JOHANNES GUTENBERG UNIVERSITÄT MAINZ

Splay trees und Link/Cut trees

von

Monika Schömer

Matrikel-Nr.: 2714940
Bachelorarbeit in Informatik
vorgelegt dem Fachbereich Physik, Mathematik und Informatik (FB 08)
der Johannes Gutenberg-Universität Mainz
am 20. August 2018

1. Gutachter: Prof. Dr. Ernst Althaus
2. Gutachter: Dr. Stefan Endler

Hiermit erkläre ich, _____ (Matr.-Nr.:) _____, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen oder Hilfsmittel (einschließlich elektronischer Medien und Online-Quellen) benutzt habe. Mir ist bewusst, dass ein Täuschungsversuch oder ein Ordnungsverstoß vorliegt, wenn sich diese Erklärung als unwahr erweist. § 18 Absatz 3 und 4 der o. g. Ordnungen gilt in diesem Fall entsprechend.

Mainz, den 20.August 2018

Monika Schömer

Institut für Informatik
Staudingerweg 9
Johannes Gutenberg-Universität D-55099 Mainz
Matrikel-Nr.: 2714940
mschoeme@students.uni-mainz.de

Inhaltsverzeichnis

1. Einleitung **2**

2. Splay trees **4**
2.1. Was ist ein Splay tree? 4
2.2. Rotationen im Splay tree 4
2.3. Die Implementation primitiver Funktionen 7
2.4. Die Laufzeitanalyse von splay() 9
2.5. Die Optimalität der Splay trees 20

3. Link/Cut trees **25**
3.1. Dinic und das Problem des maximalen Flusses 25
3.2. Was ist ein Link/Cut tree? 27
3.3. Die Datenstruktur der Link/Cut trees 28
3.3.1. Der Original und der Virtual Tree 28
3.3.2. Die splice-Operation 30
3.3.3. Access() im Virtual tree 31
3.3.4. Handhabung der Kostenwerte im Virtual Tree 34
3.4. Funktionen und ihre Implementation 38
3.5. Die Laufzeitanalyse von Access() 41
3.6. Zurück zum Problem des maximalen Flusses 44

4. Zusammenfassung und Ausblick **50**

A. Literaturverzeichnis **52**

1. Einleitung

In der Informatik interessieren uns Datenstrukturen, die einen schnellen Zugriff auf ein angefragtes Element garantieren. Angenommen, wir wollen einige Elemente verwalten (zunächst in einer Liste) und wüssten auch, wie oft auf jedes Element später einmal zugegriffen wird. Das heißt, die Zugriffshäufigkeiten sind bekannt. Dann ist es natürlich sinnvoll, die Elemente entsprechend ihrer Zugriffshäufigkeiten zu sortieren. So setzen wir das Element, welches am häufigsten angefragt wird, an den Anfang der Liste und das Element, welches am seltensten angefragt wird an das Ende der Liste. Dieser statische Aufbau der Liste liefert uns offensichtlich die bestmögliche Laufzeit, aber nur für Zugriffssequenz, die den gegebenen Zugriffshäufigkeiten entspricht. In der Realität sind diese Häufigkeiten jedoch meistens nicht bekannt. Wir benötigen also eine dynamische Datenstruktur, die sich selbstständig auch unbekannten Zugriffshäufigkeiten anpasst und dabei stets einen schnellen Zugriff auf Elemente gewährleistet. Eine solche Listen-Datenstruktur erreichen wir, indem wir nach jeder Anfrage auf ein Element, jenes besagte Element an den Anfang der Liste setzen. Durch diese kleine Zusatzoperation erhalten wir, dass häufig angefragte Elemente weiter vorne in der Liste stehen und selten angefragte Elemente weiter hinten. Diese Liste heißt MoveToFront-Liste und liefert sogar eine beweisbar gute Zugriffslaufzeit, zumindest amortisiert betrachtet. Datenstrukturen, die sich eigenständig den unbekannten Zugriffshäufigkeiten der Elemente anpassen, nennt man selbstorganisierende Datenstrukturen. In dieser Arbeit werden im zweiten Kapitel die sogenannten Splay trees behandelt. Ein Splay tree ist ein binärer Suchbaum und auch eine selbstorganisierende Datenstruktur. Die Datenstruktur der Splay trees wurde von Daniel Sleator und Robert Tarjan erfunden und zunächst unter dem Namen „Self-Adjusting Binary Search Trees“ [8] im Jahr 1985 vorgestellt. Tatsächlich arbeitet ein Splay tree analog zur MoveToFront-Liste. Sobald ein Element des Baumes angefragt wird, wird besagtes Element durch eine Operation namens *splay* zur Wurzel des Baumes befördert. Die Wurzel des Baumes entspricht dabei dem Anfang der Liste, da das Element, welches in der Wurzel des Baumes steht, die geringste Zugriffslaufzeit besitzt. Es folgt zudem wie bei der MoveToFront-Liste, dass häufig angefragte Elemente des Splay tree höher im Baum stehen, als selten angefragte Elemente. Wir werden uns intensiv mit der splay-Operation befassen, sowie mit der Implementation der Funktionen, die durch einen Splay tree unterstützt werden. Alle Funktionen werden wir natürlich hinsichtlich ihrer Laufzeit analysieren. Am Ende des zweiten Kapitels werden wir zudem zeigen, dass ein Splay tree asymptotisch gesehen genauso effizient ist, wie ein statisch optimal aufgebauter binärer Suchbaum. Dabei kennt der Splay tree im Gegensatz zum statisch optimalen Baum die Zugriffshäufigkeiten auf seine Elemente nicht.
Nachdem wir im zweiten Kapitel die Datenstruktur der Splay trees umfassend beleuchtet haben, werden wir uns im dritten Kapitel mit der Datenstruktur der Link/Cut trees befassen. Die Link/Cut trees verwalten einen dynamischen Wald, also eine Menge an Bäumen, die zusammenwachsen (link) und getrennt (cut) werden können. Dabei benutzen die Link/Cut trees die Datenstruktur der Splay trees als einen Baustein. Auch den Link/Cut trees werden wir uns ausführlich widmen. Neben Aufbau und Struktur untersuchen wir die Funktionen sowie deren Implementation und analysieren alle Operationen, die auf einem Link/Cut tree ausgeführt werden können im Hinblick auf ihre Laufzeit. Insbesondere wollen wir uns mit einer speziellen Anwendung der Link/Cut trees beschäftigen, nämlich der Laufzeitverbesserung des Dinic-Algorithmus zur Berechnung eines maximalen Flusses.
Diese Arbeit beruht auf der Ausarbeitung von Saxena [6] zum Thema Splay trees und Link/Cut

trees. Ziel dieser Arbeit ist es, das Thema Splay trees und Link/Cut trees aufzubereiten. Dabei wird auf die Ausarbeitung von Saxena [6] zum diesem Thema aufgebaut und diverse Inhalte eigenständig vertieft, unter Anderem die Optimalität der Splay trees, die Anwendung der Link/Cut trees auf das Flussproblem sowie die Laufzeitanalyse der Funktionen eines Splay trees und der access-Operation der Link/Cut trees. Zusätzlich werden hierbei fachlich anspruchsvolle Abschnitte durch Visualisierungen, Beispiele und ausführliche Erklärungen gestützt. Vor allem die Lehre interessiert sich an einem solch aufbereitetem Thema, da sowohl die Splay trees als auch die Link/Cut trees ein großes Potential zum Verstehen grundlegender und vertiefender Strukturen der theoretischen Informatik bieten.

2. Splay trees

2.1. Was ist ein Splay tree?

Wir wollen in diesem Kapitel die Datenstruktur der Splay trees betrachten. Splay trees sind binäre Suchbäume, das heißt, jeder Knoten besitzt einen Schlüssel und höchstens ein linkes und höchstens ein rechtes Kind. Hierbei gilt die Suchbaum-Eigenschaft, dass Knoten des linken Teilbaums eines Knotens x nur kleinere Schlüssel als x besitzen und dass die Knoten des rechten Teilbaums eines Knotens x nur größere Schlüssel besitzen. Wie jeder Suchbaum sollen auch die Splay trees einige Funktionen ausführen können. Sei x ein Knoten und t, t_1 sowie t_2 Splay trees. Dann werden die Funktionen $search(t, x)$, $insert(t, x)$, $delete(t, x)$ sowie $join(t_1, t_2)$ und $split(t, x)$ durch die Datenstruktur der Splay trees unterstützt. Die Funktion $join(t_1, t_2)$ verbindet zwei Splay trees t_1 und t_2 zu einem einzigen Splay tree, wobei alle Schlüssel aus t_1 kleiner sind als die Schlüssel aus t_2. Die Funktion $split(t, x)$ hingegen teilt einen Splay tree t in zwei Splay trees auf. Dabei sollen alle Knoten, die einen kleineren oder gleichen Schlüssel haben wie der Knoten x, in einem Baum sein und alle Knoten mit größerem Schlüssel im anderen. Nun ist die Eigenschaft einiger Suchbäume, die wir kennen, dass sie höhenbalanciert sind. Ein Beispiel dafür sind die AVL-Bäume. Ein balancierter Suchbaum gewährleistet für Operationen auf dem Baum eine Laufzeit, die logarithmisch in der Zahl der Knoten des Baumes ist. Jedoch müssen wir auch stets in jedem Schritt den Baum modifizieren, um ihn balanciert zu halten. Wir werden im Folgenden sehen, dass sich Splay trees mehr oder weniger durch die Operation *splay*() selbst balancieren. Dabei ist *splay*() eine Unterfunktion, die von allen anderen Funktionen benutzt wird. Durch den Aufruf von $splay(x)$ wird der Knoten x zur Wurzel des gesamten Splay trees befördert. Das klingt erst einmal ineffizient, denn durch diese Operation können lange Pfade im Baum entstehen und somit würde die Laufzeit aller Operationen auf diesem langen Pfad wachsen. Jedoch werden wir in den folgenden Abschnitten sehen, dass die Laufzeit der splay-Funktion amortisiert in $\mathcal{O}(log(n))$ liegt, wobei n die Anzahl der Knoten im Baum beschreibt. Da jede Funktion die *splay*-Funktion als Subroutine benutzt und diese die Laufzeit der Funktion dominiert, gilt für die Splay trees, dass die Laufzeit jeder Funktion amortisiert in $\mathcal{O}(log(n))$ ist. Eine weitere nützliche Eigenschaft der Splay trees folgt direkt aus der splay-Operation. Da aufgerufene Knoten zur Wurzel des Baumes befördert werden, stehen häufig aufgerufene Knoten höher im Baum als Knoten, die selten aufgerufen werden. Die Laufzeit für den Zugriff auf einen Knoten x wird also geringer, je öfter dieser aufgerufen wird. Mit dieser Eigenschaft werden wir die statische Optimalität der Splay trees zeigen.

2.2. Rotationen im Splay tree

Wir wollen uns zunächst mit der Frage befassen, was genau bei dem Aufruf von $splay(x)$ passiert. Durch $splay(x)$ wird der Knoten x zur Wurzel des Baumes. Ähnlich wie bei einem AVL-Baum rotieren wir Teilbäume, die x enthalten. Dabei wird x im gesamten Baum an Höhe gewinnen. Hängen wir nun viele solcher Rotationen aneinander, wird x schlussendlich die Wurzel des gesamten Baumes sein. Aus Gründen, die der besseren Laufzeitanalyse dienen, werden wir zwischen Doppelrotationen und Einzelrotationen unterscheiden. Bei den Doppelrotationen ist insbesondere die Position von x im Teilbaum ausschlaggebend für die Art der Rotation. Deshalb

betrachten wir im Folgenden drei verschiedene Fälle. Außerdem müssen wir darauf achten, dass die Schlüssel der Knoten nach den Rotationen noch richtig sortiert sind. Die folgende Fallanalyse orientiert sich an der Ausarbeitung von J. Erickson zu diesem Thema [3].

1. Fall - Zag case

Wir orientieren uns für diesen Fall an der Struktur eines Teilbaumes, wie sie in Abbildung 2.1 gezeigt wird. Der Knoten x ist also das linke Kind der Wurzel y. Dann ist nur noch eine Einzelrotation nötig, damit x in der Wurzel steht. Man nennt diesen Schritt auch den Zag case. Wir lösen also die Kante von y nach x auf und hängen y mitsamt dem Teilbaum C an x als rechtes Kind an. Da x das linke Kind von y war, ist y größer als x. Damit die Suchbaum-Eigenschaft erhalten bleibt, muss nun also y das rechte Kind von x werden. Somit löst sich aber auch die Kante von x nach B auf. Jedoch können wir B einfach als linkes Kind an y hängen. Das ist legitim, da y nach Auflösen der y-x Kante kein linkes Kind mehr hat. Auch hier bleibt die Suchbaum-Eigenschaft erhalten, da B zuvor im linken Teilbaum von y zu finden war. Der Fall, dass x das rechte Kind der Wurzel y ist (Zig case), wird hier nicht aufgeführt, verläuft aber analog.

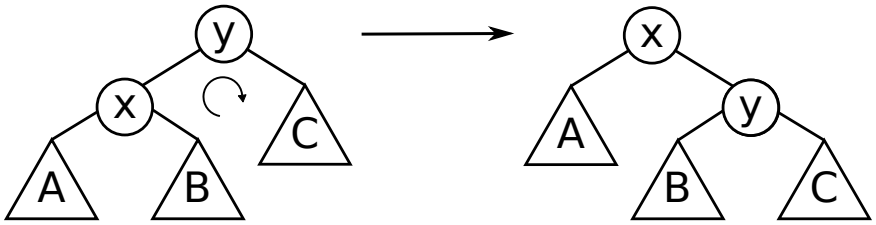


Abbildung 2.1.: Zag case: y ist die Wurzel und x das linke Kind von y - Es wird die y-x Kante rotiert.

Sei x nun kein Kind der Wurzel. Dann können wir bedenkenlos eine Doppelrotation durchführen. Jedoch sind auch hier zwei Fälle zu unterscheiden, je nach Position von x im Teilbaum.

2. Fall - Zag-Zag case

Seien nun sowohl x als auch $y = parent(x)$ beides linke Kinder ihres Vaters. Hierbei soll, wie in der nachfolgenden Abbildung 2.2 gezeigt, y der Vater von x und z der Vater von y sein. Nun ist eine Doppelrotation nötig, damit x in der Wurzel dieses Teilbaums steht. Man nennt diesen Schritt auch Zag-Zag case. Wir rotieren zuerst die y-z Kante. Dadurch steht nun y am höchsten im Teilbaum und hat als linkes Kind weiterhin x, jedoch als neues rechtes Kind z. Da z sein linkes Kind verloren hat, erhält es nun den vorherigen rechten Teilbaum C von y. Nun wird die y-x Kante rotiert, wodurch x nun am höchsten im Teilbaum steht und als rechtes Kind y hat. Als linkes Kind bekommt y nun den Teilbaum B. Durch diese zwei Rotationen bleibt die Suchbaum-Eigenschaft offensichtlich erhalten. Der Fall, bei dem sowohl x als auch y beides rechte Kinder sind (Zig-Zig case), verläuft analog.

3. Fall - Zig-Zag case

Sei nun eine Struktur wie in Abbildung 2.3 gegeben. Dann ist x das rechte Kind von y und y das linke Kind von z. Wir wollen wieder eine Doppelrotation durchführen, damit x in der Wurzel des Teilbaums steht. Man nennt diesen Schritt auch Zig-Zag case. Wir rotieren zuerst die y-x Kante. Dadurch wird y zum linken Kind von x. Der Teilbaum B, der zuvor das linke Kind von

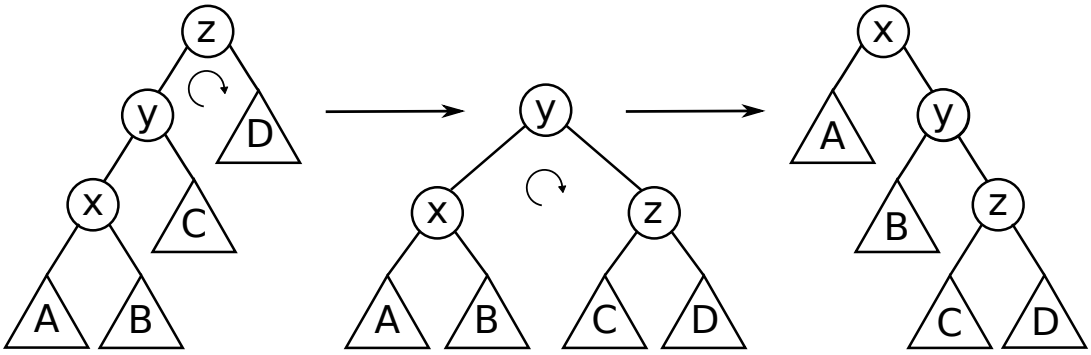


Abbildung 2.2.: Zag-Zag case: x ist das linke Kind von y und y ist das linke Kind von z - Zuerst wird die z-y Kante rotiert und dann die y-x Kante.

x war, wird nun zum rechten Kind von y. Somit bleibt auch hier die Suchbaum-Eigenschaft erhalten. Nun wird als nächstes die z-x Kante rotiert, wodurch x nun die Wurzel des Teilbaums ist und als rechtes Kind z hat. Da der Knoten z sein linkes Kind x verloren hat, bekommt er nun stattdessen den Teilbaum C. Somit ist gewährleistet, dass alle Schlüssel richtig sortiert bleiben. Im Zag-Zig case, bei dem x das linke Kind von y und y das rechte Kind von z ist, verfährt man analog.

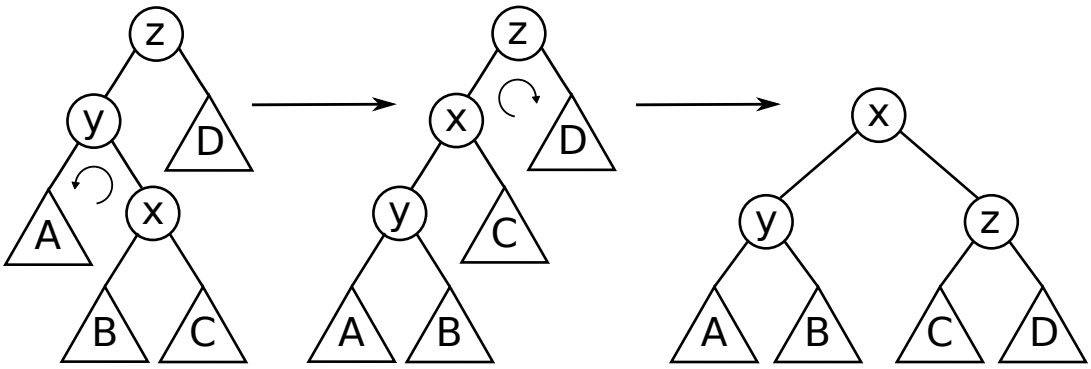


Abbildung 2.3.: Zig-Zag case: x ist das rechte Kind von y und y ist das linke Kind von z - Es wird zunächst die y-x Kante und dann die z-x Kante rotiert.

Wir werden nun also stets den Vater und den Großvater des Knoten x betrachten und nach den oben beschriebenen Fällen rotieren, solange bis x die Wurzel ist. Da x in allen Fällen um mindestens eine Höhenposition aufsteigt, wird x schlussendlich die Wurzel des gesamten Baumes sein. Erwähnenswert ist hierbei noch, dass die Einzelrotation aus dem ersten Fall höchstens einmal durchgeführt werden muss, denn der Knoten x kann höchstens einmal das Kind der Wurzel sein. Im nächsten Abschnitt wollen wir uns mit der Frage beschäftigen, wie die Operation $splay(x)$ uns hilft, alle anderen Operationen zu realisieren.

2.3. Die Implementation primitiver Funktionen

Wie jeder andere Suchbaum sollen auch Splay trees einige Operationen wie zum Beispiel $search(t, x)$, $insert(t, x)$ und $delete(t, x)$ in amortisiert $\mathcal{O}(log(n))$ unterstützen. Hierbei soll t ein Splay tree, n die Zahl der Knoten in t und x ein Knoten sein. Die Besonderheit an den Splay trees ist jedoch, dass wir einen aufgerufenen Knoten x zunächst durch den Aufruf von $splay(x)$ zur Wurzel befördern. Wie das funktioniert, haben wir im vorherigen Abschnitt gesehen. Ist der Knoten x nun die Wurzel, sind alle Operationen auf x in $\mathcal{O}(1)$ möglich. Wir werden die Vorgehensweise bei den verschiedenen Operationen nun genauer betrachten.

search(t,x)

Wir wollen den Knoten x im Baum t suchen. Dazu gehen wir so vor, wie von der normalen Suche in einem binären Suchbaum (BST-search) gewohnt. Diese Suche liefert uns entweder den Knoten x zurück oder den Knoten y, an dem die Suche gescheitert ist. Den zurückgelieferten Knoten beförden wir durch den Aufruf von $splay(x)$ bzw. $splay(y)$ zur Wurzel des Baumes.

insert(t,x)

Wir wollen den Knoten x in den Splay tree t einfügen. Dazu suchen wir x mit der BST-search im Baum und finden somit den Ort, an dem der Knoten x als Blattknoten eingefügt werden kann. Dort fügen wir den Knoten ein und rufen noch $splay(x)$ auf.

delete(t,x)

Durch diese Operation wollen wir den Knoten x aus dem Baum t entfernen. Dazu suchen wir zunächst den Knoten x mit der BST-search und machen ihn durch $splay(x)$ zur Wurzel. Jetzt löschen wir den Knoten x, wodurch der Baum in zwei Bäume t_L und t_R zerfällt. Wir müssen uns also eine neue Wurzel für t suchen. Da wir wissen, dass der Baum t_L ausschließlich kleinere Schlüssel besitzt als der Baum t_R, können wir einfach das größte Element von t_L zur neuen Wurzel von t machen. Dazu laufen wir einfach in t_L immer in den rechten Teilbaum, bis wir das größte Element w gefunden haben. Durch den Aufruf $splay(w)$ befördern wir w zur Wurzel von t_L. Da w das größte Element in t_L war, wird w kein rechtes Kind besitzten. Nun können wir aber einfach den Baum t_R als rechtes Kind an w anhängen. Dadurch haben wir erreicht, dass der Knoten x aus dem Baum t gelöscht wurde. Vergleiche hierzu auch folgende Abbildung 2.4.

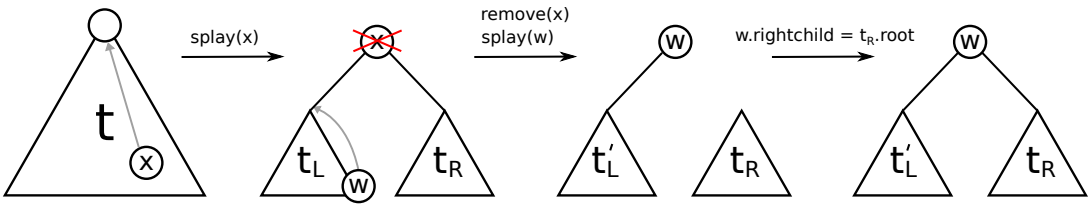


Abbildung 2.4.: Ausführung der Operation $delete(t, x)$

join(t_1,t_2)

Wir bekommen nun als Eingabe zwei Bäume t_1 und t_2, wobei alle Schlüssel aus t_1 kleiner sind als aus t_2. Wir wollen nun einen Splay tree t zurückgeben, der sowohl die Knoten von t_1 als auch von t_2 enthält. Dazu machen wir den Knoten w mit dem größten Schlüssel

aus t_1 durch den Aufruf von $splay(w)$ zur Wurzel von t_1. Jetzt hat die Wurzel von t_1 kein rechtes Kind, da w ja das größte Element in dem Baum t_1 war. Nun können wir einfach die Wurzel von t_2 als rechtes Kind an w anhängen, denn nach Voraussetzung waren ja alle Schlüssel aus t_2 größer als die Schlüssel aus t_1. Das heißt insbesondere auch größer als der Schlüssel von w. Insgesamt erhalten wir den gewünschten Splay tree t, der aus t_1 und t_2 zusammengefügt wurde. Eine Ausführung der join-Funktion ist in Abbildung 2.5 dargestellt.

