt zurück (z.B. eine `LinkedList` Java 6 Collection).

`void adaptNeighbours(ClientInterface joiningClient)`
: Diese Methode ist optional, d.h., Sie können sie leer lassen, falls Sie die Nachbarn eines Clients anders adaptieren. Der aufrufende Client fügt dem neuen Client (`joiningClient`) die entsprechenden Nachbarn hinzu und entfernt alle obsoleten Clients aus der eigenen Nachbarschaft. Alternativ kann die Methode auch dazu verwendet werden, um die gespeicherten Dokumente zu übergeben/entfernen.

Zusätzlich müssen Sie die Kommandos (Betreten, Dokument hinzufügen bzw. Dokument löschen), die ein Client ausführen kann, implementieren. Beachten Sie, dass nur der Client, der das Kommando ausführen soll, Zugriff hat (via `ClientCommandInterface`)!

`ClientInterface joinNetwork(ClientInterface entryPoint, Position p)`
: Der aufrufende Client betritt das Netzwerk mit Position `p` und mithilfe des Clients `entryPoint` (entspricht C_e in Abschnitt Funktionsweise). Gibt den Client zurück, mit dem sich der aufrufende Client den verwalteten Bereich teilt.

`void addDocumentToNetwork(Document d)`
: Der aufrufende Client speichert (falls möglich) das Dokument `d` im Netzwerk ab.

`void removeDocumentFromNetwork(String documentName)`
: Der aufrufende Client löscht (falls vorhanden) das Dokument mit dem Namen `documentName` aus dem Netzwerk.

`Document searchForDocument(String documentName)`
: Der aufrufende Client sucht nach dem Dokument mit dem Namen `documentName` und liefert es als Ergebnis zurück. Falls das Dokument nicht im Netzwerk gespeichert ist, wird `null` retourniert.

HINWEIS! Einige Methoden haben eine zusätzliche `throws CANException` clause. Diese wird ausschließlich vom Framework benötigt und betrifft Ihre Implementierung nicht.

Hinweise

`Main.printDebug(String msg)` kann verwendet werden um Debuginformationen auszugeben. Die Ausgabe erfolgt nur, wenn beim Aufrufen das Debugflag (`-d`) gesetzt wurde. Folgende Flags stehen zur Verfügung:

- `-d` ... ist das Debugflag gesetzt, wird der gerade ausgeführte Befehl ausgegeben.
- `-c` ... wird dieses Flag gesetzt, wird die gesamte CAN Struktur nach jedem Befehl, der diese ändert ausgegeben.
- `-s` ... bei gesetztem Flag wird die Suchhistorie nach jedem Suchbefehl ausgegeben.

Weitere Erläuterungen zur Ausgabe finden Sie im Abschnitt Testdaten. Sie müssen diese Ausgaben vor der Abgabe **nicht** entfernen, da das Testsystem beim Kontrollieren Ihrer Implementierung diese Flags nicht setzt.

Das Framework nutzt Methoden, die das Ausführen von *sicherheitsbedenklichem* Code auf dem Abgabesystem verhindern soll. Werden trotzdem solche Programmteile abgegeben oder sollte versucht werden, das Sicherheitssystem zu umgehen, wird das als Betrugsversuch gewertet. Die minimale Konsequenz dafür ist ein negatives Zeugnis auf diese Lehrveranstaltung.

Rückgabe des Frameworks Ist Ihre Lösung korrekt gibt das Framework `OK` zurück. Andernfalls bekommen Sie eine Fehlermeldung mit einem Hinweis zur Ursache.

Testdaten

Auf der Webseite zur LVA sind auch Testdaten veröffentlicht, die es Ihnen erleichtern sollen, Ihre Implementierung zu testen. Verarbeitet Ihr Programm diese Daten korrekt, heißt das aber nicht zwangsläufig, dass Ihr Programm alle Eingaben korrekt behandelt. Testen Sie daher auch mit zusätzlichen, selbst erstellten Daten.

Die Testdaten beschreiben Befehlssequenzen an das Netzwerk mit folgendem Format:

1. Zeile: `network XSIZE YSIZE`
   Gibt die Größe des Netzwerkes entlang der horizontalen (`XSIZE`) bzw. vertikalen (`YSIZE`) Achse an. Beide Größenangaben müssen vom Typ Integer sein.

`join ID ENTRYID XPOS YPOS M`
: Ein neuer Client `ID` betritt das Netzwerk mit Hilfe des Clients `ENTRYID`. Falls `ENTRYID` dem String `"null"` entspricht, dann eröffnet der neue Client ein neues Netzwerk der Größe $([0, \texttt{XSIZE}[; [0, \texttt{YSIZE}[)$. Die Position, an der sich der neue Client einfügen möchte, wird durch `XPOS` und `YPOS` bestimmt. `M` gibt die Anzahl der Dokumente an, die der neue Client speichern kann.

`add ID DOCNAME`
: Der Client `ID` möchte ein neues Dokument `DOCNAME` in das Netzwerk einfügen.