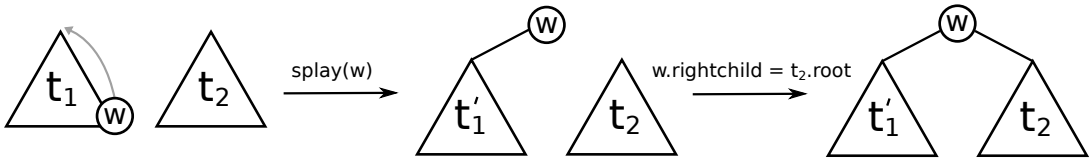


Abbildung 2.5.: Ausführung der Operation $join(t_1, t_2)$

split(t,x)

Wir sollen nun den Splay tree t in zwei Bäume t_1 und t_2 aufteilen und zwar so, dass in t_1 alle Elemente kleiner oder gleich x liegen und in t_2 alle Elemente größer x. Dazu führen wir die Operation $splay(x)$ durch. Nach $splay(x)$ ist nämlich x die Wurzel von t und somit liegen alle Elemente die kleiner als x sind im linken Teilbaum von x und alle Elemente die größer sind im rechten. Jetzt müssen wir nur noch x von seinem rechten Kind trennen, um die zwei Bäume t_1 und t_2 zu erhalten. Dadurch wird außerdem gewährleistet, dass x in dem Baum liegt, in dem auch alle Elemente liegen, die kleiner sind als x. Eine Ausführung der split-Funktion ist in Abbildung 2.6 dargestellt.

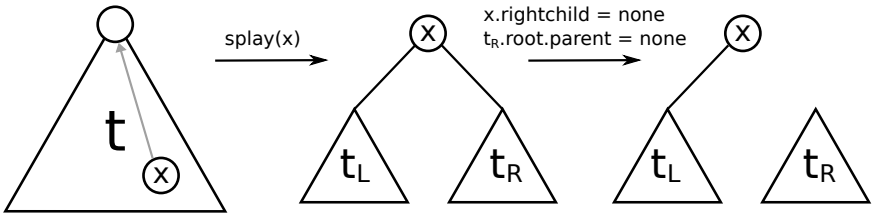


Abbildung 2.6.: Ausführung der Operation $split(t, x)$

Wir haben nun die Realisierung der unterstützten Funktionen näher betrachtet, jedoch bleibt die Frage nach der Laufzeit der Funktionen offen. Dabei verhalten sich search, insert, delete, join und split ähnlich. In jedem Fall suchen wir ein bis zwei Elemente, indem wir den Baum von der Wurzel aus in die Tiefe laufen. Ein gefundenes Element x wird zudem durch den Aufruf $splay(x)$ zur Wurzel befördert. Das passiert, indem wir den Baum von x aus in die Höhe laufen. Das heißt, dass wir bei der Suche nach einem Element und bei dem darauffolgenden splay-Aufruf dieses Elementes denselben Weg ablaufen. Da wir aber im splay-Aufruf noch viele Rotationen durchführen, dominiert die Laufzeit der splay-Operation die Laufzeit der Suche. Die Suche nach dem Knoten x kann bei der Laufzeitbetrachtung also einfach als Faktor zur Laufzeit von $splay(x)$ verrechnet werden. Alles andere, was neben Suche und der splay-Operation passiert, wie zum Beispiel das Löschen eines Knotens oder das Umhängen von Kind und Vaterpointern ist in $\mathcal{O}(1)$ realisierbar. Da die splay-Operation also alles andere dominiert, zeigen wir nun, dass die

Laufzeit von *splay*() in amortisiert $\mathcal{O}(log(n))$ liegt. Daraus werden wir im nächsten Abschnitt für alle anderen genannten Funktionen ebenfalls eine Laufzeit in amortisiert $\mathcal{O}(log(n))$ folgern.

2.4. Die Laufzeitanalyse von splay()

Wir haben nun also gesehen, dass wir alle Funktionen in amortisiert $\mathcal{O}(log(n))$ realisieren können, insofern die splay-Operation in amortisiert $\mathcal{O}(log(n))$ läuft. Wir wollen kurz zur Erinnerung betrachten was amortisiert in $\mathcal{O}(log(n))$ heißt. Eine einzelne splay-Operation kann nämlich im Worst Case in $\mathcal{O}(n)$ liegen. Das passiert zum Beispiel, wenn unser Splay tree die Form eines Pfades mit n Elementen annimmt. Jedoch ist die Worst Case Laufzeit von m aufeinanderfolgenden splay-Operationen in $\mathcal{O}(m \cdot log(n)) \subseteq \mathcal{O}(m \cdot n)$ für geeignet großes m. Intuitiv gesehen müssen nämlich zunächst viele billige Operationen ausgeführt werden, bevor man eine sehr teure splay-Operation durchführen kann. Dadurch kann man genug Laufzeit ansparen, um die teure splay-Operation zu bezahlen. Genau diese Idee benutzt auch die sogenannte Potentialfunktionmethode, mit der wir die amortisierte Laufzeitanalyse durchführen. Wir weisen unserem Splay tree ein Potential zu, das abhängig von der Struktur des Baumes ist. Dabei wollen wir ein großes Potential erreichen, wenn die Baumstruktur ungünstig für eine weitere splay-Operation ist und ein kleines Potential, wenn die Baumstruktur günstig ist. Um mit der Potentialfunktionmethode nun also die amortisierte Laufzeit von $\mathcal{O}(log(n))$ zu zeigen, müssen wir zunächst einige Parameter definieren. Sei t ein Baum mit n Knoten und x einer dieser Knoten:

Größe des Knotens x: $size(x)$ liefert die Anzahl der Elementen im Teilbaum T_x von t. Dabei ist T_x der Teilbaum, der von dem Knoten x als Wurzel aufgespannt wird. Wir beschreiben die Größe eines Knotens x also durch $size(x) = \sum_{y \in T_x} 1$.

Rang des Knotens x: $rank(x) = log_2(size(x))$.

Potential Φ des Baumes t: $\Phi(t) = \sum_{x \in t} rank(x)$. Das Potential eines Baumes t ist also die Summe über die Ränge aller Knoten aus t.

Am. Laufzeit von splay(x): $T_{\text{am}}(splay(x)) = T_{\text{tat}}(splay(x)) + \Phi_{\text{neu}}(t) - \Phi_{\text{alt}}(t)$.
Hierbei beschreibt T_{am} die amortisierte, T_{tat} die tatsächliche Laufzeit, $\Phi_{\text{neu}}(t)$ das Potential des Baumes t nach der splay-Operation und $\Phi_{\text{alt}}(t)$ vor der splay-Operation.

Diese Potentialfunktion Φ sieht auf den ersten Blick vielleicht etwas willkürlich gewählt aus. Bevor wir also anfangen zu beweisen, dass die amortisierte Laufzeit von $splay(x)$ tatsächlich in $\mathcal{O}(log(n))$ liegt, wollen wir zunächst intuitiv verstehen, warum uns die Potentialfunktion Φ genau das liefert, was wir brauchen. Wie zuvor schon erwähnt, brauchen wir ein großes Potential, wenn wir uns in der Situation befinden, dass unsere splay-Operation plötzlich sehr teuer ist. Dann können wir nämlich das Potential auflösen, um die teure Operation zu bezahlen. Ist unsere splay-Operation jedoch günstig, so brauchen wir kein großes Potential. Wir betrachten zunächst, wie die Struktur unseres Baumes aussieht, wenn die splay-Operation teuer ist. Dann muss unser Baum unbalanciert sein und lange Pfade beinhalten. Lange Pfade entstehen durch dicke Knoten. Als dicke Knoten bezeichnen wir hierbei Knoten, die sehr viele Nachkommen des Vaters als eigene Nachkommen haben. Diese dicken Knoten liefern für unseren Baum jedoch ein großes Potential. Wir werden nun das Potential für den ungünstigsten Fall „Pfad“ und für den günstigsten Fall „perfekt balancierter Baum“ beispielhaft berechnen.

Lemma 2.1 (Maximales und minimales Potential)**.** *Das maximale Potential $\Phi(t)$ eines Splay trees t mit n Knoten ist in $\mathcal{O}(n \cdot log(n))$ und das minimale Potential $\Phi(t)$ ist in $\Omega(n)$.*

Beweis. Wir betrachten zunächst den Fall des maximalen Potentials. Dieser Fall tritt ein, wenn der Baum t ein Pfad ist. Dann hat jeder Knoten mit Ausnahme des Blattknotens genau ein Kind. Wir berechnen das Potential $\Phi(t)$:

$$\begin{aligned}
\Phi(t) = \sum_{x \in t} rank(x) &= \sum_{x \in t} log_2(size(x)) \\
&= log_2(size(root)) + log_2(size(root.child)) + ... + log_2(size(leaf)) \\
&= log_2(n) + log_2(n-1) + ... + log_2(1) \\
&= \sum_{i=1}^{n} log_2(i) \\
&\approx \int_1^n log_2(i)\ di \ = \ log_2(e) \cdot \int_1^n ln(i)\ di \\
&= log_2(e) \cdot [i \cdot ln(i) - i]_1^n \\
&= log_2(e) \cdot ((n \cdot ln(n) - n) + (1)) \\
&= \mathcal{O}(n \cdot log(n))\ .
\end{aligned}$$

Nun berechnen wir noch das Potential $\Phi(t)$ für einen perfekt balancierten Baum t. Das heißt mit Ausnahme der Blattknoten sollen alle Knoten genau zwei Kinder haben. Seine Knotenanzahl soll auch n betragen, wobei $n = 1 + 2 + 4 + ... = \sum_{i=0}^{m-1} 2^i = 2^m - 1$. Hierbei entspricht m der Tiefe von t. Dann gilt für das Potential $\Phi(t)$:

$$\begin{aligned}
\Phi(t) = \sum_{x \in t} rank(x) &= \sum_{x \in t} log_2(size(x)) \\
&= log_2(size(root)) + log_2(size(root.right)) + log_2(size(root.left) + ... \\
&= log_2(n) + 2 \cdot log_2(2^{m-2} - 1) + 4 \cdot log_2(2^{m-3} - 1) + ... + n \cdot log_2(1) \\
&= \sum_{i=0}^{m-1} 2^i \cdot log_2(2^{m-i} - 1) \\
&\approx \sum_{i=0}^{m} 2^i \cdot log_2(2^{m-i}) \ = \ log_2(2) \cdot \sum_{i=0}^{m} 2^i \cdot (m-i) \\
&\overset{j=m-i}{=} \sum_{j=0}^{m} j \cdot 2^{m-j} \ = \ 2^m \cdot \sum_{j=0}^{\infty} j \cdot \frac{1}{2^j} \\
&= 2^m \cdot \frac{1}{2} \cdot \sum_{j=0}^{\infty} j \cdot \frac{1}{2^{j-1}} \\
&\overset{(*)}{=} 2^m \cdot \frac{1}{2} \cdot 4 \\
&= \Omega(n)\ .
\end{aligned}$$

$$(*) : \sum_{i=0}^{\infty} i \cdot x^{i-1} = \frac{1}{(1-x)^2} \quad \text{für } |x| < 1$$

□

Wir zeigen nun unter zu Hilfenahme des sogenannten Access Lemma, dass die amortisierte Laufzeit einer splay-Operation tatsächlich in $\mathcal{O}(log(n))$ liegt.

Lemma 2.2 (Access Lemma)**.** *Die amortisierte Laufzeit für den Aufruf von splay(x) auf einem Baum t mit Wurzel r beträgt maximal*

$$\mathcal{O}\left(1 + 3 \cdot (rank(r) - rank(x))\right) = O\left(1 + 3 \cdot log\left(\frac{size(r)}{size(x)}\right)\right).$$

Wir benutzen das Access Lemma zunächst ohne es bewiesen zu haben, jedoch folgt später ein Beweis in verallgemeinerter Form. Betrachten wir nun einen Baum t mit n Knoten. Dann befinden sich alle n Knoten im Teilbaum, der von der Wurzel r aufgespannt wird und somit ist $size(r) = n$. Über x können wir nicht viel aussagen. Wir wissen jedoch, dass sich in dem Teilbaum, der von x aufgespannt wird, mindestens ein Element befindet. Und das ist x selbst. Es gilt also $size(x) \geq 1 \ \ \forall x \in t$ und somit:

$$\begin{aligned} rank(r) &= log_2(size(r)) = log_2(n) \\ rank(x) &= log_2(size(x)) \geq log_2(1) = 0. \end{aligned}$$

Insgesamt folgt also durch das Access Lemma, dass für die amortisierte Laufzeit von $splay(x)$ gilt:

$$\begin{aligned} T_{\text{am}}(splay(x)) &= \mathcal{O}\left(1 + 3 \cdot (rank(r) - rank(x))\right) \\ &= \mathcal{O}\left(1 + 3 \cdot (log_2(n) - 0)\right) \\ &= \mathcal{O}\left((log(n))\right) \end{aligned}$$

Korollar 2.3. *Die amortisierte Laufzeit von splay(x) ist in O(log(n)).*

Nun können wir mit der Erkenntnis, dass eine splay-Operation in amortisiert $\mathcal{O}(log(n))$ abläuft, auch Aussagen über die tatsächliche Laufzeit der splay-Operation treffen. Wir betrachten nun eine Sequenz von m aufeinanderfolgenden splay-Aufrufen. Denn wie schon zuvor erläutert bietet eine einzelne splay-Operation nur eine schlechte Worst case Laufzeit von $\mathcal{O}(n)$. Betrachten wir jedoch viele splay-Aufrufe nacheinander, so führt dies zu einer besseren oberen Schranke.

Satz 2.4. *Die tatsächliche Laufzeit für m splay-Aufrufe auf einem Baum t mit n Knoten beträgt maximal*

$$\mathcal{O}((n + m) \cdot log(n)).$$

Beweis. Zunächst benötigen wir eine Formel für die tatsächliche Laufzeit einer einzelnen splay-Operation. Sei T_{am} die amortisierte Laufzeit und T_{tat} die tatsächliche. Dann gilt:

$$\begin{aligned} & T_{\text{am}}(splay(x)) = T_{\text{tat}}(splay(x)) + \Phi'(t) - \Phi(t) \\ \iff \quad & T_{\text{tat}}(splay(x)) = T_{\text{am}}(splay(x)) + \Phi(t) - \Phi'(t) \end{aligned}$$

Wobei $\Phi(t)$ das Potential vor der splay-Operation sein soll und $\Phi'(t)$ danach. Wir wollen nun aber eine Formel für die tatsächliche Laufzeit von m aufeinanderfolgenden splay-Operationen finden. Dazu müssen wir nur die tatsächlichen Laufzeiten T_{tat} aller m splay-Operationen aufaddieren. Wir erhalten:

$$T_{\text{tat}}(m \text{ splay-Ops.}) = T_{\text{am}}(m \text{ splay-Ops.}) + \Phi_{\text{alt}}(t) - \Phi_{\text{neu}}(t)$$

Hierbei soll nun $\Phi_{\text{alt}}(t)$ das Potential vor den m splay-Operationen sein und $\Phi_{\text{neu}}(t)$ nach den m splay-Operationen. Alle Potentiale die zwischenzeitlich auftreten, fallen günstigerweise in einer Teleskopsumme weg. Denn wenn der i-te splay-Aufruf mit dem Potential $\Phi(t)^{(i)}$ aufhört, so fängt der (i+1)-te splay-Aufruf mit genau diesem Potential an. Nun können wir einiges über die Parameter in der obigen Gleichung aussagen. Zum einen wissen wir nach Korollar 2.3, dass jede der m splay-Operationen eine amortisierte Laufzeit von $\mathcal{O}(log(n))$ hat. Zum anderen kann das Potential $\Phi(t)$ nach Lemma 2.1 maximal $\mathcal{O}(n \cdot log(n))$ und mininmal $\Omega(n)$ sein. Es folgt:

$$\begin{aligned} T_{\text{tat}}(m \text{ splay-Ops.}) &= T_{\text{am}}(m \text{ splay-Ops.}) + \Phi_{\text{alt}}(t) - \Phi_{\text{neu}}(t) \\ &= m \cdot \mathcal{O}(log(n)) + \mathcal{O}(n \cdot log(n)) - \Omega(n) \\ &\leq m \cdot \mathcal{O}(log(n)) + \mathcal{O}(n \cdot log(n)) \\ &= \mathcal{O}((m+n) \cdot log(n)) \end{aligned}$$

Und damit folgt die Behauptung. □

Natürlich wollen wir noch eine Aussage über die Lauzeit aller anderen Funktionen der Splay trees treffen. Dabei wird in den Funktionen search, insert, delete, join und split mindestens einmal die splay-Operation aufgerufen. Wir wollen zeigen, dass die Laufzeiten aller Funktionen in amortisiert $\mathcal{O}(log(n))$ liegen. Asymptotisch gesehen haben die Funktionen dieselbe amortisierte Laufzeit wie die splay-Operation.

Lemma 2.5. *Die amortisierten Laufzeiten der Funktionen search(), insert(), delete(), join() und split() liegen jweils in $\mathcal{O}(log(n))$.*

Beweis. Zunächst beschreiben wir unser genaues Vorgehen. Wie immer ist die amortisierte Laufzeit einer Funktion $function$, die auf einem Baum t operiert, defininiert als:

$$T_{\text{am}}(function) = T_{\text{tat}}(function) + \Phi_{\text{neu}}(t) - \Phi_{\text{alt}}(t) \tag{2.1}$$

Sagen wir nun, die Funktion $function$ besteht aus k Operationen OP_1, OP_2, ..., OP_{k}, die der Reihe nach ausgeführt werden. Dann berechnen wir die amortisierten Laufzeiten der einzelnen Operationen, um die Laufzeit der gesamten Funktion $function$ zu erhalten:

$$\begin{aligned} T_{\text{am}}(function) = &T_{\text{tat}}(OP_1) + \Phi'(t) - \Phi(t) \\ &+ T_{\text{tat}}(OP_2) + \Phi''(t) - \Phi'(t) \\ &+ ... \\ &+ T_{\text{tat}}(OP_{\text{k}}) + \Phi^{(k)}(t) - \Phi^{(k-1)}(t) \end{aligned}$$

Dabei sei $\Phi'(t)$ das Potential von t nach der ersten Operation und somit vor der zweiten Operation, $\Phi''(t)$ das Potential nach der zweiten Operation und so weiter. Mit $\Phi^{(k)}(t)$ bezeichnen wir das Potential nach der k-ten Operation OP_{k}, welches somit auch das Potential $\Phi_{\text{neu}}(t)$ beschreibt. Durch eine Teleskopsumme heben sich alle Potentiale bis auf $\Phi^{(k)}(t) = \Phi_{\text{neu}}(t)$ und $\Phi(t) = \Phi_{\text{alt}}(t)$ weg und wir erhalten die Gleichung (2.1). Das heißt, es reicht, die amortisierten Laufzeiten der einzelnen Operationen einer Funktion zu betrachten.
Wir wollen jede Funktion hinsichtlich der amortisierten Laufzeiten ihrer Operationen betrachten. Dabei wissen wir durch das Korollar 2.3 bereits, dass die amortisierte Laufzeit einer splay-Operation auf einem Baum t in $\mathcal{O}(log(n))$ liegt. Dabei sei wie immer n die Anzahl der Elemente aus t. Auch die amortisierte Laufzeit einer Suche nach einem Element x, welches dann durch

$splay(x)$ zur Wurzel befördert wird, liegt in $\mathcal{O}(log(n))$. Das gilt, da bei der Suche derselbe Weg zurückgelegt wird wie bei der darauf folgenden splay-Operation. Es sei außerdem erwähnt, dass das Gesamtpotential mehrerer Bäume die Summe der Potentiale der einzelnen Bäume ist. Bevor wir nun jede Funktion der Splay trees einzeln hinsichtlich ihrer amortisierten Laufzeit analysieren, sollten wir nochmal unsere Potentialfunktion betrachten. In den Funktionen delete, join und split sind zumindest zwischenzeitlich mehrere Splay trees vorhanden. In den Situationen, in denen wir mehrere Bäume gleichzeitig betrachten, suchen wir das Gesamtpotential der Situation. Dieses Gesamtpotential definieren wir als die Summe der Potentiale der einzelnen Bäume. Das stellt keinen Widerspruch zu unserer vorherigen Definition der Potentialfunktion dar, da wir immer noch die Summe der Ränge aller Elemente berechnen. Nur befinden sich diese Elemente eventuell nicht im selben Baum. Kommen wir nun endlich zur Laufzeitanalyse der Funktionen:

search(t,x): Die einzelnen Operationen sind $BST\text{-}search(x)$ und $splay(x)$. Es folgt direkt:

$$\begin{aligned} T_{\text{am}}(search(t,x)) &= T_{\text{am}}(BST\text{-}search(x)) + T_{\text{am}}(splay(x)) \\ &= \mathcal{O}(log(n)) \end{aligned}$$

insert(t,x): Die erste Operation ist $searchPlace(x)$. Wir suchen den Ort im Baum, an dem wir x einfügen können. Wir starten unsere Suche bei der Wurzel $root$ und suchen im linken Teilbaum, falls $root.key > x.key$ und im rechten, falls $root.key < x.key$. Wir beenden unsere Suche an dem Knoten w, der kein rechtes beziehungsweise linkes Kind hat. Dort können wir also x als neuen Blattknoten einfügen. Zuletzt wird noch die Operation $splay(x)$ aufgerufen. Es folgt:

$$\begin{aligned} T_{\text{am}}(insert(t,x)) &= T_{\text{am}}(searchPlace(x)) + T_{\text{am}}(w.Child = x) + T_{\text{am}}(splay(x)) \\ &= \mathcal{O}(log(n)) + T_{\text{am}}(w.Child = x) + \mathcal{O}(log(n)) \end{aligned}$$

Nun müssen wir die Operation, die x zum Kind von w macht noch etwas genauer betrachten. Die tatsächliche Laufzeit liegt offensichtlich in $\mathcal{O}(1)$, da wir nur einen Vater- und einen Kindzeiger einfügen müssen. Doch durch das Einfügen eines neuen Kindknotens x ändern sich die Ränge aller Vorfahren von x und somit auch das Potential von t.

$$\begin{aligned} T_{\text{am}}(w.Child = x) &= T_{\text{tat}}(w.Child = x) + \Phi'(t) - \Phi(t) \\ &= \mathcal{O}(1) + \Phi'(t) - \Phi(t) \end{aligned}$$

Wie immer sei $\Phi'(t)$ das Potential nach dem Einfügen von x und $\Phi(t)$ davor. Wir wollen nun also betrachten, wie groß die Potentialänderung höchstens sein kann, die durch das Einfügen eines Knotens ausgelöst wird. Im schlechtesten Fall ändern sich die Ränge aller Knoten des Baumes. Das passiert, wenn jeder Knoten im Baum genau ein Kind hat und x als Kind des einzigen Blattknoten eingefügt wird. Dann sind alle Knoten des Baumes Vorfahren von x. Für den Fall, dass es sich bei dem Baum um einen Pfad handelt, haben wir das Potential bereits in Lemma 2.1 berechnet. Falls t ein Pfad ist, der aus n Elementen besteht, gilt $\Phi(t) = \sum_{x \in t} rank(x) = \sum_{i=1}^{n} log_2(i)$. Fügen wir nun einen weiteren Knoten

dem Pfad hinzu, so vergrößert sich die Größe jedes Knotens um genau eins und es folgt:

$$\begin{aligned}\Phi'(t) = \sum_{x \in t} rank(x) &= \sum_{i=1}^{n} log_2(i+1) + log(1) \\ &= \sum_{i=1}^{n+1} log_2(i) \\ &= log_2(n+1) + \sum_{i=1}^{n} log_2(i) \\ &= log_2(n+1) + \Phi(t)\end{aligned}$$

Da wir den schlechtesten Fall betrachtet haben, können wir die größtmögliche Potentialänderung durch das Einfügen eines Knotens auf einem beliebigen Splay tree t durch folgende Rechnung abschätzen:

$$\begin{aligned}\Phi'(t) - \Phi(t) &\leq log_2(n+1) + \Phi(t) - \Phi(t) \\ &= log_2(n+1) \\ &= \mathcal{O}(log(n))\end{aligned}$$

Und nun folgt, da alle Teiloperationen der insert-Funktion amortisiert in $O(log(n))$ liegen, dass auch die insert-Funktion amortisiert in $\mathcal{O}(log(n))$ liegt.

delete(t,x): Betrachten wir dazu die Abbildung 2.4. Die Operationen der delete-Funktion sind: $splay(x)$, $remove(x)$ und $splay(w)$, wobei w der größte Knoten des linken Teilbaumes t_L ist. Die Teilbäume t_L und t_R entstehen durch das Löschen von x. Zuletzt setzen wir noch $w.rightChild = t_\mathrm{R}.root$. Die amortisierte Laufzeit der splay-Operationen ist wie wir aus Korollar 2.3 wissen in $\mathcal{O}(log(n))$. Auch das Löschen des Knotens x ist unproblematisch, da die tatsächliche Laufzeit in $\mathcal{O}(1)$ liegt. Da wir zuvor x durch den splay-Aufruf zur Wurzel des Baumes befördert haben, verändern sich die Ränge aller Knoten nicht, da keiner der anderen Knoten x als Nachkommen hatte. Das Potential des Baumes t verringert sich jedoch um $log(n)$, da die Wurzel gelöscht wurde. Die Wurzel hatte genau n Nachkommen. Da nach dem Löschen von x der Baum t in die Teilbäume t_L und t_R zerfällt, berechnen wir das Gesamtpotential $\Phi'(t)$ für diese Situation durch die Summe der Potentiale der einzelnen Bäume. Für die amortisierte Laufzeit der remove-Operation gilt nun:

$$\begin{aligned}T_\mathrm{am}(remove(x)) &= T_\mathrm{tat}(remove(x)) + \Phi'(t) - \Phi(t) \\ &= \mathcal{O}(1) + (\Phi(t_\mathrm{L}) + \Phi(t_\mathrm{R})) - (\Phi(t_\mathrm{L}) + \Phi(t_\mathrm{R}) + log(n)) \\ &= \mathcal{O}(1)\end{aligned}$$

Betrachten wir zuletzt noch die Operation $w.rightChild = t_\mathrm{R}.root$. Die tatsächliche Laufzeit ist in $\mathcal{O}(1)$. Das Gesamtpotential vor der Operation beträgt $\Phi(t) = \Phi({t_\mathrm{L}}') + \Phi(t_\mathrm{R}) + rank(w)$. Nach der Operation beträgt das Potential $\Phi'(t) = \Phi({t_\mathrm{L}}') + \Phi(t_\mathrm{R}) + rank'(w)$. Dann gilt für die amortisierte Laufzeit dieser Operation:

$$\begin{aligned}&T_\mathrm{am}(w.rightChild = t_\mathrm{R}.root) \\ &= T_\mathrm{tat}(w.rightChild = t_\mathrm{R}.root) + \Phi'(t) - \Phi(t) \\ &= \mathcal{O}(1) + (\Phi({t_\mathrm{L}}') + \Phi(t_\mathrm{R}) + rank'(w)) - (\Phi({t_\mathrm{L}}') + \Phi(t_\mathrm{R}) + rank(w)) \\ &= \mathcal{O}(1) + rank'(w) - rank(w) \\ &= \mathcal{O}(log(n))\end{aligned}$$

Der letzte Schritt der Rechnung folgt, da w die Wurzel von t ist und somit alle n Elemente als Nachkommen hat. Nun hat jede Operation der delete-Funktion eine amortisierte Laufzeit in $\mathcal{O}(log(n))$ und somit liegt die Laufzeit der delete-Funktion ebenfalls in $\mathcal{O}(log(n))$.

join(t_1, t_2): Wir betrachten für die Laufzeitanalyse dieser Funktion die Abbildung 2.5. Zunächst wird durch die Operation $splay(w)$ der Knoten w zur Wurzel des Baumes t_1 befördert. Dabei ist w der Knoten mit dem größten Schlüssel aus t_1. Die amortisierte Laufzeit der splay-Operation ist in $\mathcal{O}(log(n))$. Betrachten wir also noch die zweite Operation $w.rightChild = t_2.root$, wodurch die zwei Bäume t_1 und t_2 zusammenwachsen. Da diese Operation analog zur letzten Operation der delete-Funktion verläuft, können wir direkt folgern, dass die amortisierte Laufzeit der join-Funktion ebenfalls in $\mathcal{O}(log(n))$ liegt.

split(t,x): Für die Analyse dieser Funktion betrachten wir die Abbildung 2.6. Zunächst befördern wir den Knoten x durch einen splay-Aufruf zur Wurzel. Dabei wissen wir, dass die splay-Operation eine amortisierte Laufzeit in $\mathcal{O}(log(n))$ hat. Nun trennen wir den Knoten x von seinem rechten Kind durch die Operation $x.rightChild = none$ und haben somit den Baum in zwei Bäume t_L und t_R aufgeteilt. Das Trennen ist in $\mathcal{O}(1)$ realisierbar. Hierbei beschreibt $\Phi(t) = \Phi(t_\mathrm{L}) + \Phi(t_\mathrm{R}) + rank(x)$ das Potential vor der Trennung und $\Phi'(t) = \Phi(t_\mathrm{L}) + \Phi(t_\mathrm{R}) + rank'(x)$ nach der Trennung. Es ändert sich nur der Rang des Wurzelknotens x, denn nur seine Nachkommen verändern sich. Es folgt für die amortisierte Laufzeit der Trennoperation:

$$
\begin{aligned}
&T_\mathrm{am}(Trenn\text{-}Operation)\\
&= T_\mathrm{tat}(Trenn\text{-}Operation) + \Phi'(t) - \Phi(t)\\
&= \mathcal{O}(1) + (\Phi(t_\mathrm{L}) + \Phi(t_\mathrm{R}) + rank'(x)) - (\Phi(t_\mathrm{L}) + \Phi(t_\mathrm{R}) + rank(x))\\
&= \mathcal{O}(1) + rank'(x) - rank(x)\\
&= \mathcal{O}(1)
\end{aligned}
$$

Die letzte Gleichheit folgt, da der Knoten x Nachkommen verliert und sich somit der Rang von x durch die Trennung von seinem rechten Kind nur verringern kann. Deshalb gilt $rank(x) \geq rank'(x)$. Die Laufzeit der zuvor ausgeführten splay-Operation liegt jedoch in amortisiert $\mathcal{O}(log(n))$ und somit liegt die Laufzeit der split-Funktion ebenfalls in amortisiert $\mathcal{O}(log(n))$.

Wir haben für alle Funktionen eine amortisierte Laufzeit in $\mathcal{O}(log(n))$ gezeigt und somit folgt die Behauptung. □

Natürlich bleibt jetzt noch das Access Lemma zu zeigen, welches wir bei unseren bisherigen Laufzeitbetrachtungen der splay-Operation vorausgesetzt haben. Wir wollen das Access Lemma zunächst noch um einige Parameter erweitern. Durch diese Ergänzungen können wir später im Abschnitt 2.5 nämlich die sogenannte statische Optimalität von Splay-trees zeigen. Wir führen für jeden Knoten x im Splay tree t ein nicht-negatives Gewicht $w(x)$ ein und definieren $size(x)$ als Summe der Gewichte aller Knoten aus T_x. Hierbei ist T_x der Teilbaum, der durch den Knoten x aufgespannt wird.

Gewicht des Knotens x: $w(x)$ ist das nicht-negative Gewicht des Knoten x. Es ist nicht zu verwechseln mit dem Schlüssel, den jeder Knoten besitzt! Die Gewichte können bei der Laufzeitanalyse verschieden interpretiert werden und liefern somit verschiedene Ergebnisse.

Größe des Knotens x: $size(x)$ liefert die Summe der Gewichte aller Nachkommen des Knoten x, x mit eingeschlossen. Das kann man beschreiben durch $size(x) = \sum_{y \in T_\mathrm{x}} w(y)$.

Es sei erwähnt, dass wir die Gewichte jedes Knotens bei unseren vorherigen Betrachtungen gleich 1 gesetzt haben. Dadurch war die Größe eines Knotens durch die Anzahl der Nachkommen (inklusive des Knotens selbst) definiert.
Die Parameter Rang eines Knotens x ($rank(x)$), Potential $\Phi(t)$ und die amortisierte Laufzeit bleiben gleich definiert. Ein Splay tree t hat nun also ein großes Potential, wenn Knoten mit hohem Gewicht sehr tief im Baum stehen. Dadurch zählt das Gewicht der schweren Knoten nämlich zur Größe und somit zum Rang all ihrer Vorfahren dazu. Da das Potential über den Rang aller Knoten definiert ist, erhalten wir ein großes Potential. Der Baum t hat also ein kleines Potential, wenn schwere Knoten nahe bei der Wurzel stehen.
Bevor wir nun mit unseren neuen Parametern unser Access Lemma verallgemeinern und dann endlich auch beweisen, zeigen wir noch einen kleinen Hilfssatz. Dieser wird uns bei dem Beweis des Access Lemma behilflich sein.

Hilfssatz 2.6. *Gegeben seien zwei Zahlen x,y* $\in \mathbb{R}$. *Dann gilt:*

$$x + y \leq 1 \quad \Longrightarrow \quad log_2(x) + log_2(y) \leq -2.$$

Beweis. Sei x + y $\leq$ 1. Dann wollen wir daraus folgern, dass auch $log_2(x) + log_2(y) \leq -2$ gilt.

$$\begin{array}{lrcll}
 & 4 \cdot x \cdot y & = & (x+y)^2 - (x-y)^2 & \text{gilt offensichtlicherweise} \\
\Rightarrow & 4 \cdot x \cdot y & \leq & 1 - (x-y)^2 & \text{da } x + y \leq 1 \\
\Rightarrow & 4 \cdot x \cdot y & \leq & 1 & \text{da } (x-y)^2 \text{ nicht negativ} \\
\Rightarrow & log_2(4 \cdot x \cdot y) & \leq & log_2(1) & \\
\Rightarrow & log_2(4) + log_2(x) + log_2(y) & \leq & 0 & \\
\Rightarrow & log_2(x) + log_2(y) & \leq & -2 &
\end{array}$$

□

Lemma 2.7 (Das verallgemeinerte Access Lemma)**.** *Die amortisierte Laufzeit für den Aufruf von splay(x) auf einem Baum t mit Wurzel r beträgt maximal*

$$\mathcal{O}\left(1 + 3 \cdot (rank(r) - rank(x))\right) = O\left(1 + 3 \cdot log\left(\frac{size(r)}{size(x)}\right)\right)$$

für jede beliebige Belegung der Gewichte w(x).

Beweis. Die Idee in diesem Beweis ist es, die splay-Operation in ihre Komponenten aufzuteilen. Das heißt, wir analysieren alle Rotationen, die während der splay-Operation durchgeführt werden hinsichtlich ihrer Laufzeit. Im Folgenden werden wir den Rang eines Knotens x vor der Rotation als $rank(x)$ bezeichnen und nach der Rotation als $rank'(x)$. Bei der Laufzeitanalyse der Rotationen sehen wir, dass die Kosten einer Einzelrotation in $\mathcal{O}(1+3\cdot(rank'(x)-rank(x)))$ liegen und die Kosten einer Doppelrotation in $\mathcal{O}(3 \cdot (rank'(x) - rank(x)))$. Bevor wir diese Aussagen beweisen, betrachten wir zunächst, was wir daraus folgern können. Wenn wir die Kosten aller Rotationen aufaddieren, erhalten wir die amortisierten Kosten der splay-Operation:

$$\begin{aligned}
T_{\text{am}}(splay(x)) &= 3 \cdot (rank'(x) - rank(x)) \\
&\quad + 3 \cdot (rank''(x) - rank'(x)) \\
&\quad + ... \\
&\quad + 3 \cdot (rank^{final\prime}(x) - rank^{almostfinal\prime}(x)) + 1 \\
&\overset{Teleskop.}{=} 3 \cdot (rank^{final\prime}(x) - rank(x)) + 1 \\
&= 3 \cdot (rank(r) - rank(x)) + 1
\end{aligned}$$

Durch eine Teleskopsumme fallen fast alle Terme weg und wir erhalten $T_{\text{am}}(splay(x) = 3 \cdot (rank^{finalr}(x) - rank(x)) + 1$. Dabei gilt jedoch, dass nach der finalen Rotation der Knoten x in der Wurzel des Splay trees steht und somit denselben Rang hat, wie die Wurzel r vor den Rotationen. Die amortisierte Laufzeit der splay-Operation liegt also folglich in amortisiert $\mathcal{O}(1+3 \cdot (rank(r) - rank(x)))$. Und das war zu zeigen. Es sollte noch bemerkt werden, dass die Addition der Konstante 1 im letzten Schritt auf eine Einzelrotation zurückzuführen ist. Diese Einzelrotation wird nur ausgeführt, wenn der Knoten x im Baum eine ungerade Höhe hatte. Jetzt bleibt nur noch die Laufzeit der einzelnen Rotationen zu zeigen, damit die Behauptung folgt. Wir benutzen wieder unsere Formel für die amortisierte Laufzeit:

$$T_{\text{am}} = T_{\text{tat}} + \Phi_{\text{neu}}(t) - \Phi_{\text{alt}}(t).$$

In den folgenden Betrachtungen wird der Übersichtlichkeit halber der Rang eines Knotens x als $r(x)$ und die Größe eines Knotens als $s(x)$ abgekürzt. Wie gewohnt sollen dann $r'(x)$ und $s'(x)$ den Rang und die Größe von x nach einer Rotation bezeichnen. Die folgende Fallanalyse orientiert sich an der Arbeit von J. Erickson [3].

1.Fall - Zag case

Da es sich hierbei um eine Einzelrotation handelt, wollen wir eine amortisierte Laufzeit von $\mathcal{O}(1 + 3 \cdot (rank'(x) - rank(x)))$ zeigen. Wir berechnen also

$$T_{\text{am}}(Zag) = T_{\text{tat}}(Zag) + \Phi'(t) - \Phi(t)$$

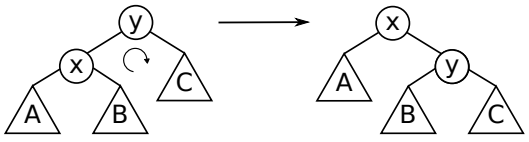
Wie immer soll hierbei $\Phi'(t)$ das Potential des Baumes nach der Zag-Rotation sein und $\Phi(t)$ das Potential davor. Die tatsächlichen Kosten von Zag sind in O(1), da nur einmal rotiert wird. Die Differenz des Potential abzuschätzen ist etwas komplizierter. Wie man in der Abbildung 2.1 sieht, verändert sich bei einer Zag-Rotation aber nur der Rang von x und seinem Vater y, da sie andere Nachkommen erhalten. Alle Vorfahren von x und y behalten ihre Nachkommen, auch wenn sich deren Platz im Baum ändert. Dadurch ist die Änderung im Potential also nur von der Rangänderung der Knoten x und y abhängig. Wir schätzen zunächst die Laufzeit der Zag-Rotation ab. Die einzelnen Schritte der Rechnung werden danach näher erläutert.

$$\begin{aligned}
1 + \Phi'(t) - \Phi(t) &= 1 + (r'(x) + r'(y)) - (r(x) + r(y)) \\
&= 1 + r'(x) - r(x) + r'(y) - r(y) \\
&\leq 1 + r'(x) - r(x) && [r'(y) \leq r(y)] \\
&\leq 1 + 3 \cdot r'(x) - 3 \cdot r(x) && [r'(x) \geq r(x)]
\end{aligned}$$

Die zwei benutzen Ungleichungen lassen sich leicht anhand folgender Abbildung erklären (vgl. Abbildung 2.1).

$r'(y) \leq r(y)$, da der Knoten y durch die Rotation den Knoten x und den Teilbaum A als Nachkommen verliert.

$r'(x) \geq r(x)$, da der Knoten x durch die Rotation y und den Teilbaum C als Nachkommen gewinnt.



2.Fall - Zag-Zag case

Die Zag-Zag Rotation ist eine Doppelrotation, weshalb wir nun eine amortisierte Laufzeit von $\mathcal{O}(3 \cdot (rank'(x) - rank(x)))$ zeigen wollen. Auch in diesem Fall hängt die Potentialänderung

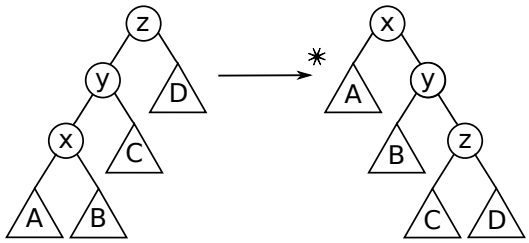
nur von der Rangänderung einiger Knoten ab. Wir brauchen deshalb nur die Knoten x, $y = parent(x)$ und $z = parent(y)$ hinsichtlich ihres Ranges zu betrachten, denn nur diese Knoten haben nach der Rotation nicht mehr dieselben Nachkommen wie zuvor. Für die tatsächlichen Kosten von Zag-Zag veranschlagen wir diesmal 2 Einheiten, da wir zwei mal rotieren. Die einzelnen Schritte der folgenden Abschätzung werden danach erklärt.

$$
\begin{aligned}
& 2 + \Phi'(t) - \Phi(t) \\
= \;& 2 + (r'(x) + r'(y) + r'(z)) - (r(x) + r(y) + r(z)) \\
= \;& 2 + r'(y) - r(z) + r'(x) + r'(z) - r(y) - r(x) \\
\leq \;& 2 + r'(x) + r'(z) - r(x) - r(y) && [r'(y) \leq r(z)] \\
\leq \;& 2 + r'(x) + r'(z) - 2 \cdot r(x) && [r(x) \leq r(y)] \\
= \;& 2 + r'(x) + r'(z) - 2 \cdot r(x) + (2 \cdot r'(x) - 2 \cdot r'(x)) + (r(x) - r(x)) && [+0] \\
= \;& 2 + (r'(z) - r'(x)) + (r(x) - r'(x)) + 3 \cdot (r'(x) - r(x)) \\
= \;& 2 + log_2(\tfrac{s'(z)}{s'(x)}) + log_2(\tfrac{s(x)}{s'(x)}) + 3 \cdot (r'(x) - r(x)) && [Def.\ Rang] \\
\leq \;& 2 + (-2) + 3 \cdot (r'(x) - r(x)) && [Satz\ 2.6] \\
= \;& 3 \cdot (r'(x) - r(x))
\end{aligned}
$$

Die in der Rechnung getroffenen Abschätzungen sowie die Voraussetzungen des Hilfssatzes 2.6 lassen sich anhand folgender Abbildung erklären (vgl. Abbildung 2.2). Wir begründen im Folgenden auch die richtige Verwendung des Hilfssatzes 2.6.

$r'(y) \leq r(z)$, da der Knoten z vor der Rotation dieselben Nachkommen hat wie der Knoten y nach der Rotation, jedoch zusätzlich noch Knoten x mit Teilbaum A.

$r(x) \leq r(y)$, da der Knoten y zusätzlich zum Knoten x und seinen Nachkommen noch den Teilbaum C als Nachkommen hat.



$s'(z) + s(x) \leq s'(x)$, da die Größe von x vor der Rotation aus den Gewichten von x, A und B bestehen und die Größe von z nach der Rotation aus den Gewichten von z, C und D. Wir addieren diese Gewichte auf und vergleichen sie mit der Größe von x nach der Rotation, welche aus den Gewichten von x, y, z, A, B, C und D besteht. Teilen wir nun die Ungleichung durch $s'(x)$ so erhalten wir $\frac{s'(z)}{s'(x)} + \frac{s(x)}{s'(x)} \leq 1$. Deshalb folgt nun mit Hilfssatz 2.6, dass $log_2(\frac{s'(z)}{s'(x)}) + log_2(\frac{s(x)}{s'(x)}) \leq -2$.

3.Fall - Zig-Zag case

Die Zig-Zag Rotation ist erneut eine Doppelrotation, weshalb wir wieder eine amortisierte Laufzeit von $\mathcal{O}(3 \cdot (rank'(x) - rank(x)))$ zeigen wollen. Die tatsächlichen Kosten belaufen sich wieder auf zwei Einheiten, da wir zweimal rotieren. Die Betrachtung der Potentialänderung können wir einschränken auf die Betrachtung der Rangänderung der Knoten x, $y = parent(x)$ und $z = parent(y)$. Auch die kritischen Schritte der Laufzeitabschätzung der Zag-Zag-Rotation

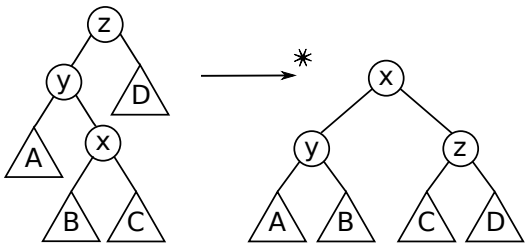
werden im Folgenden nach der Rechnung erläutert.

$$
\begin{aligned}
2 + \Phi'(t) - \Phi(t) &= 2 + (r'(x) + r'(y) + r'(z)) - (r(x) + r(y) + r(z)) \\
&= 2 + r'(x) - r(z) + r'(y) + r'(z) - r(y) - r(x) \\
&= 2 + r'(y) + r'(z) - r(y) - r(x) && [r'(x) = r(z)] \\
&\leq 2 + r'(y) + r'(z) - 2 \cdot r(x) && [r(x) \leq r(y)] \\
&= 2 + r'(y) + r'(z) - 2 \cdot r(x) + (2 \cdot r'(x) - 2 \cdot r'(x)) && [+0] \\
&= 2 + (r'(y) - r'(x)) + (r'(z) - r'(x)) + 2 \cdot (r'(x) - r(x)) \\
&= 2 + log_2(\tfrac{s'(y)}{s'(x)}) + log_2(\tfrac{s'(z)}{s'(x)}) + 2 \cdot (r'(x) - r(x)) && [Def.\ Rang] \\
&\leq 2 + (-2) + 2 \cdot (r'(x) - r(x)) && [Satz\ 2.6] \\
&= 2 \cdot (r'(x) - r(x)) \\
&\leq 3 \cdot (r'(x) - r(x))
\end{aligned}
$$

Auch hier sollten wir die einzelnen Schritte mit Hilfe folgender Abbildung etwas genauer erklären (vgl. Abbildung 2.3).

$r'(x) = r(z)$, da der Knoten x nach der Rotation dieselben Nachkommen hat wie der Knoten z vor der Rotation.

$r(x) \leq r(y)$, da der Knoten y zusätzlich zum Knoten x und seinen Nachkommen noch den Teilbaum A als Nachkommen hat.



$s'(y) + s'(z) \leq s'(x)$, da die Größe von y nach der Rotation aus den Gewichten von y, A und B bestehen und die Größe von z nach der Rotation aus den Gewichten von z, C und D. Zur Größe des Knoten x zählt jedoch zusätzlich zu diesen Gewichten noch sein eigenes. Also gilt die Ungleichung. Teilen wir nun noch durch $s'(x)$ so erhalten wir $\frac{s'(y)}{s'(x)} + \frac{s'(z)}{s'(x)} \leq 1$. Deshalb folgt nun mit Hilfssatz 2.6, dass $log_2(\frac{s'(y)}{s'(x)}) + log_2(\frac{s'(z)}{s'(x)}) \leq -2$. Somit haben wir den Hilfssatz 2.6 richtig angewendet. □

Natürlich sollten wir nun noch unsere Lemmata über das maximale und minimale Potenial sowie über die amortisierte Laufzeit der splay-Funktion erneuern. Denn im Hinblick auf die neuen Gewichte müssen wir einiges verallgemeinern. Aus dem verallgemeinerten Access Lemma folgt jedoch direkt die amortisierte Laufzeit für einen splay-Aufruf. Betrachten wir also einen Baum t mit n Knoten und Wurzel r, so ist nach Definition $size(r) = \sum_{i \in t} w(i) := W$. Über die Größe eines Knotens x können wir nur sagen, dass $size(x)$ mindestens so groß ist wie das Gewicht $w(x)$ von x. Dann gilt:

$$
\begin{aligned}
&rank(r) = log_2(size(r)) = log_2\left(\sum_{i \in t} w(i)\right) = log_2(W) \\
&rank(x) = log_2(size(x)) \geq log_2(w(x)).
\end{aligned}
$$

Insgesamt folgt also durch das verallgemeinerte Access Lemma:

$$\begin{aligned} T_{\text{am}}(splay(x)) &= \mathcal{O}\left(1 + 3 \cdot (rank(r) - rank(x))\right) \\ &= \mathcal{O}\left(1 + 3 \cdot (log_2(W) - log_2(w(x)))\right) \\ &= \mathcal{O}\left(1 + 3 \cdot log\left(\frac{W}{w(x)}\right)\right) \end{aligned}$$

Korollar 2.8. *Die amortisierte Laufzeit von splay(x) auf einem Splay tree t mit beliebigen Gewichten $w(i) \geq 0$ auf allen Knoten $i \in t$ ist in $\mathcal{O}\left(1 + 3 \cdot log\left(\frac{W}{w(x)}\right)\right)$.*

Lemma 2.9 (Maximales und minimales Potential für beliebige Gewichte)**.** *Das maximale Potential $\Phi(t)$ eines Splay trees t mit n Knoten und beliebigen nicht-negativen Gewichten $w(x)$ $\forall x \in t$ ist in $\mathcal{O}(n \cdot log(W))$ mit $W := \sum_{i \in t} w(i)$ und das minimale Potential $\Phi(t)$ ist in $\Omega(\sum_{x \in t} log(w(x))$.*

Beweis. Zunächst wollen wir eine obere Schranke für das Potential finden, also das maximale Potential. Dann kann jeder Knoten $x \in t$ höchstens eine Größe $size(x) = W$ und somit einen Rang $rank(x) = log_2(W)$ haben. Das passiert genau dann, wenn der Knoten x die Wurzel von t ist. Wir können nun das maximale Potential abschätzen, indem wir annehmen, dass jeder der n Knoten die Größe und somit den Rang der Wurzel hat.

$$\Phi_{\max} \leq n \cdot log(W)$$

Nun bestimmen wir noch eine untere Schranke für das Potential. Dazu betrachten wir die kleinstmögliche Größe, die ein Knoten x haben kann. Wenn x ein Blattknoten ist, so zählt zur Größe von x nur das eigene Gewicht. Also gilt für jeden Knoten $x \in t$: $size(x) \geq w(x)$ und somit $rank(x) \geq log(w(x))$. Nun können wir das minimale Potential abschätzen.

$$\Phi_{\min} \geq \sum_{i \in t} log(w(i))$$

□

Ohne den Knoten konkrete Gewichte zugewiesen zu haben, liefern uns das Korollar 2.8 und das Lemma 2.9 jedoch keine nützlichen Informationen. Dabei lassen sich durch die Verallgemeinerung einige interessante Eigenschaften der Splay trees zeigen. Eine dieser Eigenschaften ist die sogenannte Optimalität der Splay trees, welche wir im nächsten Abschnitt genauer betrachten wollen.