`delete ID DOCNAME`
: Der Client `ID` möchte das Dokument `DOCNAME` aus dem Netzwerk löschen.

`search ID DOCNAME`
: Der Client `ID` möchte das Dokument `DOCNAME` im Netzwerk suchen.

`begin_test_can`
: Beginnt eine Test-Section, die die gesamte erwartete Struktur des Netzwerks zu diesem Zeitpunkt angibt. Die tatsächliche Struktur des Netzwerks kann damit überprüft werden. Weiters werden alle Anweisungen, die an das Netzwerk – seit der letzten Strukturmodifikation – gestellt wurden, auf ihre Richtigkeit überprüft. Das Format der Test-Section sieht wie folgt aus:

```
begin_test_can
CLIENT_ID_1 : NEIGHBOUR_ID_1, ..., NEIGHBOUR_ID_N
```

```
CLIENT_ID_2 : NEIGHBOUR_ID_1, ..., NEIGHBOUR_ID_N
...
--- stored documents ---
CLIENT_ID_1 : DOC_NAME_1, DOC_NAME_2, ..., DOC_NAME_N
CLIENT_ID_2 : DOC_NAME_1, DOC_NAME_2, ..., DOC_NAME_N
...
end_test_can
```

Der `stored documents` Abschnitt gibt für jeden Client alle gespeicherten Dokumente an. Die Reihenfolgen spielen hierbei keine Rolle.

`test_search ID ID ...`
: Testet die Suchhistorie des letzten ausgeführten Suchbefehls. Die Liste der `ID`s gibt die Reihenfolge der besuchten Clients an. Ein Client gilt als "besucht", wenn seine `searchForDocument`, `searchForResponsibleClient` oder `getDocument` Methode aufgerufen wird. Sollte derselbe Client mehrmals direkt hintereinander besucht werden, scheint dieser trotzdem nur einmal in der Liste auf. Ein `null` als `ID` bedeutet, dass der vorangegangene Client nach dem gesuchten Dokument gefragt, (`getDocument`) es jedoch nicht gespeichert hatte. Nach einem `null` folgt immer der nach dem Dokument suchende Client, da dieser ein Rehashing durchführen muss und die Suche erneut beginnt.

`begin_load_can`
: Lädt die vorgegebene Struktur eines CAN Netzwerkes. Die bisherige Struktur wird verworfen. Entspricht großteils dem `begin_test_can` Befehl.

```
begin_test_can
CLIENT_ID_1 <PARENT, M>: NEIGHBOUR_ID_1, ...
CLIENT_ID_2 <PARENT, M>: NEIGHBOUR_ID_1, ...
...
--- stored documents ---
CLIENT_ID_1 : DOC_NAME_1, DOC_NAME_2, ..., DOC_NAME_N
CLIENT_ID_2 : DOC_NAME_1, DOC_NAME_2, ..., DOC_NAME_N
...
end_test_can
```

`PARENT` gibt die ID jenes Clients an, von dem der beschriebene Client seinen Bereich erhalten hat (C_p in der Beschreibung des Betretens des Netzwerks). Der `PARENT` des ersten Clients ist `null`. `M` gibt die Anzahl der speicherbaren Dokumente an.

Sie können davon ausgehen, dass **alle** von uns erstellten Testinstanzen dieser Spezifikation entsprechen, und müssen selbst keine Fehlerprüfung durchführen.

Abgabe

Die Abgabe Ihrer Implementierung der Klasse `Client` erfolgt über TUWEL. Bedenken Sie bitte, dass Sie maximal 30-mal abgeben können und ausschließlich die jeweils letzte Abgabe bewertet wird. Prinzipiell sollte es nicht nötig sein, mehr als eine Abgabe zu tätigen, wenn Sie Ihr Programm entsprechend getestet haben.

Hinweis

Verwenden Sie bei Ihrer Implementierung unter keinen Umständen Sonderzeichen, wie zum Beispiel Umlaute (ä, ö, ü, Ä, Ö, Ü) oder das Zeichen „ß“. Dies kann sonst – bei unterschiedlichen Zeichensätzen am Entwicklungs- und Abgabesystem – zu unvorhersehbaren Problemen und Fehlern führen, was schlussendlich auch als fehlerhafter Abgabeversuch gewertet wird.

Die Überprüfung Ihres abgegebenen Codes erfolgt automatisch, wobei in drei Schritten getestet wird:

Kompilation Es wird der **Bytecode** erzeugt, der beim anschließenden Testen verwendet wird.

veröffentlichte Testdaten Bei diesem Schritt wird Ihr Programm mit den auf der Webseite veröffentlichten Daten getestet.

unveröffentlichte Testdaten Abschließend wird Ihr Programm noch mit Ihnen nicht bekannten aber den Spezifikationen entsprechenden Eingabedaten ausgeführt.

Nach Beendigung der Tests, die direkt nach Ihrer Abgabe gestartet werden, erhalten Sie eine Rückmeldung über das Abgabesystem in TUWEL mit entsprechenden kurzen Erfolgs- bzw. Fehlermeldungen.