2.5. Die Optimalität der Splay trees

In diesem Abschnitt wollen wir uns mit der bestmöglichen Laufzeit einer Sequenz von m Zugriffen auf einen binären Suchbaum mit n Elementen beschäftigen. Zunächst betrachten wir dieses Problem unabhängig von den Splay trees. Angenommen wir wüssten, wie oft das i-te Element x_{i} in dieser Sequenz zugegriffen wird. Sagen wir, das i-te Element x_{i} wird mit der Häufigkeit $1 \leq q(i) \leq m$ zugegriffen. Wir nennen $p(i) = \frac{q(i)}{m}$ die Frequenz des i-ten Elements x_{i}. Es gilt $\sum_{i=1}^{n} q(i) = m$ und $\sum_{i=11}^{n} p(i) = 1$. Mit diesen Informationen könnten wir uns dann doch einen optimalen binären Suchbaum bauen, der uns die beste Laufzeit für die gegebene Sequenz von Zugriffen garantiert. Es soll also keinen anderen binären Suchbaum geben, der dieselben n Elemente enthält, uns aber eine bessere Laufzeit für die gegebe Sequenz liefert. Da wir uns den Baum vor den m Zugriffen aufbauen und während der Sequenz nicht mehr ändern,

nennen wir ihn auch den statischen optimalen Suchbaum. Tatsächlich ist es jedoch gar nicht so einfach, diesen optimalen statischen Suchbaum aufzustellen. Nach Melhorn [5] kann man aber einen nahezu optimalen binären Suchbaum erstellen, der asymptotisch gesehen dieselbe Laufzeit hat wie der optimale Suchbaum. Die Idee ist hierbei, den Baum im Hinblick auf die Frequenz balanciert zu halten. Das heißt, wir wollen während der m Zugriffe ungefähr gleich oft in den linken Teilbaum wie in den rechten Teilbaum der Wurzel laufen. Um das zu erreichen, sortieren wir unsere n Elemente zunächst nach ihrem Schlüssel. Sei also $x_1 \leq x_2 \leq ... \leq x_n$. Nun suchen wir das Element x_i, sodass gilt: $\sum_1^{i-1} p(i) \approx \sum_{i+1}^{n} p(i)$ und machen es zur Wurzel unseres nahezu optimalen Suchbaums. Im rechten und linken Teilbaum gehen wir genauso vor. Somit ist jeder Teilbaum unseres nahezu optimalen Suchbaums für seine Elemente wieder ein nahezu optimaler Suchbaum. Wir erläutern den Aufbau an einem kurzen Beispiel:
Seien also die Elemente $x_1, ..., x_7$ aufsteigend nach ihrem Schlüssel sortiert und ihre Frequenz wie folgt gegeben.

Tabelle 2.1.: Elemente, ihre Schlüssel und ihre Frequenz zum beispielhaften Aufbau eines nahezu optimalen binären Suchbaumes.

Element:	x_1	x_2	x_3	x_4	x_5	x_6	x_7
Schlüssel:	1	2	6	10	12	13	27
Frequenz:	$\frac{5}{16}$	$\frac{2}{16}$	$\frac{2}{16}$	$\frac{3}{16}$	$\frac{2}{16}$	$\frac{1}{16}$	$\frac{1}{16}$

Zunächst suchen wir die Wurzel des Baumes. Für x_3 gilt:
$p(1) + p(2) = p(4) + p(5) + p(6) + p(7) = \frac{7}{16}$, also sind der linke und rechte Teilbaum von x_3 als Wurzel sogar perfekt ausbalanciert hinsichtlich ihrer Frequenz. Wenn wir nun den rechten Teilbaum von x_3 betrachten, erkennen wir, dass wir ihn zwar nicht perfekt, aber annähernd ausbalancieren können. Dazu wählen wir x_5 als Wurzel des rechten Teilbaumes von x_3, denn $p(4) = \frac{3}{16} \approx \frac{2}{16} = p(6) + p(7)$. Im linken Teilbaum von x_3 sind nur noch zwei Elemente. Das heißt, die Wurzel dieses Teilbaumes hat nur ein Kind. Da das andere Kind nicht existiert und nicht aufgerufen wird, hat es Frequenz 0. Wir machen also x_1 mit der höheren Frequenz zur Wurzel des linken Teilbaumes, denn dadurch wird der Teilbaum eher ausbalanciert. Im rechten Teilbaum von x_5 ist es egal, wie die Elemente x_6 und x_7 angeordnet werden, denn sie haben dieselbe Frequenz. Es ergibt sich also folgender statisch optimaler Suchbaum für die gegebenen Zugriffshäufigkeiten:

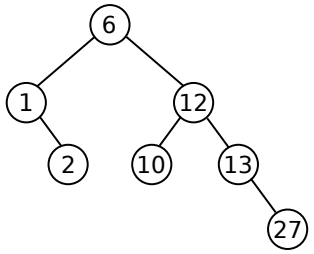


Abbildung 2.7.: Der statische optimale Suchbaum für die gegebenen Elemente und Häufigkeiten aus Tabelle 2.1

Kommen wir nun zur Laufzeitanalyse der Sequenz von m Zugriffen auf unseren statischen nahezu optimalen Suchbaum. Dazu benötigen wir eine Abschätzung von Melhorn [5], welche die Tiefe d_i eines Elementes x_i im Baum in Verbindung zu seiner Frequenz p(i) setzt.

Lemma 2.10. *Sei d_i die Tiefe eines Elementes x_i und p(i) die Frequenz dieses Elementes. Dann gilt:*

$$d_i + 1 \leq c \cdot log\Big(\tfrac{1}{p(i)}\Big) + 2,$$

wobei c >1 eine Konstante ist.

Wir benutzen dieses Lemma, ohne es zu beweisen. Ein Beweis findet sich in der Arbeit von Kurt Melhorn [5] auf der Seite 292. Sei t nun ein nahezu optimaler binärer Suchbaum, der nach dem obigen Schema aufgebaut ist und n Elemente besitzt. Wir wissen, dass d_i die Tiefe des Elementes x_i im Baum ist. Dann entspricht $d_\mathrm{i} + 1$ der benötigten Zeit, um das Element x_i zu suchen, denn wir brauchen $d_\mathrm{i} + 1$ Vergleiche auf dem Pfad von der Wurzel zu x_i. Somit ist die Laufzeit für die Sequenz der m Zugriffe $\sum_{i=1}^{n} q(i) \cdot (d_\mathrm{i} + 1)$. Nach Lemma 2.10 folgt nun:

$$\begin{aligned}
\sum_{i=1}^{n} q(i) \cdot (d_\mathrm{i} + 1) &\leq \sum_{i=1}^{n} q(i) \cdot \left(c \cdot log\left(\frac{1}{p(i)}\right) + 2 \right) \\
&= \left(\sum_{i=1}^{n} q(i) \cdot \left(c \cdot log\left(\frac{1}{p(i)}\right)\right)\right) + \left(\sum_{i=1}^{n} q(i) \cdot 2 \right) \\
&= m \cdot 2 + m \cdot \sum_{i=1}^{n} \frac{q(i)}{m} \cdot \left(c \cdot log\left(\frac{1}{p(i)}\right)\right) \\
&= m \cdot 2 + m \cdot \sum_{i=1}^{n} p(i) \cdot \left(c \cdot log\left(\frac{1}{p(i)}\right)\right) \\
&= O\left(m + m \cdot \sum_{i=1}^{n} p(i) \cdot log\left(\frac{1}{p(i)}\right)\right)
\end{aligned}$$

Und genau diese asymptotische Laufzeit für eine Sequenz von m Zugriffen wollen wir nun auch für unsere Splay trees zeigen. Dann wäre der Splay tree asymptotisch gesehen genauso optimal wie der statische optimale Suchbaum, obwohl er die Zugriffshäufigkeiten auf seine Elemente nicht kennt und sich mit jedem Zugriff verändert. Intuitiv können wir die Optimalität des Splay trees damit begründen, indem wir als Gewicht eines Knotens seine Frequenz wählen. Dann werden die schweren Knoten, also die Knoten, auf die oft zugegriffen wird, weiter oben im Baum stehen. Elemente, auf die häufig zugegriffen wird, verursachen also eine geringe Laufzeit.

Theorem 2.11 (Static Optimality Theorem)**.** *Sei t ein Splay tree mit n Elementen. Die Laufzeit für eine Sequenz von m Zugriffen auf diesen Baum t, wobei mit der Häufigkeit $1 \leq q(i)$ auf das Element x_i zugegriffen wird, ist in*

$$O\Big(m + m \cdot \textstyle\sum_{i=1}^{n} p(i) \cdot log\left(\frac{1}{p(i)}\right)\Big)$$

und hat somit asymptotisch gesehen die gleiche Laufzeit wie der optimale statische binäre Suchbaum.

Beweis. Wir berechnen die tatsächliche Laufzeit der Sequenz von m Zugriffen, indem wir wieder die amortisierten Laufzeit der Sequenz berechnen und die maximale Änderung im Potential. Dazu wählen wir als Gewicht $w(i)$ eines Knotens x_i seine Frequenz $p(i)$. Also sei

$$w(x) = p(x) \ \forall x \in t$$

Betrachten wir zunächst die amortisierten Kosten der Sequenz. Diese setzen sich zusammen aus den amortisierten Kosten der einzelnen Zugriffe. Nach Korollar 2.8 liegt die Laufzeit eines Zugriffs auf das Element x_i in $\mathcal{O}\left(1 + 3 \cdot log(\frac{W}{w(x_\mathrm{i})})\right)$. Da wir als Gewicht der Knoten ihre Frequenz gewählt haben, gilt $W = \sum_{x_\mathrm{i} \in t} w(x_\mathrm{i}) = \sum_{x_\mathrm{i} \in t} p(x_\mathrm{i}) = 1$. Außerdem greifen wir $q(i)$ mal auf das i-te Element x_i zu, somit liegt die Laufzeit von $q(i)$ Zugriffen auf das Element x_i in $\mathcal{O}\left(q(i) + 3 \cdot q(i) \cdot log(\frac{1}{p(x_\mathrm{i})})\right)$. Jetzt müssen wir nur noch die Laufzeit der Zugriffe auf jeden der n Knoten zusammenrechnen und erhalten

$$\begin{aligned}
\mathcal{O}\left(\sum_{i=1}^{n} q(i) + q(i) \cdot log\left(\frac{1}{p(i)}\right)\right) &= \mathcal{O}\left(\sum_{i=1}^{n} q(i) + \sum_{i=1}^{n} q(i) \cdot log\left(\frac{1}{p(i)}\right)\right) \\
&= \mathcal{O}\left(m + m \cdot \sum_{i=1}^{n} \frac{q(i)}{m} \cdot log\left(\frac{1}{p(i)}\right)\right) \\
&= \mathcal{O}\left(m + m \cdot \sum_{i=1}^{n} p(i) \cdot log\left(\frac{1}{p(i)}\right)\right)
\end{aligned}$$

Betrachten wir nun noch die maximale Potentialänderung. Nach Lemma 2.9 gilt:

$$\begin{aligned}
\Phi_\mathrm{max} &\leq n \cdot log(W) \\
\Phi_\mathrm{min} &\geq \sum_{i \in t} log(w(i))
\end{aligned}$$

Und damit können wir die maximale Änderung im Potential berechnen.

$$\begin{aligned}
\Phi_\mathrm{max} - \Phi_\mathrm{min} &\leq n \cdot log(W) - \sum_{i \in t} log(w(i)) \\
&= log(W) - log(w(1)) + log(W) - log(w(2)) + ... + log(W) - log(w(n)) \\
&= \sum_{i \in t} (log(W) - log(w(i))) \\
&= \sum_{i \in t} log\left(\frac{W}{w(i)}\right)
\end{aligned}$$

Nun setzen wir noch die Frequenz als Gewicht ein und erhalten als maximale Potentialänderung:

$$\begin{aligned}
\sum_{i \in t} log\left(\frac{1}{p(i)}\right) &= m \cdot \sum_{i \in t} \frac{1}{m} \cdot log\left(\frac{1}{p(i)}\right) \\
&\leq m \cdot \sum_{i \in t} p(i) \cdot log\left(\frac{1}{p(i)}\right)
\end{aligned}$$

Die letzte Abschätzung folgt, da wir auf jedes Element mindestens einmal zugreifen wollen. Insgesamt folgt nun für die tatsächliche Laufzeit einer Sequenz von m Zugriffen:

$$\begin{aligned} \mathrm{T}_{\mathrm{tat}}(\text{m splay-Ops.}) &= \mathrm{T}_{\mathrm{am}}(\text{m splay-Ops.}) + \Phi_{\mathrm{alt}}(t) - \Phi_{\mathrm{neu}}(t) \\ &= \mathcal{O}\left(m + m \cdot \sum_{i=1}^{n} p(i) \cdot log(\frac{1}{p(i)})\right) + m \cdot \sum_{i \in t} p(i) \cdot log\left(\frac{1}{p(i)}\right) \\ &= \mathcal{O}\left(m + m \cdot \sum_{i=1}^{n} p(i) \cdot log(\frac{1}{p(i)})\right) \end{aligned}$$

Und damit folgt die Behauptung. □

Insgesamt haben wir für die Funktionen der Splay trees zunächst eine amortisierte Laufzeit von $O(log(n))$ gezeigt. Die Splay trees sind also asymptotisch gesehen genauso effizient wie andere binäre Suchbäume (z.B. AVL-Bäume) auch. Zudem sind die Splay trees asymptotisch gesehen genauso effizient wie der statisch optimal aufgebaute binäre Suchbaum, der dieselben Elemente enthält. Dabei kennt der Splay tree die Zugriffshäufigkeiten im Gegensatz zum statisch optimal aufgebauten Baum nicht. Außerdem ist der statisch aufgebaute Baum auch nur für eine Sequenz von Zugriffen, die den gegebenen Zugriffshäufigkeiten entspricht, optimal. Ändern sich die Häufigkeiten, so verschlechtert sich die Laufzeit eines Zugriffes auf den statischen Baum. Zudem unterstützt der statische Baum keine Funktionen, die den Baum verändern würden, wie etwa *insert* oder *delete*. Der Splay tree jedoch ist dynamisch und passt sich eigenständig den sich verändernden Häufigkeiten an. Neben der effizienten Verwaltung von Elementen können die Splay trees auch als Baustein einer anderen dynamischen Datenstruktur verwendet werden, den sogenannten Link/Cut trees. Im nächsten Kapitel werden wir uns intensiv mit den Link/Cut trees beschäftigen und auch die Rolle der Splay trees erörtern.

3. Link/Cut trees

3.1. Dinic und das Problem des maximalen Flusses

In der Graphentheorie interessiert man sich oft für den maximalen Fluss auf einem Netzwerk. Wir wollen in diesem Abschnitt das Problem des maximalen Flusses und die Lösung durch den Algorithmus von Dinic wiederholen. In den folgenden Abschnitten werden wir die Datenstruktur Link/Cut trees betrachten und damit den Algorithmus von Dinic hinsichtlich seiner Laufzeit verbessern. Die Begriffe Netzwerk G, Restnetzwerk G_f, Levelgraph ${G_\mathrm{f}}^\mathrm{L}$ (auch geschichtetes Restnetzwerk genannt), blockierender Fluss g sowie die Kapazitätsfunktion c und der Fluss f auf einem Netzwerk werden dabei in diesem Abschnitt vorausgesetzt.
Sei ein Netzwerk $G = (V, E)$ und eine Kapazitätsfunktion $c : E \to \mathbb{R}_{>0}$ gegeben. Ein Netzwerk ist dabei ein gerichteter Graph mit einer ausgezeichneten Quelle s $\in V$, die nur ausgehende Kanten hat, sowie einer Senke t $\in V$, welche nur eingehende Kanten hat. Die Kapazitätsfunktion weist jeder Kante eine positive reelle Zahl zu. Gesucht ist nun ein Fluss f, der möglichst viele Einheiten von s nach t transportiert. Dabei muss die Kapazitätsbeschränkung sowie die Flusserhaltung eingehalten werden. Mathematisch gesehen suchen wir also eine Funktion $f : E \to \mathbb{R}_{>0}$ mit $0 \leq f(u, v) \leq c(u, v) \; \forall \; (u, v) \; \in E$, das heißt, der Fluss darf die Kapazität der Kante nicht überschreiten. Unter Flusserhaltung verstehen wir, dass jeder Knoten $x \in V \setminus \{s, t\}$ genauso viel Fluss weiterschicken muss, wie er zuvor erhalten hat.
Ziel ist es nun, den maximalen Fluss eines Netzwerks G zu bestimmen. Dafür hat Dinic eine schon recht effiziente Lösung gefunden. Seine Idee war es, möglichst schnell einen blockierenden Fluss g im Restnetzwerk ${G_\mathrm{f}}^\mathrm{L}$ zu finden und den Fluss um g zu erhöhen. Dabei müssen wir nur beschränkt viele blockierende Flüsse berechnen, denn die maximale Anzahl von Schichten in ${G_\mathrm{f}}^\mathrm{L}$ ist in $\mathcal{O}(|V|)$ und durch jeden gefundenen blockierenden Fluss erreichen wir, dass sich die Pfadlänge um mindestens eins erhöht. Dabei entspricht $|V|$ der maximalen Weglänge von s nach t. Wir werden nun etwas ausführlicher auf die Berechnung des blockierenden Flusses g eingehen.

1) Konstruiere das geschichtete Restnetzwerk ${G_\mathrm{f}}^\mathrm{L}$ mit Breitensuche, wobei wir beim Knoten s starten und den Knoten t suchen. Somit erhalten wir nur Kanten in ${G_\mathrm{f}}^\mathrm{L}$, die auf dem kürzesten Weg von s nach t liegen. Des Weiteren deklarieren wir s als den aktuellen Knoten u.

2) Wir erweitern den Pfad ausgehend von u solange, bis wir den Knoten t gefunden haben oder uns aber in einer Sackgasse befinden, wenn wir keine ausgehenden Kanten des aktuellen Knotens mehr finden. Solange also noch Kanten der Form $(u, v) \in {E_\mathrm{f}}^\mathrm{L}$ existieren, so fügen wir diese Kante unserem Pfad hinzu und machen v zum aktuellen Knoten, also $u = v$. Wir speichern uns den aktuellen Teilpfad stets ab. Falls nun der aktuelle Knoten $u = t$ ist, so gehe zu Schritt 4).

3) Wenn wir am Ende von Schritt 2) nicht beim Knoten t angelangt sind, so sind wir in eine Sackgasse gelaufen insofern $u \neq s$. Wir löschen den aktuellen Knoten u und all seine eingehenden Kanten. Zum aktuellen Knoten machen wir den Knoten, der auf dem bereits

gefundenen Teilpfad vor u lag.
War der aktuelle Knoten jedoch s, so haben wir den blockierenden Fluss von ${G_f}^L$ gefunden und sind mit diesem Levelgraphen fertig.

4) Zum Schritt 4) kommen wir nur, wenn wir einen Pfad von s nach t gefunden haben. Dann suchen wir die Kante m mit der minimalen Kapazität $c(m)$ auf diesem Pfad, welche auch die bottleneck Kante genannt wird, und erhöhen unseren Fluss f um diesen Wert $c(m)$. Die Kapazität aller Kanten auf dem Pfad wird um $c(m)$ verringert und die Kanten, die jetzt eine Kapazität von *0* haben, werden gelöscht. Wir setzten $u = s$ und gehen zu Schritt 2), um weitere Pfade von s nach t zu finden, bis wir schlussendlich den blockierenden Fluss gefunden haben.

Kommen wir zur Laufzeit zur Berechnung des blockierenden Flusses.

1) $\mathcal{O}(|E| + |V|)$, denn ${G_f}^L$ hat höchstens genauso viele Knoten und Kanten wie der Graph.

2) $\mathcal{O}(|V| \cdot |E|)$, denn es kostet $\mathcal{O}(|V|)$, um einen Pfad von s nach t zu finden, denn jeder Weg im Levelgraph hat eine Länge $\leq |V|$. Der Schritt 2) wird dabei höchstens $|E|$ mal aufgerufen, denn nach jeder erfolgreichen oder auch nicht erfolgreichen Pfadsuche wird eine Kante gelöscht.

3) $\mathcal{O}(|E|)$, denn jede Kante kann nur einmal gelöscht werden.

4) $\mathcal{O}(|V| \cdot |E|)$, denn wir müssen Operationen auf dem Pfad ausführen. Das Finden der minimalen Kante, sowie das Aktualisieren der Kapazitäten kostet also $\mathcal{O}(|V|)$. Nach jeder erfolgreichen Pfadsuche wird eine Kante gelöscht, weshalb dieser Schritt 4) nur $\mathcal{O}(|E|)$ mal auftreten kann.

Insgesamt folgt also eine Laufzeit von $\mathcal{O}(|V| \cdot |E|)$ für die Berechnung des blockierenden Flusses und eine Laufzeit von $\mathcal{O}(|V|^2 \cdot |E|)$ für den Algorithmus von Dinic, denn wir müssen ja maximal $|V|$-mal den Levelgraph ${G_f}^L$ aufstellen. Wir wollen nun das Beispiel zur Berechnung des blockierenden Flusses aus Abbildung 3.1 betrachten. Dabei sei ein Levelgraph gegeben. Wie das zugehörige Netzwerk bzw. Restnetzwerk aussieht, ist hierbei irrelevant.

Wir wollen unsere Aufmerksamkeit auf die im Beispiel 3.1 gefundenen Pfade legen. Die neuen roten Pfade laufen oftmals über Teilpfade, die zuvor schon entdeckt wurden. Im Beispiel sind diese als dicke Kanten gekennzeichnet. Jedoch werden diese Teilpfade in unserem Algorithmus zur Berechnung des blockierenden Flusses nicht berücksichtigt und müssen erneut abgelaufen werden. Wenn wir nun eine Möglichkeit hätten, diese Teilpfade zu verwalten und somit einige Operationen schneller durchzuführen, dann könnten wir die Laufzeit des Algorithmus zur Berechnung des blockierenden Flusses verbessern und somit auch die Laufzeit des Dinic-Algorithmus. An dem Faktor $|E|$ können wir jedoch nichts ändern, da wir jede Kante betrachten müssen, um alle möglichen Pfade zu finden. Aber der Faktor $|V| = n$, der darauf beruht, dass wir einen Pfad finden und Operationen entlang dieses Pfades ausführen, könnte auf $\mathcal{O}(log(n))$ verbessert werden. Unsere Datenstruktur, welche die gefundenen Teilpfade verwaltet, müsste also folgende Funktionen in $\mathcal{O}(log(n))$ unterstützen.

- Eine Kante einem Teilpfad hinzufügen.
- Die minimale Kante auf dem Pfad von s nach t finden.

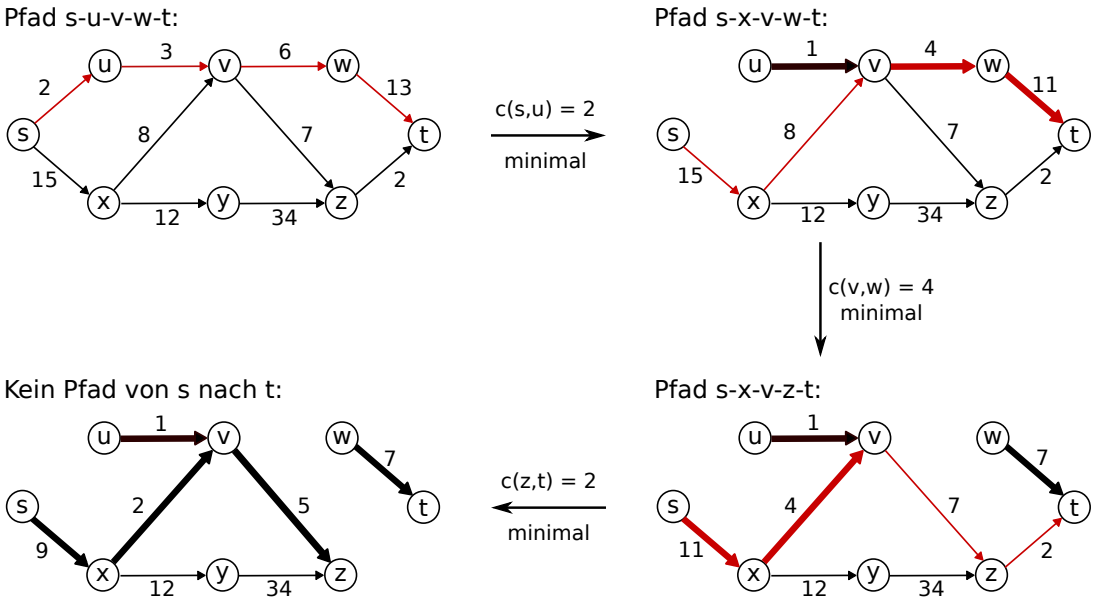


Abbildung 3.1.: Die Berechnung eines blockierenden Flusses für einen gegebenen Levengraphen. Die roten Kanten stellen den aktuell gefundenen s-t-Pfad dar, die dicken Kanten kennzeichnen Teilpfade, die zuvor bereits gefunden wurden.

- Allen Knoten auf dem Pfad von s nach t einen Wert abziehen.
- Eine Kante auf einem Pfad löschen, wenn die Kapazität 0 ist.

Im nächsten Abschnitt werden wir die Datenstruktur der sogenannten Link/Cut trees, die zur Verwaltung von Bäumen dient, genauer betrachten und sehen, dass wir mit ihnen die gewünschte Laufzeitverbesserung des Dinic Algorithmus erreichen können.

3.2. Was ist ein Link/Cut tree?

Unter Link/Cut trees versteht man zunächst einmal einen Wald M, also eine Menge von Bäumen. Jeder Baum $T \in M$ hat eine Wurzel und jeder Knoten $x \in T$ hat einen Kostenwert $cost(x)$. Da wir im vorherigen Kapitel sehr viel über binäre Suchbäume gesprochen haben, ist zu erwähnen, dass es sich bei einem Link/Cut tree T weder um einen binären noch um einen sortierten Baum handelt. Das heißt, jeder Knoten $x \in T$ kann beliebig viele Kinder haben und die Knoten sind keineswegs nach ihren Kostenwerten sortiert.
Die Link/Cut trees sind dynamische Bäume und können schrumpfen und wachsen, indem man Bäume der Menge verlinkt oder einen Baum in zwei Bäume zerteilt. Sie unterstützen folgende Operationen, die wir im Laufe dieses Kapitels noch genauer analysieren werden:

makeTree(x): Erstellt einen neuen Link/Cut tree, der nur aus dem Knoten x besteht.

findCost(x): Gibt den Kostenwert des Knotens x zurück.

findRoot(x): Gibt die Wurzel des Link/Cut trees zurück, in dem sich der Knoten x befindet.

findMin(x): Diese Funktion betrachtet alle Knoten auf dem Pfad von x zur Wurzel des Link/Cut trees und gibt den Knoten mit minimalen Kostenwert auf diesem Pfad zurück.

addCost(x,c): Betrachtet erneut den Pfad vom Knoten x zur Wurzel des Link/Cut trees (findRoot(x)) und addiert auf den Kostenwert jedes Knotens dieses Pfades eine Konstante c.

cut(x): Trennt den Knoten x von seinem Vater. Nun bildet der Knoten x mit all seinen Nachkommen einen neuen Link/Cut tree.

link(x,y): Der Knoten x ist die Wurzel eines Link/Cut trees und der Knoten y ein beliebiger Knoten eines anderen Link/Cut tree. Dann wird der Knoten x das Kind von y. Somit sind zwei Link/Cut trees zu einem zusammengewachsen.

Im nächsten Abschnitt wollen wir die Datenstruktur der Link/Cut trees betrachten.

3.3. Die Datenstruktur der Link/Cut trees

Da wir noch keine Struktur auf unserem Baum T haben, kann es sein, dass in T sehr lange Pfade vorkommen. Operationen auf langen Pfaden kosten jedoch viel Laufzeit und sind somit im wahrsten Sinne des Wortes ungünstig für uns. Die Idee ist es nun, Pfade in T intern als binäre Suchbäume zu repräsentieren.

3.3.1. Der Original und der Virtual Tree

Um eine Struktur zu erhalten, unterteilen wir unseren Baum T zunächst in Pfade. Dazu bezeichnen wir jede Kante entweder als **solide Kante** oder als **dashed Kante**. Dabei soll gelten, dass jeder Knoten mit Ausnahme der Blattknoten genau eine soldie Kante zu einem Kind hat. Somit erreichen wir, dass unser Baum T in **solide Pfade** unterteilt ist. Es entstehen Pfade, weil jeder Knoten nur ein solides Kind hat. Jede Wurzel eines soliden Pfads, mit Ausnahme der Wurzel von T, ist nun wiederum durch eine dashed Kante zu einem anderen soliden Pfad verbunden. Der Link/Cut tree auf dem wir keine Veränderung vorgenommen haben, nennen wir im Folgenden den **Original tree**. Der zugehörige Link/Cut tree, der bereits in solide und dashed Kanten unterteilt wurde, nennen wir den **Solid/Dashed tree**. Die Abbildung 3.2 ist von Saxena [6] fast identisch übernommen und zeigt uns, wie ein Baum in solide Pfade aufgeteilt sein könnte. In unserem Beispiel aus der Abbildung 2.2 bilden die A,B,C und D Knoten jeweils einen soliden Pfad. Aber auch die Knoten E,F,G,H und J bilden jeweils einen soliden Pfad. Dieser besteht hierbei jedoch nur aus einem Knoten. Wir wollen intuitiv verstehen, was diese Unterteilung in solide und dashed Kanten für unsere Laufzeitverbesserung des Dinic-Algorithmus bedeutet. Wir betrachten die Knoten des Baumes als Knoten des Levelgraphen aus dem Problem des blockierenden Flusses. Dabei ist die Wurzel des Link/Cut trees T die Senke t, denn alle Pfade aus T laufen auf diesen Knoten zu. Die gefundenen Teilpfade aus dem Levelgraph entsprechen also den Pfaden im Link/Cut tree. So bildet der solide Pfad, der die Wurzel von T beinhaltet, den aktuellen Pfad nach t im Levelgraphen. Aber auch Pfade aus T die unter Anderem über dashed Kanten zur Wurzel verlaufen, liefern wertvolle Informationen, denn sie zeigen auch bereits gefundene Teilpfade aus dem Levelgraphen an. Die dashed Kanten sind dabei lediglich ein Zeichen dafür, dass dieser Pfad schon veraltet ist und ein neuer Pfad über den Knoten mit dem dashed Kind gefunden wurde. Wie genau die Link/Cut trees benutzt werden, um die gefunden Teilpfade im Dinic-Algorithmus zu verwalten, sehen wir aber erst in Abschnitt 3.6.
Die soliden Pfade sind sehr unhandlich und wie wir wissen, kosten Operationen auf langen Pfaden viel Laufzeit. Deshalb repräsentieren wir nun jeden soliden Pfad als binären Suchbaum und nennen diesen dann den **soliden Baum**. Dabei soll die Sortierung der Knoten im Suchbaum nach deren **Höhe im soliden Pfad** erfolgen. Die Wurzel im soliden Pfad wird dann als

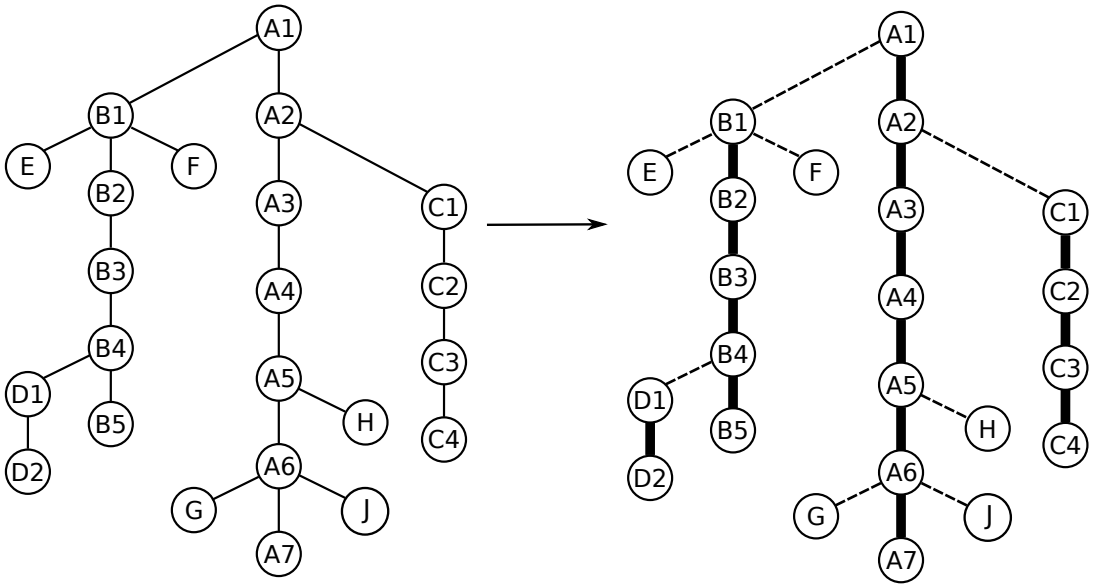


Abbildung 3.2.: Der Original tree (links). Ein Baum T wurde in solide Pfade zerlegt und ergibt den Solid/Dashed tree (rechts). Dicke Kanten sind hierbei soldie, gestrichelte Kanten dashed.

größtes Element im soliden Baum angesehen und der Blattknoten als kleinstes. Das hat den Sinn, dass man später die Struktur des Pfades wieder aus dem Suchbaum gewinnen kann. Die Struktur des Pfades ist dabei sehr wichtig, denn sie gibt an, wie die Knoten im Levelgraphen miteinander verbunden sind. Den Baum T, in dem alle soliden Pfade als binäre Suchbäume repräsentiert werden, nennen wir den **Virtual tree**. Zwar liefern der Solid/Dashed tree und der Virtual tree dieselben Informationen über einen Link/Cut tree, jedoch ist der Virtual tree die Baumstruktur, die später implementiert wird, da sie eine bessere Laufzeit garantiert. Die Abbildung 3.3 zeigt uns, wie der Virtual tree für unser vorheriges Beispiel aussieht. Als binären Suchbaum werden wir in unseren weiteren Betrachtungen **Splay trees** verwenden. Dabei werden die Funktionen des Splay trees nur auf dem soliden Baum ausgeführt. Alle dashed Kinder bleiben unangetastet bei ihrem Vater. Erwähnenswert ist, dass wir alle Operationen auf dem Solid/Dashed tree ausführen wollen. Jedoch hat dieser wie der Original tree unter anderem sehr lange Pfade, weshalb die Operationen sehr teuer werden können. Deshalb repräsentieren wir den Solid/Dashed tree als Virtual tree, um die Laufzeit zu verbessern. Das heißt aber, dass sich der Solid/Dashed tree und der Virtual tree gleich verhalten müssen. Wir wollen also noch etwas genauer betrachten, wie sich die soliden Bäume im Vergleich zu den soliden Pfaden verhalten. Im soliden Baum hat jeder Knoten x bis zu zwei solide Kinder. Dabei ist das rechte solide Kind im soliden Pfad höher als x und das linke solide Kind tiefer, da die Sortierung im soliden Baum der Höhe im soliden Pfad entspricht. Interessant ist auch die Betrachtung der Wurzel. So ist die Wurzel des Solid/Dashed tree nicht unbedingt auch die Wurzel des Virtual tree. In unserem Beispiel kann man sehen, dass die Wurzel des Solid/Dashed tree $A1$ sich im Virtual tree im selben soliden Baum wie die Wurzel des Virtual trees $A4$ befindet. Und zwar als größtes Element dieses soliden Baumes. Diese Ordnung liefert uns aus dem soliden Baum auch wieder den soliden Pfad. Denn wenn $A1$ das größte Element dieses Baumes war, so liegt es im korrespondierenden soliden Pfad am höchsten und kann somit nur die Wurzel des

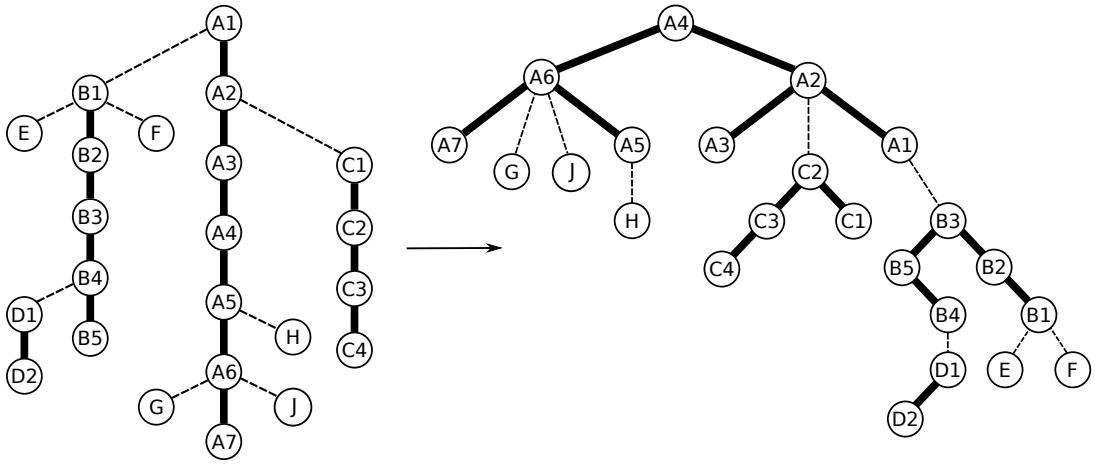


Abbildung 3.3.: Der Virtual tree (rechts). Die soliden Pfade aus dem Solid/Dashed tree werden als binäre Suchbäume dargestellt.

soliden Pfades sein. Wie wir im vorherigen Abschnitt gesehen haben, soll der Link/Cut tree nun Operationen auf einem Pfad, der an der Wurzel des Baumes endet, unterstützen. Wenn wir nun beispielhaft Operationen auf dem Pfad von $A5$ zur Wurzel $A1$ des Original trees T durchführen wollen, müssen wir zunächst die Knoten auf dem Pfad von $A5$ zu $A1$ im Virtual tree ausfindig machen. Dazu können wir aber einfach den Knoten $A5$ durch den Aufruf von splay($A5$) zur Wurzel in seinem soliden Baum machen. Dann liegen alle Elemente, die größer sind als $A5$, im rechten soliden Teilbaum von $A5$. Hierbei wären ja alle größeren Elemente definiert als die Knoten, die im soliden Pfad über $A5$ liegen und somit auf dem Pfad von $A5$ nach $A1$.
Was passiert aber wenn wir Operationen auf dem Pfad von $C2$ zur Wurzel $A1$ des Solid/Dashed trees durchführen wollen? Da $C2$ und $A1$ nicht im selben soliden Baum liegen, reicht es nicht aus $C2$ in seinem soliden Baum durch einen splay-Aufruf zur Wurzel zu machen, um alle Elemente auf dem Pfad von $C2$ zu $A1$ einfach ablesen zu können. Wir brauchen also noch eine Möglichkeit, solide Bäume miteinander zu verbinden. Das erreichen wir durch die sogenannte splice-Operation.

3.3.2. Die splice-Operation

Zunächst betrachten wir die splice-Operation im Solid/Dashed tree. Sei y das solide Kind von x und z ein dashed Kind von x wie in Abbildung 3.4 gezeigt. Dann wird durch den Aufruf von *splice*(z) die x-z Kante solide und die x-y Kante dashed. Es entstehen zwei neue solide Pfade. Die splice-Operation auf dem Virtual Tree verläuft ähnlich. Wir müssen nur aufpassen, dass wir die Ordnung des soliden Baums nicht verändern. Die Sortierung im soliden Baum soll natürlich nach der splice-Operationen immer noch der Höhe im soliden Pfad entsprechen. Da z das Kind von x ist, liegt es im soliden Pfad tiefer als x. Somit müssen wir z zum linken soliden Kind machen. Mit dieser splice-Operation können wir nun einfach die Elemente auf dem soliden Pfad von $C2$ nach $A1$ im Virtual tree des Beispiels aus Abbildung 3.3 ausfindig machen. Durch die Operation *splice*($C2$) liegt $C2$ nun im selben soliden Baum wie die Wurzel $A1$. Wir können

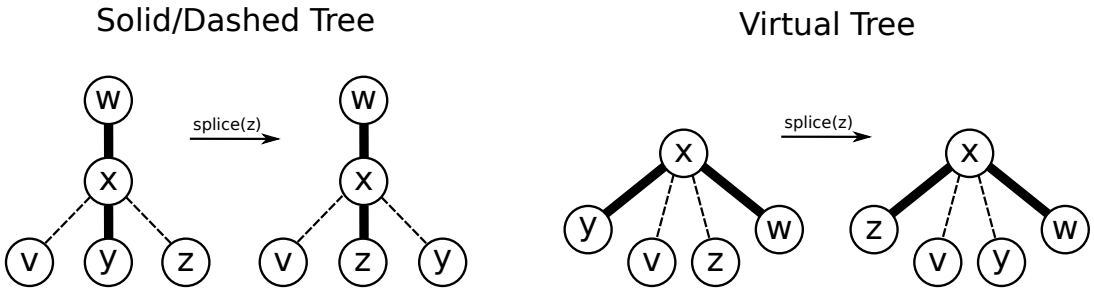


Abbildung 3.4.: Die splice-Operation vertauscht ein solides mit einem dashed Kind. Die Abbildung zeigt dies einmal im Solid/Dashed tree und einmal im Virtual tree

also wieder unsere vorherige Strategie anwenden. Durch den Aufruf von $splay(C2)$ wird $C2$ zur Wurzel im soliden Baum. Alle Elemente, die im soliden Pfad über $C2$ liegen, sind nun im soliden Baum im rechten Teilbaum von $C2$.
Die Strategie einen Knoten zur Wurzel zu machen, um Operationen zu vereinfachen, ist uns schon von den Splay trees bekannt. Auch die Link/Cut trees haben eine ähnliche Funktion namens $access()$. Durch den Aufruf von $access(x)$ wird ein Knoten x zur Wurzel des gesamten Virtual tree gemacht und liegt somit insbesondere in einem soliden Baum, wie die Wurzel des Solid/Dashed tree. Wir werden sehen, dass die access-Funktion als Subroutine für alle unterstützten Funktionen dient. Wir wollen nun im nächsten Abschnitt genauer betrachten, wie wir systematisch vorgehen können, um in einem Virtual tree T einen Knoten $x \in T$ zur Wurzel zu machen.

3.3.3. Access() im Virtual tree

Im Folgenden betrachten wir nun die Funktion access(), die als Subroutine für alle Funktionen eines Link/Cut trees dient. Sei also T ein Link/Cut tree mit Wurzel r im Original tree und $x \in T$ ein beliebiger Knoten. Dann wird durch den Aufruf von $access(x)$ der Knoten x zur Wurzel des Virtual trees T. Zuvor haben wir schon eine Strategie entwickelt, die einen Knoten x zur Wurzel des Virtual trees macht, insofern dieser Knoten x im selben soliden Baum liegt wie die Wurzel r. Dann müssen wir nur noch in dem soliden Baum $splay(x)$ aufrufen. Nun muss aber der Knoten x nicht im selben soliden Baum liegen wie die Wurzel r. Durch geeignete splice-Operationen können wir aber erreichen, dass sich die zwei Knoten im selben soliden Baum befinden. Wir systematisieren unsere Idee durch folgenden Algorithmus und führen diesen beispielhaft Schritt für Schritt aus:

1. Schritt

Sei x der Knoten, den wir zur Wurzel des Virtual Trees machen wollen. Zunächst wollen wir alle Kanten auf dem Weg von x zur Wurzel finden, welche dashed sind. Denn diese müssen wir durch eine splice-Operation solide machen. Dann splayen wir x in seinem soliden Baum, somit ist x die Wurzel in seinem soliden Baum und auf jeden Fall durch eine dashed Kante zu seinem (eventuell neuen) Vater $parent(x)$ verbunden. Der Knoten $parent(x)$ liegt nun in einem anderen soliden Baum als x. Wir splayen also $parent(x)$ in seinem soliden Baum. Somit ist er die Wurzel seines soliden Baumes und wiederum durch eine dashed Kante zu seinem Vaterknoten verbunden. Diesen Knoten splayen wir wieder. Als Ergebnis erhalten wir, dass alle Vorfahren von x in ihren

soliden Bäumen durch einen splay-Aufruf zur Wurzel des soliden Baumes gemacht wurden. Der Pfad von x zur Wurzel des Virtual trees ist somit dashed. Außerdem können wir aussagen, dass wenn wir in k soliden Bäumen die splay-Operation durchgeführt haben, im Virtual tree der Pfad von x zur Wurzel des Virtual trees nun genau die Länge k hat.

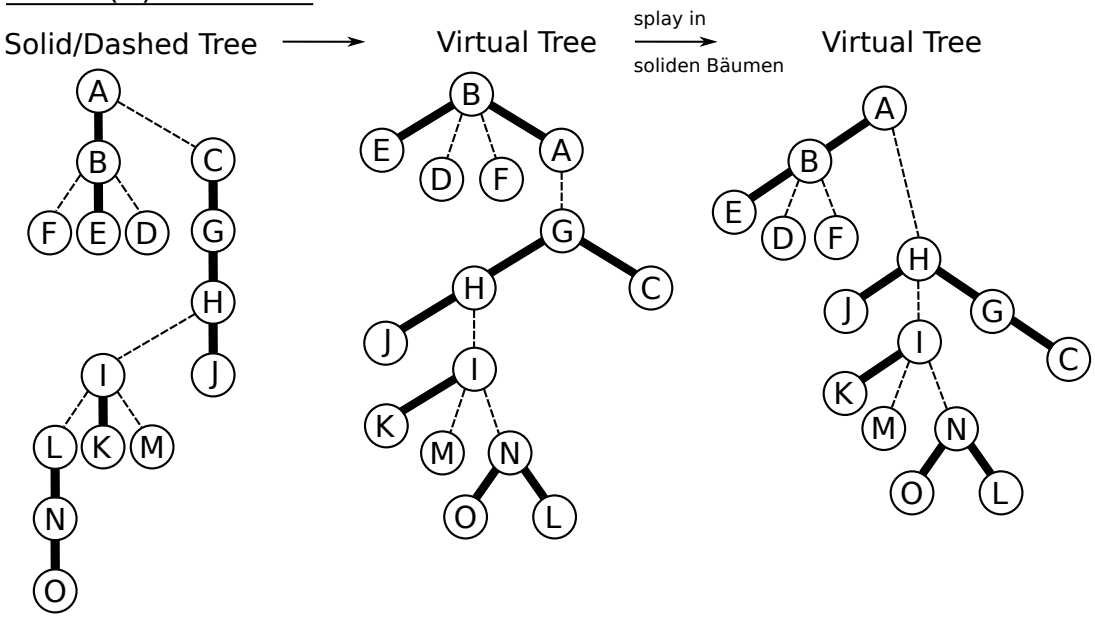


Abbildung 3.5.: Der 1. Schritt von access(N). Die Vorfahren von N werden jeweils durch einen splay-Aufruf zur Wurzel ihres soliden Baumes. Der Pfad von N zur Wurzel des Virtual trees ist nun dashed. Der obige Beispielbaum ist von Erik Demaine übernommen [2].

2. Schritt

In diesem Schritt laufen wir im Virtual tree den Pfad vom Knoten x zur Wurzel des Virtual trees entlang und machen dabei durch splice-Operationen jede Kante solide. Nach Schritt 1 wird jede Kante von x zu seinen Vorfahren dashed sein. Durch die Ausführung von Schritt 2 erreichen wir, dass der Knoten x nun im selben soliden Baum liegt wie die Wurzel des Virtual trees und insbesondere im selben wie die Wurzel des Solid/Dashed treest. Betrachten wir nach den splice-Operationen die soliden Pfade im Solid/Dashed tree, so stellen wir fest, dass x nun auf einem Pfad mit der Wurzel des Original trees liegt. Die Wurzel des Original tree ist hierbei immer die Wurzel des Solid/Dashed tree. Eine Ausführung findet sich in Abbildung 3.6.