Aufgrund der großen Hörerzahl kann es zu Verzögerungen beim Verarbeiten Ihrer Abgaben kommen. Geben Sie daher Ihre Lösung nicht erst in den letzten Stunden ab, sondern versuchen Sie, rechtzeitig die Aufgabenstellung zu lösen. Beachten Sie bitte auch, dass wir Ihren Code mit den von Ihren Kollegen abgegebenen Programmen automatisch vergleichen werden, um Plagiate zu erkennen. Geben Sie daher nur selbst implementierte Lösungen ab!

Punktevergabe

Für jede der drei Stufen gibt es Testinstanzen, die die Funktionalität Ihrer Lösung überprüfen. Arbeitet Ihr Programm **alle** Testinstanzen für eine Stufe ab, wird diese vom Abgabesystem als korrekt bewertet und Sie haben die Möglichkeit beim Abgabegespräch die Punkte für diese Stufe zu erhalten. Bitte beachten Sie, dass eine korrekte Abgabe von Stufe 1 notwendig ist, um überhaupt ein Abgabegespräch führen zu können und somit die Programmieraufgabe positiv abzuschließen. Die Testinstanzen teilen sich wie folgt auf:

- Stufe 1: **0000**, **0001**, 0002, 0003, 0004
  Diese Instanzen testen die Suche eines Clients nach Dokumenten.

- Stufe 2: **0005**, 0006, 0007, 0008, 0009
  Diese Instanzen testen die Suche eines Clients nach Dokumenten und darüber hinaus das Einfügen bzw. Löschen von Dokumenten.
- Stufe 3: **0010**, **0011**, 0012, 0013, 0014
  Diese Instanzen testen die Suche eines Clients nach Dokumenten, das Einfügen bzw. Löschen von Dokumenten und das Betreten eines neuen Clients in das Netzwerk.

Die hervorgehobenen Instanzen sind öffentlich. Sollten Sie Stufe 2 und 3 korrekt lösen aber Stufe 1 nicht (d.h. ein besonderer Randfall wurde nicht abgedeckt), wird Ihre Lösung als negativ bewertet. Sollten Sie Stufe 1 und 3 korrekt lösen aber Stufe 2 nicht (da wiederum ein Randfall nicht abgedeckt wurde), können Sie beim Abgabegespräch die Punkte für Stufe 1 und 3 erhalten nicht aber für Stufe 2.

Theoriefragen (1 Punkt)

Beim Abgabegespräch müssen Sie u.a. Ihre Überlegungen zu folgenden Punkten präsentieren können:

- Wann kann die Suche nach einem Dokument D garantiert abgebrochen werden, wenn es nicht im Netzwerk gespeichert ist?
- Was muss bei der Wahl der Dimensionen des Netzwerks in Bezug auf die gegebenen Hashfunktionen beachtet werden?
- Wieso wurden keine Hashfunktionen basierend auf Primzahlen verwendet?
- Was ist beim Sondieren im 2-dimensionalen Raum zu beachten? Wäre simples lineares Sondieren möglich?
- Wie viele Clients können an einem Netzwerk der Größe $([0, X[, [0, Y[) \in \mathbb{N}^2$ maximal teilnehmen und Dokumente speichern? Wieso ist diese Zahl beschränkt?
- Geben Sie eine möglichst gute obere Schranke für die Laufzeit ihres Algorithmus für die Suche nach einem Dokument, in Abhängigkeit relevanter Kenngrößen, in $\mathcal{O}$-Notation an.
- Was müsste man beim Entfernen eines Clients aus dem Netzwerk alles beachten?

Abgabesystem

Unser Abgabesystem mit Intel Xeon X5650 CPU ruft Ihr Programm mit folgendem Kommandozeilenbefehl auf:

```
java ads1.ss14.can.Main input > output
```

wobei `input` eine Eingabedatei darstellt. Die Ausgabe wird in die Datei `output` gespeichert. Pro Testinstanz darf eine maximale Ausführungszeit von 15 Sekunden nicht überschritten werden, anderenfalls wird die Ausführung abgebrochen.

Bitte beachten Sie, dass unser Abgabesystem **kein** Debugging-Tool ist!

Bewertung

Abschließend seien nochmals die wesentlichen Punkte zusammengefasst, die in die Bewertung Ihrer Abgabe einfließen und schließlich über die Anzahl der Punkte für diese Programmieraufgabe entscheiden:

- Anzahl der korrekt gelösten Teilaufgaben
- Laufzeiteffizienz
- Speicherverbrauch
- Rechtfertigung des Lösungsweges
- Antworten auf die theoretischen Fragen der Aufgabenstellung

Voraussetzung für eine positive Absolvierung des Abgabegesprächs ist jedenfalls grundsätzliches Verständnis der Problemstellung.

Zusätzliche Informationen

Lesen Sie bitte auch die auf der Webseite zu dieser LVA veröffentlichten Hinweise. Wenn Sie Fragen oder Probleme haben, wenden Sie sich rechtzeitig an die AlgoDat1-Hotline unter `algodat1-ss14@ads.tuwien.ac.at`.