3. Schritt

Der Knoten x liegt nach den vorherigen beiden Schritten nun in einem soliden Baum mit der Wurzel des Original trees. Es genügt nun, durch den Aufruf von $splay(x)$, den Knoten x zur Wurzel dieses soliden Baumes zu machen. Da es keinen höhergelegenen soliden Baum geben kann, da die Wurzel des Original treet in diesem soliden Baum enthalten ist, wird x somit zur Wurzel des Virtual Trees. Die Ausführung des 3. Schritts findet sich in Abbildung 3.7.

access(N): 2. Schritt

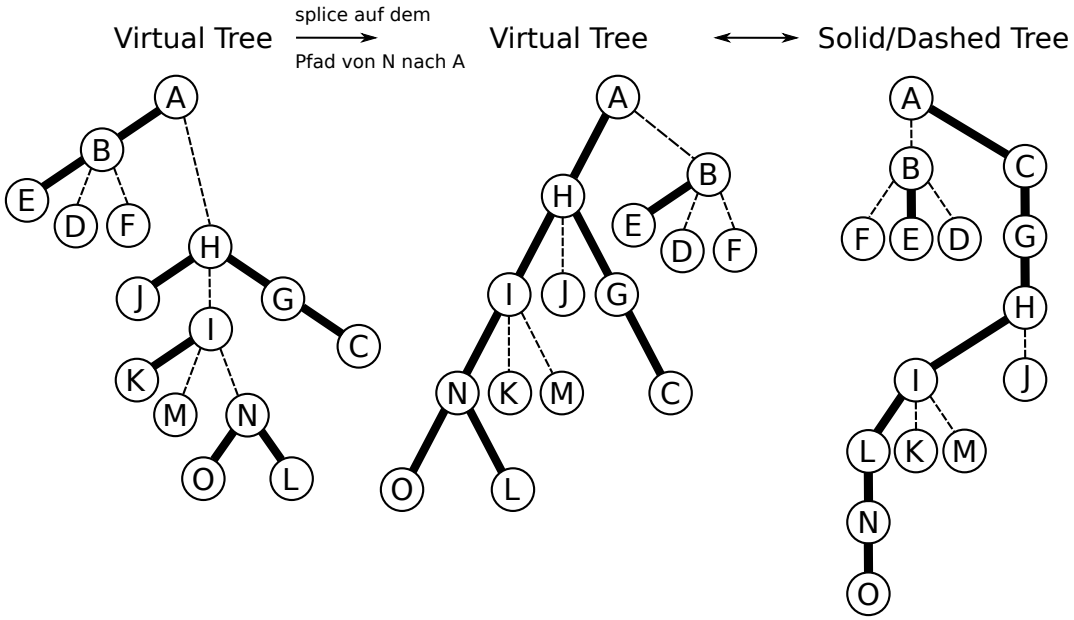


Abbildung 3.6.: Der 2. Schritt von access(N). Die Kanten auf dem Pfad von N zur Wurzel A des Virtual trees werden durch splice-Operationen solide. Im rechten Bild werden die soliden Pfade im Solid/Dashed tree nach den splice-Operationen gezeigt. N liegt nun in einem soliden Pfad mit der Wurzel A.

access(N): 3. Schritt

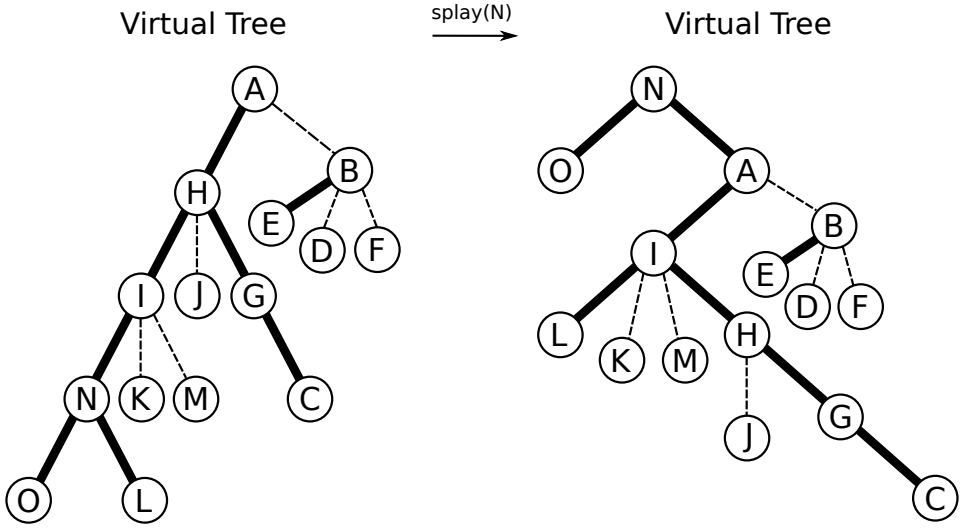


Abbildung 3.7.: Der 3. Schritt von access(N). Der Knoten N wird durch eine einzelne splay-Operation zur Wurzel des Virtual trees befördert.

Mithilfe der access-Funktion können wir nun also einen Knoten zur Wurzel des Virtual trees machen. Wir sehen in Abschnitt 3.4, dass wir alle Funktionen, die durch einen Link/Cut tree unterstützt werden, mithilfe der access-Funktion implementieren können. Bevor wir jedoch die Implementation der Funktionen betrachten, müssen wir jedem Knoten neben seinem Kostenwert noch weitere Werte zuweisen. Diese werden uns später helfen, auf einem Pfad den Knoten mit minimalsten Kosten zu finden sowie auf alle Kostenwerte der Knoten dieses Pfades eine Konstante zu addieren.

3.3.4. Handhabung der Kostenwerte im Virtual Tree

Jeder Knoten x eines Link/Cut trees hat, wie wir bereits wissen, einen Kostenwert $cost(x)$. Tatsächlich wird dieser jedoch im Virtual tree nicht direkt bei seinem Knoten abgespeichert. Denn wenn wir zum Beispiel die Funktion $addCost(x, c)$ aufrufen, welche eine Konstante c auf die Kostenwerte aller Knoten des soliden Pfades von x zur Wurzel des Original trees addiert, dann müssten wir jeden dieser Knoten einzeln aufrufen und ihre Kostenwerte aktualisieren. Das würde aber unsere gewünschte Laufzeit von $\mathcal{O}(log(n))$ sprengen. Durch die im folgenden Abschnitt vorgestellte δ-Darstellung wird uns einiges vereinfacht.
Sei $minCost(x)$ der Wert des Knotens mit minimalsten Kosten im soliden Teilbaum, der von x aufgespannt wird. Dann definieren wir folgende Parameter:

$$\delta min(x) = cost(x) - minCost(x)$$

$$\delta cost(x) = \begin{cases} cost(x) & x \text{ ist die Wurzel eines soliden Baumes} \\ cost(x) - cost(parent(x)) & \text{sonst} \end{cases}$$

Diese zwei Parameter sind die einzigen, die wir direkt bei einem Knoten des Virtual trees speichern. Wir müssen jedoch noch betrachten, wie wir die Parameter beim Durchführen von Operationen aktualisieren. Dazu sind die folgenden zwei Lemmata von Nutzen.

Lemma 3.1. *Für einen Knoten x, der nicht die Wurzel eines soliden Baumes ist, gilt:*

$$minCost(x) = \delta cost(x) - \delta min(x) + cost(parent(x))$$

Beweis. Die Behauptung folgt direkt durch das Einsetzen der Definitionen der beiden Parameter.

$$\begin{aligned} &\delta cost(x) - \delta min(x) + cost(parent(x)) \\ &= [cost(x) - cost(parent(x))] - [cost(x) - minCost(x)] + cost(parent(x)) \\ &= minCost(x) \end{aligned}$$

□

Mit dem Lemma 3.1 können wir nun folgendes Lemma beweisen.

Lemma 3.2. *Sei z ein Knoten eines soliden Baumes und u, v seine beiden soliden Kinder, davon u das linke und v das rechte Kind. Dann gilt:*

$$\delta min(z) = \max\{0,\ \delta min(u) - \delta cost(u),\ \delta min(v) - \delta cost(v)\}$$

Beweis. Betrachten wir für den Beweis zunächst $minCost(z)$. Da $minCost(z)$ den minimalen Knoten im Teilbaum von z beschreibt, gibt es nur drei Möglichkeiten, wo sich dieser Knoten befinden könnte. Entweder er liegt im linken bzw. rechten Teilbaum von z oder es ist der Knoten z selbst. Es gilt:

$$minCost(z) = min\{cost(z),\ minCost(u),\ minCost(v)\}$$

Und damit folgt nun für $\delta min(z)$:

$$\begin{aligned}\delta min(z) &= cost(z) - minCost(z)\\ &= cost(z) - min\{cost(z),\ minCost(u),\ minCost(v)\}\\ &= max\{0,\ cost(z) - minCost(u),\ cost(z) - minCost(v)\}\end{aligned}$$

Mit dem Lemma 3.1 und unter Berücksichtigung, dass z der Vater von u und v ist, folgt nun:

$$\begin{aligned}\delta min(z) &= max\{0,\ cost(z)\text{-}minCost(u),\ cost(z)\text{-}minCost(v)\}\\ &= max\{0,\ cost(z)\text{-}\delta cost(u)+\delta min(u)\text{-}cost(z),\ cost(z)\text{-}\delta cost(v)+\delta min(v)\text{-}cost(z)\}\\ &= max\{0,\ \delta min(u)\text{-}\delta cost(u),\ \delta min(v)\text{-}\delta cost(v)\}\end{aligned}$$

□

Jede Operation, die etwas an der Datenstruktur verändert, besteht aus Rotationen und splice-Operationen. Und genau diese zwei elementaren Operationen wollen wir nun hinsichtlich der Aktualisierungen der δmin und $\delta cost$ Werte betrachten. Was mit δmin und $\delta cost$ bei dem Aufruf der Funktionen $link(x, y)$ und $cut(x)$ passiert, wird im nächsten Abschnitt diskutiert. Kommen wir nun also zunächst zu den Rotationen in einem Link/Cut tree. Es genügt eine Einzelrotation zu betrachten, wie sie in folgender Abbildung gezeigt ist. Die Abbildung findet sich so ähnlich in der Ausarbeitung von Saxena (Figure 12.5) [6].

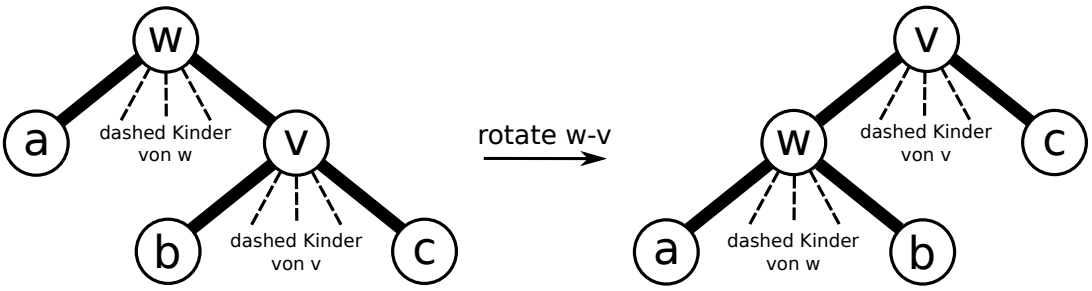


Abbildung 3.8.: Eine Einzelrotation im Virtual tree.

Im Folgenden bezeichne nun δmin und $\delta cost$ die im Knoten gespeicherten Werte vor der Rotation und $\delta min'$ und $\delta cost'$ nach der Rotation. Wir können $\delta min'$ und $\delta cost'$ einfach aus den alten Werten berechnen.
Wir beschäftigen uns zunächst mit $\delta cost$. Da die Kostenwerte $cost$ durch die Rotation nicht verändert werden, müssen wir die eventuelle Änderung der Väter der Knoten betrachten. Durch die Rotation erhält der Knoten v einen neuen Vater, nämlich den vorherigen Vater des Knoten w. Mit dieser Erkenntnis können wir $\delta cost'(v)$ auf bereits bekannte Werte zurückführen. Die

folgenden Rechnungen orientieren sich an denen von Saxena [6].

$$\begin{aligned}
\delta cost'(v) &= cost(v) - cost(parent'(v)) \\
&= cost(v) - cost(parent(w)) \\
&= cost(v) \underbrace{-cost(w) + cost(w)}_{=0} - cost(parent(w)) \\
&= (cost(v) - cost(w)) + (cost(w) - cost(parent(w))) \\
&= \delta cost(v) + \delta cost(w)
\end{aligned}$$

Wir berechnen noch die anderen $\delta cost$ Werte. Auch die Väter der Knoten w und b verändern sich durch die Rotation und somit auch ihre $\delta cost$ Werte. Der neue Vater von w ist v und der neue Vater von b ist w. Somit gilt:

$$\begin{aligned}
\delta cost'(w) &= cost(w) - cost(parent'(w)) \\
&= cost(w) - cost(v) \\
&= -\delta cost(v)
\end{aligned}$$

$$\begin{aligned}
\delta cost'(b) &= cost(b) - cost(parent'(b)) \\
&= cost(b) \underbrace{-cost(v) + cost(v)}_{=0} - cost(w) \\
&= (cost(b) - cost(v)) + (cost(v) - cost(w)) \\
&= \delta cost(b) + \delta cost(v)
\end{aligned}$$

Da die Knoten a und c weder ihren Vater noch ihren Kostenwert ändern, gilt:

$$\begin{aligned}
\delta cost'(a) &= \delta cost(a) \\
\delta cost'(c) &= \delta cost(c)
\end{aligned}$$

Kommen wir nun noch zur Anpassung der δmin Werte. Durch die Rotation verändern diese sich für die Knoten w und v, denn die beiden Knoten wechseln ihre Kinder. Wir stellen fest, dass v nach der Rotation genau die Nachkommen hat, die der Knoten w vor der Rotation hatte. Somit gilt $minCost'(v) = minCost(w)$ und es folgt:

$$\begin{aligned}
\delta min'(v) &= cost(v) - minCost'(v) \\
&= cost(v) - minCost(w) \\
&= cost(v) \underbrace{-cost(w) + cost(w)}_{=0} - minCost(w) \\
&= (cost(v) - cost(w)) + (cost(w) - minCost(w)) \\
&= \delta cost(v) + \delta min(w)
\end{aligned}$$

Da die Knoten a, b und c weder Kostenwerte noch Nachkommen wechseln, gilt:

$$\delta min'(x) = \delta min(x) \qquad für\ x \in a, b, c$$

Und durch Lemma 3.2 gilt für den Knoten w:

$$\begin{aligned}\delta min'(w) &= max\{0, \delta min'(a) - \delta cost'(a), \delta min'(b) - \delta cost'(b)\} \\ &= max\{0, \delta min(a) - \delta cost(a), \delta min(b) - \delta cost(b)\}\end{aligned}$$

Wir aktualisieren also stets bei jeder durchgeführten Rotation unsere δmin und $\delta cost$ Werte. Diese Aktualiserung kostet nur konstant viel Zeit, denn wie wir gesehen haben, lassen sich die neuen Parameter aus den alten berechnen. Die konstanten Laufzeitkosten können also mit den konstanten Laufzeitkosten der Rotation verrechnet werden.
Bleibt noch die splice-Operation hinsichtlich der $\delta cost$ und δmin Aktualisierungen zu betrachten. Eine allgemeine splice-Operation im Virtual tree ist in folgender Abbildung 3.9 gezeigt:

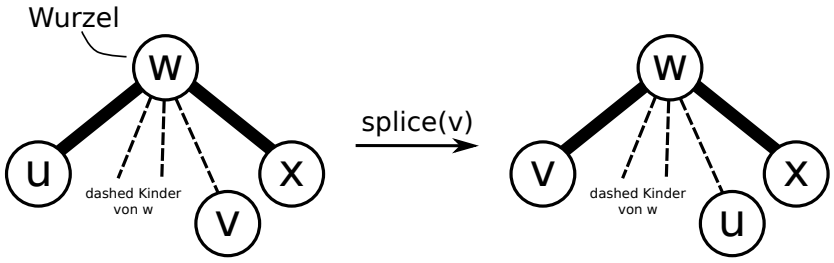


Abbildung 3.9.: Eine splice-Operation im Virtual tree. Der Knoten w ist hier die Wurzel eines soliden Baumes.

Bei dem Aufruf von $splice(v)$ wird der Knoten v zum soliden linken Kind des Knotens w. Der Knoten w soll hierbei die Wurzel eines soliden Baumes sein. Tatsächlich schränkt das unsere Aktualisierungsbetrachtung nicht ein, denn die splice-Operation wird nur auf Knoten angewendet, deren Vater die Wurzel eines soliden Baumes ist. Diese Knoten werden nämlich im ersten Schritt der access-Funktion durch einen splay-Aufruf zur Wurzel gemacht. Also sei der Knoten w die Wurzel des soliden Baumes und u sein linkes solides Kind. Durch die splice-Operation werden keine Kostenwerte verändert und nur die Knoten u und v ändern ihren soliden Vater. Da vor dem splice-Aufruf sowohl w als auch v die Wurzel ihres soliden Baumes sind, gilt $cost(w) = \delta cost(w)$ sowie $cost(v) = \delta cost(v)$. Somit folgt:

$$\begin{aligned}\delta cost'(v) &= cost(v) - cost(w) \\ &= \delta cost(v) - \delta cost(w)\end{aligned}$$

Nach der splice-Operation wird der Knoten u ein dashed Kind von w sein und deshalb auch die Wurzel eines soliden Baumes. Dann folgern wir:

$$\begin{aligned}\delta cost'(u) &= cost(u) \\ &= \delta cost(u) + cost(w) \\ &= \delta cost(u) + \delta cost(w)\end{aligned}$$

Kommen wir nun noch zur Aktualisierung der δmin Werte. Die Kostenwerte bleiben unverändert und nur der Knoten w wechselt seine Kinder. Da sich somit eventuell auch der minimale Knoten im soliden Teilbaum, der von w aufgespannt wird, ändert, betrachten wir zuletzt noch $\delta min'(w)$. Durch Lemma 3.2 folgt:

$$\begin{aligned}\delta min'(w) &= max\{0, \delta min'(v) - \delta cost'(v), \delta min'(x) - \delta cost'(x)\} \\ &= max\{0, \delta min'(v) - \delta cost(v), \delta min(x) - \delta cost(x)\}\end{aligned}$$

Wobei x das rechte solide Kind des Knotens w beschreibt, welches unverändert bleibt.
Auch diese Aktualiserungen lassen sich in konstanter Zeit realisieren und können deshalb mit der Laufzeit der splice-Operation verrechnet werden.
Wie wir diese zwei Parameter und die access-Funktion benutzen, um alle Funktionen effizient implementieren zu können, wollen wir im folgenden Abschnitt diskutieren.

3.4. Funktionen und ihre Implementation

Wir wollen nun jede Funktion, die durch einen Link/Cut tree unterstützt werden soll, betrachten und ihre Implementation besprechen. Die Funktionen wurden bereits in Abschnitt 3.2 vorgestellt. Die Struktur dieses Abschnitts ist der Arbeit von Saxena [6] zu diesem Thema entnommen.

makeTree(x): Erstellt einen neuen Link/Cut tree, der nur aus dem Knoten x besteht. Das heißt, wir erstellen einen Baum mit Wurzelknoten x und setzen $\delta cost(x) = cost(x)$ sowie $\delta min(x) = cost(x)$. Die Implementation ist trivial und die Laufzeit beträgt $\mathcal{O}(1)$.

findCost(x): Gibt den Kostenwert $cost(x)$ zurück. Der Kostenwert eines Knotens ist, wie wir im vorherigen Abschnitt gesehen haben, aber nicht direkt im Virtual tree abgespeichert. Nur für den Wurzelknoten r des Virtual trees gilt, das $\delta cost(r) = cost(r)$. Durch eine access-Operation können wir den Knoten x aber zur Wurzel des Virtual trees machen und den Kostenwert $cost(x) = \delta cost(x)$ einfach ablesen. Es folgt für die Implementation:

$$access(x), \text{ return } \delta cost(x)$$

Da nach dem access-Aufruf der Knoten x die Wurzel des Virtual trees ist, kann man den Wert $\delta cost(x)$ in $\mathcal{O}(1)$ zurückgeben. Die Gesamtlaufzeit der findCost-Operation beläuft sich also auf die Laufzeit der access-Operation.

findRoot(x): Gibt die Wurzel des Link/Cut trees zurück, indem sich der Knoten x befindet. Wir suchen hier die Wurzel des Original tree! Durch den Aufruf von $access(x)$ wird der Knoten x zur Wurzel des Virtual trees und ist somit zumindest im selben soliden Baum wie die Wurzel des Original trees. Wir finden das größte Element im soliden Baum, indem wir von x immer in den rechten Teilbaum laufen. Das größte Element des soliden Baumes wird das Element sein, dass im soliden Pfad am höchsten liegt und somit die Wurzel des Original trees ist. Da wir uns in einem Splay Tree befinden, müssen wir das gesuchte Element wiederum durch einen splay-Aufruf zur Wurzel machen. Dann gilt für die Implementation:

$$access(x), \text{ suche das größte Element im soliden Baum von } x, \text{ sagen wir } r, splay(r), \text{ return } r$$

Bei dieser Funktion setzt sich die Laufzeit aus der access-Operation und der splay-Operation zusammen. Da bei der Suche nach dem größten Element derselbe Weg genommen wird wie bei dem Splay, kann die Suche durch einen Faktor zum Splay verrechnet werden. Im zweiten Kapitel haben wir für jede Funktion der Splay trees also insbesondere der splay-Operation eine Laufzeit von amortisiert $\mathcal{O}(log(n))$ gezeigt. Die Laufzeit von $findRoot(x)$ beträgt also amortisiert $\mathcal{O}(access(x) + log(n))$.

findMin(x): Gibt den Knoten mit minimalen Kostenwert auf dem Pfad von x zur Wurzel r des Original trees zurück. Durch den Aufruf von $access(x)$ liegen alle Knoten dieses Pfades innerhalb eines soliden Baumes. Da der Knoten x im soliden Pfad der tiefste Knoten war und nach dem access-Aufruf zur Wurzel des Virtual trees wird, liegen alle Knoten des Pfades im rechten soliden Teilbaum von x. Nun nutzen wir das Lemma 3.1, um den Knoten mit minimalem Kostenwert im Pfad zu finden. Wir fangen beim rechten soliden Kind von x an, sagen wir $y = x.rightChild$ und berechnen $minCost(y.leftChild)$ sowie $minCost(y.rightChild)$. Nun hat einer der Knoten $y.leftChild$ bzw $y.rightChild$ den geringeren $minCost$-Wert. Dann suchen wir bei diesem Knoten weiter, nach dem minimalen Knoten. Wir beenden unsere Suche bei dem Knoten z, dessen Kinder beide größere $minCost$-Werte aufweisen als z selbst. Somit muss z der Knoten mit dem minimalen Kostenwert sein. Da wir die Suche auf den gewünschten Pfad eingeschränkt haben, sind wir nun fast fertig. Es bleibt noch den Knoten z in seinem soliden Baum durch einen splay-Aufruf zur Wurzel zu machen. Die Implementation ist also:

$access(x)$, suche den Knoten mit minimalem Kostenwert im rechten soliden Teilbaum von x unter zu Hilfenahme des Lemmas 3.1, sagen wir dieser Knoten heißt z, $splay(z)$, return z

Auch hier setzt sich die Laufzeit aus der access-Funktion und einer splay-Operation zusammen. Die Laufzeit der Suche kann wieder zur splay-Operation gerechnet werden, denn es wird derselbe Weg im Virtual tree abgelaufen und die einmalige Anwendung des Lemmas 3.1 ist in $\mathcal{O}(1)$. Zusammengenommen erhalten wir erneut eine Laufzeit in amortisiert $\mathcal{O}(access(x) + log(n))$ für die Funktion $findMin(x)$.

addCost(x,c): Addiert eine Konstante c auf alle Kostenwerte der Knoten des Pfades von x zur Wurzel des Original trees. Auch hier beginnen wir mit dem Aufruf access(x). Dann liegen alle gewünschten Knoten im Original tree auf einem soliden Pfad und somit im Virtual tree im rechten soliden Teilbaum von x, der nun die Wurzel des Virtual trees ist. Nun machen wir uns beim Addieren der Konstante die im vorherigen Abschnitt besprochenen δ-Werte zu Nutze. Es reicht nun, die Konstante c nur auf den Kostenwert der Wurzel zu addieren, also auf $\delta cost(x) = cost(x)$. Betrachten wir nun nämlich die soliden rechten Nachkommen von x, so stellen wir fest, dass diese ihre $\delta cost$-Werte nicht verändern. Denn für jeden soliden Nachkommen y von x gilt nun $cost(y) - cost(parent(y)) = \delta cost(y) = (cost(y) + c) - (cost(parent(y)) + c)$. Das heißt, obwohl wir auf alle $cost$-Werte eine Konstante addiert haben, bleiben trotzdem die $\delta cost$-Werte gleich. Nur bei dem Wurzelknoten x müssen wir die Konstante c addieren, denn da x keinen Vaterknoten hat, wird sich die Konstante nicht durch eine Differenz wegheben. Die Addition der Konstante propagiert sich durch die δ-Darstellung zu allen Nachkommen. Zuletzt betrachten wir noch alle linken Nachkommen des Knotens x. Auch auf die cost-Werte dieser Knoten haben wir die Konstante c indirekt addiert. Wir ziehen also die Konstante c von dem linken soliden Kind von x wieder ab. Die Implementation ergibt sich somit als:

$access(x)$, addiere c zu $\delta cost(x)$, addiere $-c$ zu $\delta cost(x.leftChild)$

Die Laufzeit setzt sich aus zwei Additionen sowie einem access-Aufruf zusammen, der die konstanten Additionen dominiert. Die Funktion $addCost(x, c)$ läuft also in $\mathcal{O}(access(x))$.

cut(x): Trennt den Knoten x von seinem Vater im Original tree. Nun bildet der Knoten x mit all seinen Nachkommen einen neuen Link/Cut tree. Zunächst machen wir dazu den Knoten x zur Wurzel des Virtual trees, indem wir $access(x)$ aufrufen. Dabei erreichen wir, dass der Knoten x und sein Vater y aus dem Original tree im selben soliden Baum liegen. Wenn wir nun die Kante von x zu seinem rechten soliden Kind trennen, haben wir im Original tree den Knoten x von seinem Vater getrennt. Wir haben nun zwei Link/Cut trees statt einem zu verwalten. Außerdem müssen wir noch die δ-Werte anpassen, denn der Knoten x verliert sein rechtes soliden Kind, sagen wir x_r, weshalb sich sein Wert δmin verändern könnte und x_r verliert seinen Vater, weshalb sich sein Wert $\delta cost$ ändert. Der minimale Knoten im Unterbaum von x ist nun der Knoten x selbst oder er befindet sich im linken Teilbaum. Es gilt $\delta min(x) = max\{0,\ \delta min(x.leftChild) - \delta cost(x.leftChild)\}$. Um die $\delta cost$-Werte im neuen Link/Cut tree, der von x_r aufgespannt wird, anzupassen, reicht es $cost(x)$ auf $\delta cost(x_\mathrm{r})$ zu addieren. Dadurch gilt $\delta cost(x_\mathrm{r}) = cost(x_\mathrm{r}) - cost(x) + cost(x) = cost(x_\mathrm{r})$. Die $\delta cost$-Werte aller Nachkommen von x_r verändern sich nicht. Als Implementation folgt:

$access(x)$, addiere $\delta cost(x) = cost(x)$ zu $\delta cost(x_\mathrm{r})$, lösche die Kante zwischen den Knoten x und x_r, aktualisiere $\delta min(x)$

Die Laufzeit setzt sich zusammen aus einem access-Aufruf sowie Konstanten, u.a. Addition und das Entfernen von Vater-Kind Zeigern. Wir erhalten also insgesamt eine Laufzeit von $\mathcal{O}(access(x))$.

link(x,y): Der Knoten x ist die Wurzel eines Link/Cut trees und der Knoten y ein beliebiger Knoten eines anderen Link/Cut tree. Dann wird der Knoten x das Kind von y. Somit sind zwei Link/Cut trees zu einem zusammengewachsen. Durch den Aufruf von $access(x)$ sowie $access(y)$ werden die Knoten x und y zu den Wurzeln ihres Virtual tree. Dann machen wir den Knoten x mit all seinen Nachkommen zum dashed Kind des Knotens y. Wir müssen weder $\delta cost$-Werte noch δmin-Werte aktualisieren, da sich die soliden Bäume nicht verändern. Zuletzt können wir noch $splice(x)$ aufrufen, um den Knoten x zum linken soliden Kind des Knotes y zu machen. Das vorherige linke Kind wird dashed. Dabei ist y die Wurzel seines soliden Baumes und wir können die Aktualisierungen der δ-Werte wie in Abschnitt 3.3.4 beschrieben, vornehmen. Die Implementation ist somit:

$access(y)$, $access(x)$, mache x zum dashed Kind von y, $splice(x)$

Die Laufzeit beträgt also $\mathcal{O}(access(y) + access(x))$, denn man kann in konstanter Zeit die zwei Vater-Kind Zeiger aktualisieren.

Wir haben nun jede Funktion kleinschrittig untersucht. Dabei haben wir erkannt, dass sich die Laufzeit stets auf die Laufzeit der access-Operation beschränkt. Im Fall der Funktion $findMin(x)$ haben wir zwar eine amortisierte Laufzeit von $\mathcal{O}(access(x) + log(n))$ jedoch werden wir nun zeigen, dass auch die Laufzeit der access-Operation in amortisiert $O(log(n))$ liegt. Deshalb können wir den Summand $\mathcal{O}(log(n))$ in diesem Fall ignorieren. Im nächsten Abschnitt diskutieren wir nun also über die Laufzeit von $access(x)$, denn diese entscheidet auch die Laufzeit aller anderen unterstützten Funktionen.

3.5. Die Laufzeitanalyse von Access()

Kommen wir nun also zur Laufzeitanalyse der access-Funktion. Wir haben diese in Abschnitt 3.3.3 in drei Schritte unterteilt und ausführlich besprochen. Genau diese drei Schritte wollen wir nun hinsichtlich ihrer Laufzeit untersuchen und zeigen, dass die Laufzeit der access-Funktion in amortisiert $\mathcal{O}(log(n))$ liegt. Dabei sei n wie immer die Anzahl der Knoten des Link/Cut trees. Für die Analyse benutzen wir erneut die Potentialfunktionmethode. Intuitiv wählen wir die Potentialfunktion ähnlich zu der aus der Analyse der splay-Funktion. Denn auch bei der Datenstruktur der Link/Cut trees wollen wir ein hohes Potential erreichen, wenn die Baumstruktur für eine weitere access-Operation ungünstig ist. Deshalb definieren wir nun folgende Parameter:

Größe von Knoten x: $size(x)$ ist die Anzahl aller Nachkommen von x inklusive x selbst. Das heißt, wir zählen sowohl die soliden als auch die dashed Kinder. Dies bedeutet unter anderem auch, dass die Gewichte jedes Knotens gleich 1 sind.

Rang von Knoten x: Wie zuvor, $rank(x) = log_2(size(x))$.

Potential Φ von Baum t: $\Phi(t) = \alpha \cdot \sum_{x \in t} rank(x)$, wobei α ein positiver, konstanter Faktor ist.

Tatsächliche Kosten einer splay-Operation: $\beta \cdot$ Anzahl der durchgeführten Rotationen. Hierbei ist β eine positive Konstante, jedoch gilt $\beta \leq \alpha$. Es sei erwähnt, dass bei der Laufzeitanalyse der splay-Operation $\beta = 1$ gesetzt war. Die Kosten einer Rotation in der splay-Operation lagen also bei einer Recheneinheit.

Durch die Verallgemeinerung des Potentials Φ und den Kosten einer Rotation durch die Parameter α und β können wir in der Laufzeitanalyse von $access()$ auftretende Konstanten verrechnen, indem wir zum Beispiel pro Rotation zwei Recheneinheiten, statt nur einer verlangen. Da wir die Definition des Potentials und der Kosten einer Rotation verändert haben, passen wir nun auch noch das Access Lemma hinsichtlich der Parameter α und β an. Das sogenannte parametrisierte Access Lemma hilft uns in der Laufzeitanalyse der access-Funktion die Laufzeit der durchgeführten splay-Operationen abzuschätzen.

Lemma 3.3 (Das parametrisierte Access Lemma)**.** *Die amortisierte Laufzeit für den Aufruf von splay(x) auf einem Splay tree t mit Wurzel r beträgt maximal*

$$\mathcal{O}\left(\beta + 3 \cdot \alpha \cdot (rank(r) - rank(x))\right) = O\left(\beta + 3 \cdot \alpha \cdot log\left(\frac{size(r)}{size(x)}\right)\right).$$

Der Beweis des parametrisierten Access Lemmas verläuft analog zum Beweis des verallgemeinerten Access Lemmas 2.7. Ein Beweis findet sich zudem in der Ausarbeitung von Saxena [6]. Es ist nur darauf zu achten, dass stets $\beta \leq \alpha$ gilt. Denn bei den Splay trees bezahlen wir teure splay-Operationen mit einer Verkleinerung unseres angesparten Potentials. Dabei ist eine teure splay-Operation durch viele Rotationen gekennzeichnet. Verlangen wir nun β Recheneinheiten für eine Rotation statt einer Recheneinheit, so vervielfacht sich die tatsächliche Laufzeit der splay-Operation um β. Dementsprechend muss auch die Potentialänderung um mindestens den Faktor β vergrößert werden, um die tatsächlichen Kosten abzufangen. Deshalb muss stets $\beta \leq \alpha$ gelten. Im Folgenden sei $\alpha = 3$. Mit Hilfe des parametrisierten Access Lemma können wir nun die Laufzeitanalyse der einzelnen Schritte der access-Funktion durchführen.

1. Schritt

Bei dem Aufruf von $access(x)$ rufen wir im ersten Schritt zunächst $splay(x)$ auf und befördern somit x zur soliden Wurzel in seinem soliden Baum. Dann ist der Knoten x über eine dashed Kante zu seinem Vater $parent(x)$ verbunden, den wir wiederum durch $splay(parent(x))$ zur soliden Wurzel seines Baumes machen und so weiter. Als Ergebnis erhalten wir, dass der Weg von x zur Wurzel des Virtual trees dashed ist. Haben wir in k soliden Bäumen einen splay-Aufruf getätigt, so wird der Weg genau k Kanten beinhalten. Dann können wir nach Lemma 3.3 über jede durchgeführte Operation der Form $splay(x)$ aussagen, dass die Laufzeit maximal $\beta + 3 \cdot \alpha \cdot (rank(r_1) - rank(x))$ beträgt. Dabei sei r_1 die Wurzel des soliden Baumes, in dem sich der Knoten x befindet. Wir wollen zunächst einsehen, warum wir das parametrisierte Access Lemma auf die soliden Bäume des Virtual trees anwenden dürfen. Zwar handelt es sich bei einem soliden Baum um einen Splay tree, jedoch haben die Knoten des soliden Baumes zusätzliche Kinder, die über dashed Kanten verbunden sind. Da diese dashed Kinder im parametrisierten Access Lemma nicht beachtet wurden, zeigen wir nun, dass die dashed Kinder die Laufzeit der splay-Operation nicht beeinflussen. Wir betrachten eine einzelne Rotation im soliden Baum, wie in Abbildung 3.10 gezeigt.

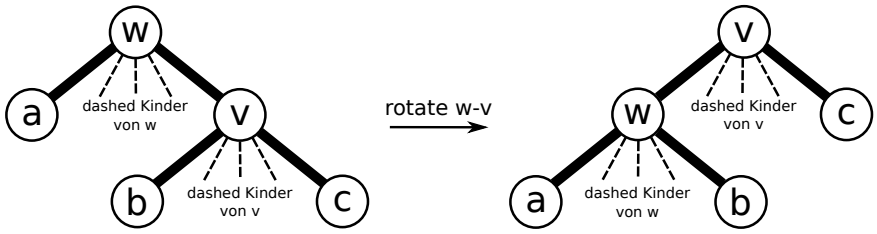


Abbildung 3.10.: Eine Einzelrotation im soliden Baum eines Virtual tree.

Da die dashed Kinder aller Knoten bei einer Rotation ihren Vater behalten, verursachen sie zunächst keine Kosten. Tatsächlich ziehen sie auch keine direkte Potentialänderung nach sich, denn die einzigen Knoten, die ihren Rang und somit das Potential des soliden Baumes ändern, sind die Knoten w und v. Jedoch ändern die Knoten w und v nur ihre soliden Nachkommen und diese Rangänderung wird im Beweis des parametrisierten Access Lemmas berücksichtigt. Auch die Vorfahren von w und v verändern ihren Rang nicht, denn sie behalten ihre Nachkommen. Nun haben die dashed Kinder also keinen Einfluss auf die Potentialänderung einer einzelnen Rotation und somit auch keinen Einfluss auf die Potentialänderung einer Sequenz von Rotationen, so wie sie bei einer splay-Operation stattfindet. Wir folgern also, dass wir das parametrisierte Access Lemma bedenkenlos auf die soliden Bäume des Virtual trees anwenden können.
Dann setzen wir den Parameter $\beta = 1$ und wie zuvor erwähnt sei $\alpha = 3$. Mit dem parametrisierten Access Lemma können wir nun die Summe der Laufzeiten aller splay-Operationen berechnen, die im 1.Schritt der access-Funktion ausgeführt werden.

$$\begin{aligned} T_{\text{am}}(1.\ Schritt) \leq\ & 1 + 9 \cdot (rank(r_1) - rank(x)) \\ & + 1 + 9 \cdot (rank(r_2) - rank(parent(r_1)) \\ & + 1 + 9 \cdot (rank(r_3) - rank(parent(r_2)) \\ & + ... \\ & + 1 + 9 \cdot (rank(r_{\text{k}}) - rank(parent(r_{\text{k-1}})) \end{aligned}$$

Dabei sind die r_i mit $i \in \{1, ..., k\}$ die Wurzeln der soliden Bäume, in denen wir die splay-Operation aufrufen. Insbesondere ist r_k die Wurzel des Virtual tree, denn der letzte solide Baum in dem wir *splay* aufrufen, ist der, der die Wurzel des Original trees enthält. Durch Umstellen einiger Summanden erhalten wir für die amortisierte Laufzeit des 1. Schrittes:

$$\begin{aligned} T_{am}(1.\ Schritt) \leq\ & 9 \cdot (rank(r_k) - rank(x)) \\ & + 9 \cdot (rank(r_1) - rank(parent(r_1)) \\ & + 9 \cdot (rank(r_2) - rank(parent(r_2)) \\ & + ... \\ & + 9 \cdot (rank(r_{k\text{-}1}) - rank(parent(r_{k\text{-}1})) \\ & + k \end{aligned}$$

Nun gilt aber für jeden Knoten y und seinen Vater $parent(y)$, dass $size(parent(y)) \geq size(y)$ gilt. Denn $size(parent(y))$ zählt alle Nachkommen von $parent(y)$, also insbesondere auch y und alle seine Nachkommen. Da $size(y)$ stets größer oder gleich *1* ist, folgt weiterhin für alle Knoten y, dass auch $log(size(parent(y))) \geq log(size(y))$ gilt. Dann gilt:

$$rank(y) - rank(parent(y)) \leq 0$$

Durch diese Erkenntnis fallen in unserer Berechnung der amortisierten Laufzeit des 1. Schrittes einige Summanden weg und es folgt:

$$T_{am}(1.\ Schritt) \leq k + 9 \cdot (rank(r_k) - rank(x))$$

Nun gilt für die Wurzel r_k des Virtual trees, dass die Größe von r_k gleich n ist. Denn zur Größe der Wurzel eines Virtual trees zählen alle n Knoten des Baumes. Daraus folgt, dass $rank\,(r_k) = log\,(size(r_k)) = n$ gilt. Da $size(x) \geq 1$ und somit $rank(x) = log(size(x)) \geq 0$ ist, können wir die amortisierte Laufzeit des 1. Schrittes folgendermaßen abschätzen:

$$T_{am}(1.\ Schritt) \leq k + 9 \cdot log(n)$$

Die Konstante k kann sehr groß werden und somit unsere gewünschte amortisierte Laufzeit von $\mathcal{O}(log(n))$ dominieren. Wir werden bei der Betrachtung der nächsten Schritte die Konstante k durch eine geeignete Wahl von β in der Gesamtlaufzeit con *access*() verrechnen können. Kommen wir zunächst zum 2. Schritt der access-Funktion.

2. Schritt

Im 2. Schritt von $access(x)$ soll der Pfad von x zur Wurzel des Original trees solide werden. Da der Pfad genau k Kanten beinhaltet, werden wir k mal die splice-Operation aufrufen. Die Laufzeit der splice-Operation ist konstant und beträgt, sagen wir, eine Recheneinheit. Somit beträgt die Laufzeit aller ausgeführten splice-Operationen $T_{tat}(k \text{ splice-Ops.}) = k$. Wir haben also erneut einen $+k$ Summanden. Auch diesen müssen wir in der Gesamtlaufzeit der access-Funktion verrechnen. Das passiert im 3. Schritt.

3. Schritt

Im letzten Schritt der access-Funktion wird nur noch die Operation $splay(x)$ durchgeführt.

Da wir zuvor sichergestellt haben, dass der Pfad von x zur Wurzel des Original trees solide ist, wird nun der Knoten x nach der splay-Operation die Wurzel seines soliden Baumes und somit des gesamten Virtual trees sein. Nun wissen wir, dass die splay-Operation genau k Rotationen durchführt. Denn der Weg von x zur Wurzel des Virtual trees beinhaltet genau k Kanten. Da wir noch den Summanden $+2 \cdot k$ aus dem 1. und 2. Schritt abarbeiten müssen, veranschlagen wir für jede durchgeführte Rotation der splay-Operation nun 3 Recheneinheiten statt einer. Wir setzen somit $\beta = 3$. Das heißt, die tatsächlichen Kosten des 3. Schrittes belaufen sich auf $T_{\text{tat}}(splay(x)) = 3 \cdot k$. Nun können wir die amortisierte Laufzeit der splay-Operation durch das parametrisierte Access Lemma 3.3 abschätzen. Mit $\alpha = 3 = \beta$ folgt: $T_{\text{am}}(splay(x)) = 3 + 9 \cdot (rank(r) - rank(x))$, wobei r die Wurzel des Virtual trees ist. Für die Wurzel r des Virtual trees gilt wie immer $size(r) = n$ und für den Knoten x beträgt die Größe mindestens 1. Nach der Definition des Ranges gilt dann $rank(r) = log(size(r)) = log(n)$, sowie $rank(x) = log(size(x)) \geq log(1) = 0$. Mit diesen Erkenntnissen schätzen wir die amortisierte Laufzeit des 3. Schrittes wie folgt ab:

$$T_{\text{am}}(3.\ Schritt) \leq 3 + 9 \cdot log(n)$$

Hierbei sind schon die beiden Summanden k aus dem 1. und 2. Schritt der access-Funktion mit einberechnet.
Insgesamt folgt für die access-Funktion eine amortisierte Laufzeit in

$$\mathcal{O}(9 \cdot log(n) + 3 + 9 \cdot log(n)) = \mathcal{O}(log(n)).$$

Und genau das wollten wir zeigen. Denn nun folgt für alle Funktionen, die durch die Link/Cut trees unterstützt werden, eine amortisierte Laufzeit in $\mathcal{O}(log(n))$. Im vorherigen Abschnitt haben wir nämlich gesehen, dass die Laufzeit jeder Funktion durch die Laufzeit der access-Funktion dominiert wird. Widmen wir uns also im nächsten Abschnitt einer Anwendung der Link/Cut trees.

3.6. Zurück zum Problem des maximalen Flusses

Da wir nun eine Laufzeit von $\mathcal{O}(log(n))$ für alle Funktionen eines Link/Cut trees gezeigt haben, kommen wir zurück zum Problem des maximalen Flusses. Wir erinnern uns, dass wir die Laufzeit des Dinic-Algorithmus mit Hilfe der Link/Cut trees verbessern wollten. Dabei verwenden wir die Link/Cut trees, um die Berechnung eines blockierenden Flusses auf dem Levelgraphen ${G_{\text{f}}}^{\text{L}}$ eines Netzwerks $G = (V, E)$ von $\mathcal{O}(|V| \cdot |E|)$ auf $O(log(|V|) \cdot |E|)$ zu verringern.
Wie zuvor in Abschnitt 3.3.1 gesehen, wollen wir uns nun bei der Berechnung des blockierenden Flusses bereits gefundene Teilpfade auf dem Weg von s nach t merken. Betrachten wir also erneut die Berechnung des blockierenden Flusses, so wie wir sie schon von Abschnitt 3.1 her kennen, nun auch hinsichtlich der Verwendung der Link/Cut trees und später anhand eines Beispiels. Die Idee, wie wir die Link/Cut trees verwenden, stammt von Wang [10] und wurde hier ausführlich ausgeführt.

1) **Auf dem Levelgraphen ${G_{\text{f}}}^{\text{L}}$:** Konstruiere das geschichtete Restnetzwerk ${G_{\text{f}}}^{\text{L}}$ mit Breitensuche, wobei wir beim Knoten s starten und den Knoten t suchen. Des Weiteren deklarieren wir s als den aktuellen Knoten u.

Link/Cut trees: Zusätzlich erstellen wir für jeden Knoten x des Levelgraphen $G_{\mathrm{f}}^{\mathrm{L}}$ einen Link/Cut tree durch die Operation $makeTree(x)$. Das heißt, wir haben zunächst viele Bäume, die nur aus einem Knoten bestehen. Diese sind noch nicht verlinkt, denn wir haben noch keine Teilpfade gefunden. Die $\delta cost$- und δmin-Werte des Knoten t setzen wir zunächst auf ∞. Die $\delta cost$- und δmin-Werte aller anderen Knoten setzen wir auf 0. Auch in der Link/Cut tree Datenstruktur verwalten wir einen aktuellen Knoten u, dieser aktuelle Knoten ist immer der Knoten am Ende des Teilpfades, den wir mit der Tiefensuche bereits gefunden haben. Er korrespondiert mit dem aktuellen Knoten aus dem Levelgraphen und ist somit auch zu Beginn als der Knoten s festgesetzt. Ausgehend von dem Knoten s starten wir nämlich in Schritt 2 die Suche nach dem Knoten t.

2) **Auf dem Levelgraphen $G_{\mathbf{f}}^{\mathbf{L}}$:** Wir erweitern den Pfad in $G_{\mathrm{f}}^{\mathrm{L}}$ ausgehend vom aktuellen Knoten u durch eine **Tiefensuche** solange, bis wir den Knoten t gefunden haben oder uns aber in einer Sackgasse befinden, wenn wir keine ausgehenden Kanten des aktuellen Knotens mehr finden. Solange also noch Kanten der Form $(u, v) \in E_{\mathrm{f}}^{\mathrm{L}}$ existieren, so fügen wir diese Kante unserem Pfad hinzu und machen v zum aktuellen Knoten, also $u = v$. Wir speichern uns den aktuellen Teilpfad stets ab. Falls nun der aktuelle Knoten $u = t$ ist, so gehe zu Schritt 4). Wenn wir in einer Sackgasse gelandet sind, gehen wir zu Schritt 3).

Link/Cut trees: Wenn wir uns in der Tiefensuche befinden und die Kante (u, v) gefunden haben, so müssen wir uns diesen Teilpfad natürlich auch in der Link/Cut tree Datenstruktur merken. Durch die Operation $link(u, v)$ können wir den Knoten u zum linken soliden Kind des Knoten v machen. Dabei gilt, dass die Knoten u und v zuvor in verschiedenen Link/Cut trees waren, denn sonst wäre die Kante (u, v) durch die Tiefensuche nicht gefunden worden. Außerdem ist der Knoten u die Wurzel seines Link/Cut trees, denn er war der aktuelle Knoten und somit das Ende des Teilpfades. Durch die Tiefensuche verlinken wir also viele Knoten und merken uns dadurch die gefundenen Teilpfade. Ziel ist es nun, den Knoten t als aktuellen Knoten und somit als Wurzel des Link/Cut tree zu erhalten, denn dann haben wir einen Pfad von s nach t gefunden. Erwähnenswert ist zudem auch, dass im Levelgraphen ein Knoten zwar mehrere ausgehende Kanten haben kann, jedoch in den Link/Cut trees nie der Fall auftritt, dass ein Knoten mehrere Väter hat. Da wir uns nämlich in der Tiefensuche befinden, arbeiten wir einen speziellen Pfad ab, bis wir in eine Sackgasse laufen und dann den Sackgassen-Knoten sowie seine eingehende Kanten, wie in Schritt 3) beschrieben, löschen. Wenn wir Kanten löschen, dann schneiden wir natürlich auch in den Link/Cut trees die korrespondierende Kanten auseinander. Erst dann können wir einem Knoten einen neuen Vater zuweisen.
Kommen wir noch zu den Kantengewichten des Levelgraphen. Diese müssen wir auch auf die Link/Cut trees übertragen. Doch Kanten in Bäumen tragen keine Gewichte. Jedoch wissen wir, dass jeder Knoten nur einen Vaterknoten haben kann, dieser ist also eindeutig. Bei einem Aufruf von $link(u, v)$ wollen wir also dem Knoten u den Kostenwert $cost(u) = c(u, v)$ zuweisen, wobei $c(u, v)$ wie gewohnt der Kapazität der Kante (u, v) entspricht. Da der Virtual tree keine $cost$-Werte abspeichert, müssen wir aber die $\delta cost$-Werte so anpassen, damit die Kantenkapazitäten richtig wiedergegeben werden. Dabei nutzen wir die Arbeit der Operation $link(u, v)$, denn diese ruft sowohl $access(u)$ als auch $access(v)$ auf. Das heißt, die Knoten u und v sind zwischenzeitlich die Wurzeln ihres Virtual trees, somit auch ihrer soliden Bäume und für diese gilt: $\delta cost = cost$. Da wir nur den Kostenwert des Knotens u auf $c(u, v)$ ändern wollen, aber nicht die Kostenwerte seiner Nachkommen, führen wir nun folgende Operationen aus:

$$addCost(u.solidChild, -(c(u, v) - cost(u))), \text{ sowie } addCost(u, c(u, v) - \delta cost(u))$$

Durch die zweite addCost-Operation wird die Konstante $c(u,v) - \delta cost(u)$ auf die $\delta cost$-Werte aller soliden Nachkommen von u addiert. Für den Knoten u gilt dann:

$$\begin{aligned} & \delta cost'(u) = \delta cost(u) + c(u,v) - cost(u) \\ \Rightarrow\ & \delta cost'(u) = c(u,v) \end{aligned}$$

Die erste addCost-Operation zieht genau diese Konstante $c(u,v) - \delta cost(u)$ zuvor von allen Nachkommen von u ab. Somit ändern sich die Kostenwerte der Nachkommen von u nicht. Außerdem ändert sich auch nur der $\delta cost$-Wert des direkten soliden Kindes von u. Es sei auch erwähnt, dass in unserer Anwendung, der Knoten u kein rechtes solides Kind haben kann, denn sonst läge dieses Kind im Original tree höher als u. Der Knoten u war aber das Ende des Teilpfades und somit die Wurzel.
Die Knoten u und v sind bis jetzt nur durch eine dashed Kante verlinkt worden. Als letzten Schritt der link-Operation wird noch $splice(u)$ ausgeführt. Somit halten wir den aktuellen Pfad stets solide.

3) **Auf dem Levelgraphen G_f^L:** Wenn wir am Ende von Schritt 2) nicht beim Knoten t angelangt sind, so sind wir in eine Sackgasse gelaufen insofern $u \neq s$. Wir löschen den aktuellen Knoten u und all seine eingehenden Kanten. Zum aktuellen Knoten machen wir den Knoten, der auf dem bereits gefundenen Teilpfad vor u lag.
War der aktuelle Knoten jedoch s, so haben wir den blockierenden Fluss von $G_f{}^L$ gefunden und sind mit diesem Levelgraphen fertig.

Link/Cut trees: Wenn wir einen Knoten x und all seine eingehenden Kanten im Levelgraphen löschen, so müssen wir auch den korrespondierenden Knoten x in den Link/Cut trees und alle Verbindungen zu seinen Kindern löschen, denn diese sind keine validen Teilpfade mehr. Sei (w,x) eine eingehende Kante des Sackgassenknotens x. Dann rufen wir auf den Link/Cut trees die Operation $cut(w)$ auf, um den Knoten w von x zu trennen. Die δ-Werte werden durch die cut-Operartion aktualisiert und müssen deshalb nicht erneut betrachtet werden. Der Knoten w hat dabei zwar noch die Kapazität der Kante (w,x) gespeichert, dies ist jedoch nicht problematisch. Würde der Knoten w einen neuen Vater erhalten, so wird der $cost$-Wert durch die $link$-Operation richtig aktualisiert.

4) **Auf dem Levelgraphen G_f^L:** Zum Schritt 4) kommen wir nur, wenn wir einen Pfad von s nach t gefunden haben. Dann suchen wir die Kante m mit der minimalen Kapazität $c(m)$ auf diesem Pfad, welche auch die bottleneck Kante genannt wird, und erhöhen unseren Fluss f um diesen Wert $c(m)$. Die Kapazität aller Kanten auf dem Pfad wird um $c(m)$ verringert und die Kanten, die jetzt eine Kapazität von *0* haben, werden gelöscht. Wir setzten $u = s$ und gehen zu Schritt 2), um weiter Pfade von s nach t zu finden, bis wir schlussendlich den blockierenden Fluss gefunden haben.

Link/Cut trees: Jetzt kommt die Datenstruktur der Link/Cut trees erst richtig zum Einsatz. Zunächst wollen wir den minimalen $cost$-Wert auf dem Pfad von s nach t bestimmen. Wir wissen dabei, dass t der Wurzelknoten seines Link/Cut trees ist. Durch den Aufruf von $access(s)$ wird s zur Wurzel des Virtual trees und alle Knoten auf dem Pfad von s zur Wurzel des Original trees liegen im rechten Teilbaum von s. Sei dabei $s.right$ das rechte solide Kind von s. Wir erinnern uns, dass wir neben den $\delta cost$-Werten auch noch einen δmin-Wert in jedem Knoten gespeichert haben. Dieser ist für einen Knoten x definiert als $\delta min(x) = cost(x) - minCost(x)$. Dabei beschreibt $minCost(x)$

den Wert des Knotens mit minimalen Kosten im soliden Teilbaum, der von x aufgespannt wird. Dann gilt für die minimale Kapazität $c(m)$ auf dem Pfad von s nach t:

$$c(m) = min\{cost(s), minCost(s.right)\}$$

Und mit Lemma 3.1 sowie dem Fakt, dass s nach der access-Operation die Wurzel des Virtual tree und somit auch die Wurzel seines soliden Baumes ist, gilt nun:

$$\begin{aligned} c(m) &= min\{cost(s), \delta cost(s.right) - \delta min(s.right) + cost(parent(s.right)\} \\ &= min\{cost(s), \delta cost(s.right) - \delta min(s.right) + \delta cost(s)\} \end{aligned}$$

Da wir nun die Kapazität der minimalen Kante $c(m)$ auf dem Pfad von s nach t gefunden haben, können wir den Fluss um $c(m)$ erhöhen. Durch die Operation $addCost(s, -c(m))$ passen wir die *cost*-Werte der Knoten auf dem Pfad von s nach t an, indem wir den Wert $c(m)$ abziehen. Das entspricht dem Verringern der Kantenkapazitäten im Levelgraph. Nun wird mindestens ein *cost*-Wert auf dem Pfad gleich *0* sein. Wir finden diesen Knoten indem wir die Funktion $findMin(s)$ aufrufen. Diese Funktion liefert den Knoten mit minimalem *cost*-Wert. Sei x dieser Knoten mit $cost(x) = 0$ und $y = parent(x)$ der Vater im Original tree. Dann können wir über die Kante (x, y) keinen Fluss mehr schicken. Durch die Operation $cut(x)$ schneiden wir den Knoten x mitsamt seiner Nachkommen vom Knoten y ab. Es könnten natürlich mehrere Kantengewichte auf *0* gefallen sein. Sobald alle Knoten x mit $cost(x) = 0$ von ihrem Vater getrennt wurden, sind wir mit diesem Schritt fertig und beginnen im Schritt 2) damit, erneut einen Pfad von s nach t zu suchen. Dabei benutzen wir die Teilpfade, die wir bereits gefunden haben.

Deutlich wird die Anwendung der Link/Cut trees durch folgende Abbildungen. Wir berechnen den blockierenden Fluss erneut für das Beispiel aus Abbildung 3.1 nun unter zu Hilfenahme der Link/Cut trees. Damit das Ganze übersichtlich bleibt, betrachten wir jedoch nur die Darstellung der Link/Cut trees als Solid/Dashed tree sowie die *cost*- und *minCost*-Werte. Es sei erneut erwähnt, dass grade die Darstellung des Original tree als Virtual tree und die $\delta-$Darstellungen der *cost*- und *minCost*-Werte die Link/Cut trees als Datenstruktur effizient machen.
Wir wollen nun also den blockierenden Fluss auf einem Levelgraphen G_f^L (vgl. Abbildung 3.11) mit Hilfe der Link/Cut trees berechnen. Beginnen wir mit dem 1. Schritt. Zu jedem Knoten x des Levelgraphen G_f^L erstellen wir einen zugehörigen Link/Cut tree. Die Kostenwerte der Knoten werden auf *0* gesetzt. Nur der Knoten t erhält den Kostenwert ∞.

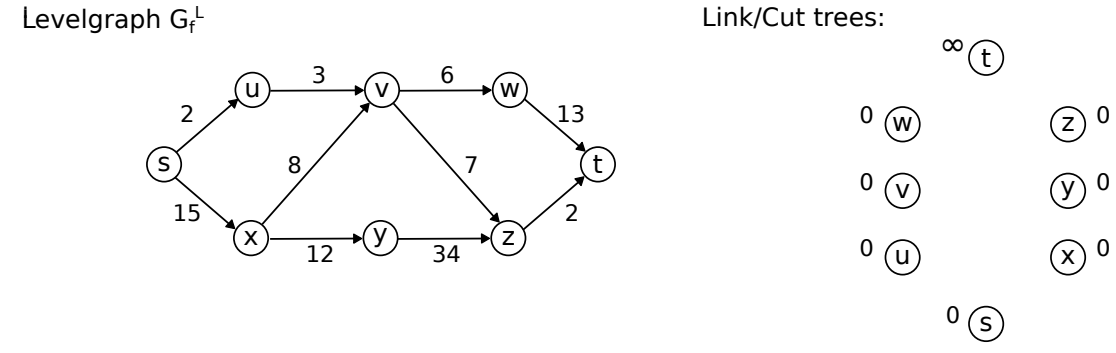


Abbildung 3.11.: Der Levelgraph G_f^L und die Erstellung der zugehörigen Link/Cut trees.

Nun suchen wir im 2. Schritt einen Pfad vom Knoten s zum Knoten t im Levelgraphen durch eine Tiefensuche. Mit jedem Level, welches die Tiefensuche durchläuft, verknüpfen wir nun unsere Link/Cut trees durch link-Operationen. So übertragen wir die Struktur des Levelgraphen auf unsere Link/Cut trees. Wie in Abbildung 3.12 gezeigt, finden wir den Pfad $s-u-v-w-t$. Da wir einen Pfad gefunden haben, gehen wir zum 4. Schritt und suchen die Kante mit minimaler Kantenkapazität, verringern die Kapazitäten auf dem Pfad um diesen Wert und löschen anschließend alle Kanten mit Kapazität $= 0$. In unserem Beispiel ist der geringste Kostenwert eines Knotens im soliden Pfad von t der Wert 2 und gehört zum Knoten s. Wir verringern also die Kapazitäten aller Knoten auf dem soliden Pfad um *2*. Dadurch verringert sich der Kostenwert von s auf *0*. Da s die Kantenkapazität der Kante $s-u$ abgespeichert hat, löschen wir also die Kante $s-u$ im Levelgraphen und trennen dementsprechend auch den Knoten s von seinem Vater in den Link/Cut trees. Die Abbildungen 3.13 sowie 3.14 zeigen die ausgeführten Operationen auf den Link/Cut trees nach dem Finden weiterer Pfade im Levelgraphen. Wir sehen insbesondere daran, dass wir weniger link-Operationen durchführen müssen, da wir gefundene Teilpfade nutzen. Diese sind schon verlinkt. Somit haben wir also den Algorithmus

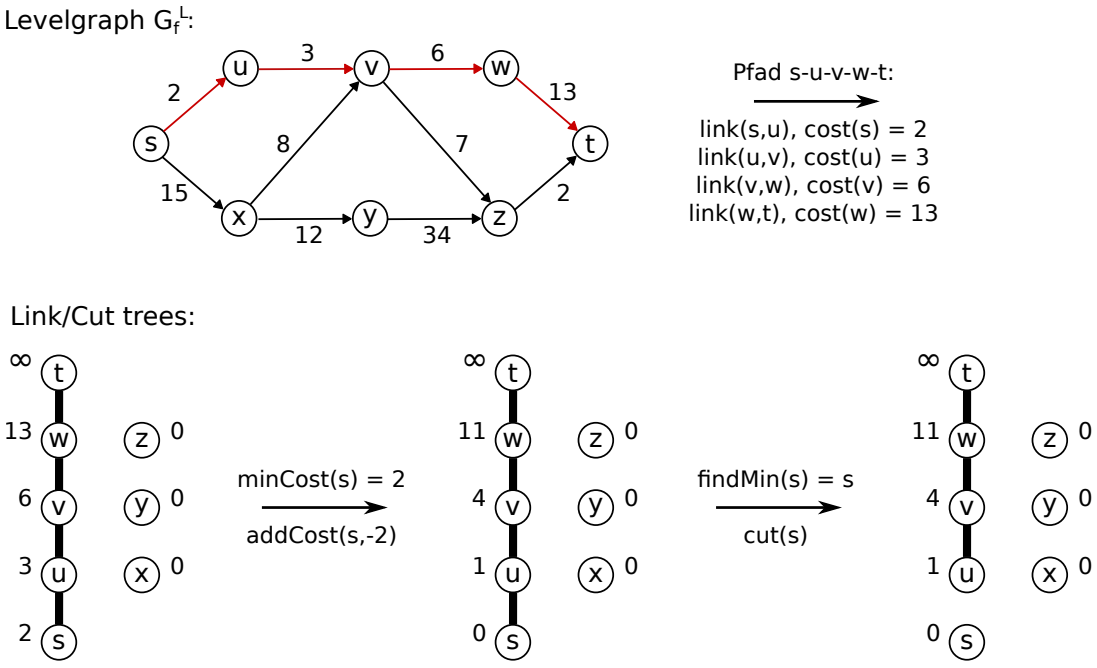


Abbildung 3.12.: Der erste gefundene Pfad und die Operationen auf den Link/Cut trees.

zur Berechnung eines blockierenden Flusses mit Hilfe der Datenstruktur der Link/Cut trees realisiert. Die Laufzeit beträgt nun amortisiert $\mathcal{O}(log(n) \cdot m)$, wobei n die Anzahl an Knoten und m die Anzahl der Kanten im Levelgraphen beschreibt. Dies gilt, da wir jede Kante nur einmal hinzufügen beziehungsweise löschen. Da wir die Link/Cut trees verwenden, ist die Laufzeit des Hinzufügens (link) und des Löschens (cut) in amortisiert $\mathcal{O}(log(n))$. Auch die Laufzeiten der anderen durchgeführten Operationen, wie das Finden der minimalen Kante auf dem Pfad von s nach t (findMin) und das Addieren einer Konstante auf diesem Pfad (addCost), sind in $\mathcal{O}(log(n))$. Die Link/Cut trees liefern uns also eine Laufzeitverbesserung gegenüber dem blockierenden Fluss-Algorithmus, den wir in Abschnitt 3.1 kennengelernt haben und verbessern

somit den Algorithmus von Dinic zur Berechnung des maximalen Flusses auf eine Laufzeit von $\mathcal{O}(|V| \cdot log(|V|) \cdot |E|)$.

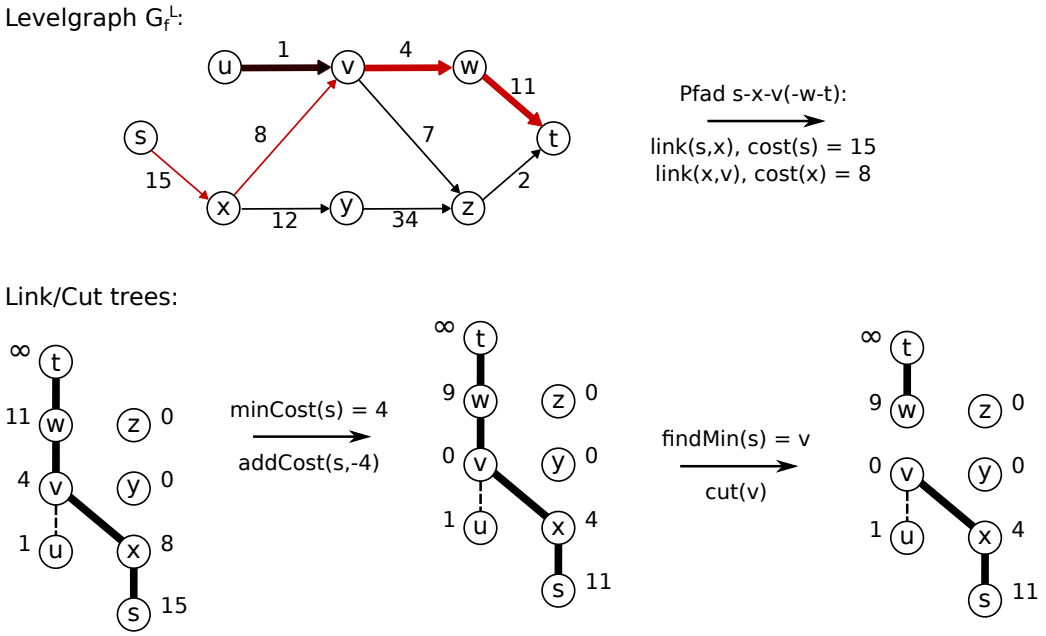


Abbildung 3.13.: Der zweite gefundene Pfad und die Operationen auf den Link/Cut trees.

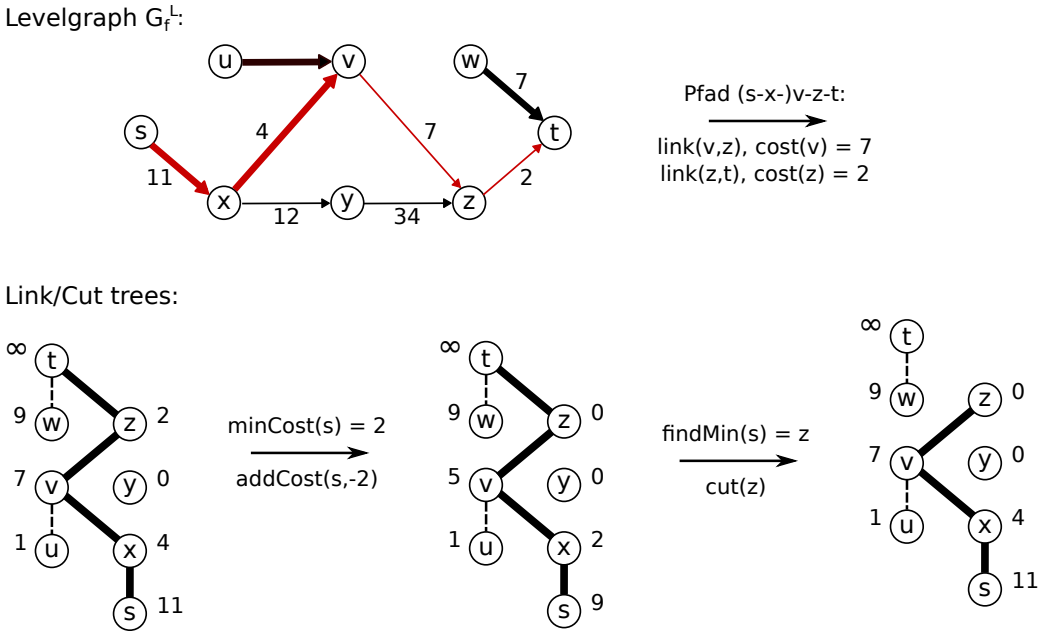


Abbildung 3.14.: Der dritte gefundene Pfad und die Operationen auf den Link/Cut trees.

4. Zusammenfassung und Ausblick

Wir haben in dieser Arbeit nun die Datenstrukturen der Splay trees und der Link/Cut trees ausführlich besprochen. Für die Splay trees haben wir gesehen, wie die Operation splay funktioniert, welche ein Element durch Rotationen zur Wurzel des binären Suchbaumes macht. Die Laufzeitanalyse durch die Potentialfunktionmethode hat ergeben, dass die Laufzeit der splay-Operation amortisiert logarithmisch in der Anzahl der Elemente des Baumes liegt. Da alle anderen unterstützten Funktionen die splay-Operation als Subroutine nutzen, liegen ihre Laufzeiten ebenfalls amortisiert in $\mathcal{O}(log(n))$, wobei n die Anzahl der Elemente des Splay trees beschreibt. Das heißt, die Splay trees sind asymptotisch gesehen genauso effizient wie andere binäre Suchbäume und im Hinblick auf die selbstorganisierende Liste sogar effizienter. Da bei jedem Zugriff auf ein Element, jenes Element durch die splay-Operation zur Wurzel befördert wird, liegen häufig angefragte Elemente höher im Baum als selten angefragte Elemente. Diese Eigenschaft haben wir genutzt, um die statische Optimalität der Splay trees zu zeigen. Ein Splay tree ist asymptotisch gesehen genauso effizient, wie ein statisch optimal aufgebauter binärer Suchbaum mit denselben Elementen. Dabei kennt der Splay tree im Gegensatz zum statisch optimalen Baum die Zugriffshäufigkeiten seiner Elemente nicht. Die Splay trees gewährleisten also eine gute amortisierte Laufzeit, auch wenn die Zugriffshäufigkeiten der gespeicherten Elemente unbekannt sind. Eine weitere Anwendung der Splay trees haben wir durch die Link/Cut trees gesehen.
Die Link/Cut trees werden unter anderem dazu genutzt, einen blockierenden Fluss auf einem Levelgraphen in $\mathcal{O}(m \cdot log(n))$ zu finden. Hierbei soll m die Anzahl der Kanten und n die Anzahl der Knoten im Levelgraphen beschreiben. Mit der effizienten Berechnung des blockierenden Flusses kann man das Problem des maximalen Flusses effizient lösen. Dabei werden gefundene Teilpfade auf dem Levelgraphen durch die Link/Cut trees verwaltet und modifiziert. Generell verwaltet ein Link/Cut tree einen Baum. Dieser wird in Pfade unterteilt, welche wiederum als Splay tree dargestellt werden, um die Laufzeiten von Zugriffen zu verkürzen. Der Baum, in dem die Pfade als Splay trees repräsentiert wurden, nannten wir Virtual tree. Bei jedem Zugriff auf ein Element wurde ähnlich wie bei den Splay trees die access-Operation aufgerufen, welche das besagte Element zur Wurzel des Virtual trees befördert hat. Diese access-Operation haben wir wiederum verwendet, um alle anderen Funktionen zu implementieren. Da die access-Operation als Subroutine die Laufzeit aller anderen Operationen einer Funktion dominiert, haben wir nur die Laufzeit der access-Operation analysiert, welche in amortisiert $\mathcal{O}(log(n))$ lag. Somit lagen auch die Laufzeiten aller anderen Funktionen in $O(log(n))$. Besonders ist zudem die Handhabung der Kostenwerte in einem Link/Cut tree. Da wir Kostenwerte auf ganzen Pfaden verändern wollten, wurde die δ-Darstellung der Kostenwerte eingeführt. Diese ermöglichte es uns unter anderem, eine Konstante auf die Kostenwerte aller Elemente auf einem Pfad (also in einem Splay tree) in $\mathcal{O}(1)$ zu addieren.
Wir schließen ab, dass die Eigenschaften der Splay trees und Link/Cut trees in vielerlei Hinsicht (auch in der Praxis) nützlich sind. Interessant sind die zwei Datenstrukturen auch für die Lehre, da durch die Behandlung dieses Themas grundlegende sowie vertiefende Strukturen der theoretischen Informatik vermittelt werden. Dies reicht von einfachen Datenstrukturen über Laufzeitanalyse mit der Potentialfunktionmethode, der Verknüpfung von Datenstrukturen bis zur Anwendung. Die Splay und Link/Cut trees liefern somit ein breites Spektrum an Lernmöglichkeiten.

Abbildungsverzeichnis

2.1. Zag case: y ist die Wurzel und x das linke Kind von y - Es wird die y-x Kante rotiert. 5
2.2. Zag-Zag case: x ist das linke Kind von y und y ist das linke Kind von z - Zuerst wird die z-y Kante rotiert und dann die y-x Kante. 6
2.3. Zig-Zag case: x ist das rechte Kind von y und y ist das linke Kind von z - Es wird zunächst die y-x Kante und dann die z-x Kante rotiert. 6
2.4. Ausführung der Operation $delete(t, x)$ 7
2.5. Ausführung der Operation $join(t_1, t_2)$ 8
2.6. Ausführung der Operation $split(t, x)$ 8
2.7. Der statische optimale Suchbaum für die gegebenen Elemente und Häufigkeiten aus Tabelle 2.1 21

3.1. Die Berechnung eines blockierenden Flusses für einen gegebenen Levengraphen. Die roten Kanten stellen den aktuell gefundenen s-t-Pfad dar, die dicken Kanten kennzeichnen Teilpfade, die zuvor bereits gefunden wurden. 27
3.2. Der Original tree (links). Ein Baum T wurde in solide Pfade zerlegt und ergibt den Solid/Dashed tree (rechts). Dicke Kanten sind hierbei soldie, gestrichelte Kanten dashed. 29
3.3. Der Virtual tree (rechts). Die soliden Pfade aus dem Solid/Dashed tree werden als binäre Suchbäume dargestellt. 30
3.4. Die splice-Operation vertauscht ein solides mit einem dashed Kind. Die Abbildung zeigt dies einmal im Solid/Dashed tree und einmal im Virtual tree 31
3.5. Der 1. Schritt von access(N). Die Vorfahren von N werden jeweils durch einen splay-Aufruf zur Wurzel ihres soliden Baumes. Der Pfad von N zur Wurzel des Virtual trees ist nun dashed. Der obige Beispielbaum ist von Erik Demaine übernommen [2]. 32
3.6. Der 2. Schritt von access(N). Die Kanten auf dem Pfad von N zur Wurzel A des Virtual trees werden durch splice-Operationen solide. Im rechten Bild werden die soliden Pfade im Solid/Dashed tree nach den splice-Operationen gezeigt. N liegt nun in einem soliden Pfad mit der Wurzel A. 33
3.7. Der 3. Schritt von access(N). Der Knoten N wird durch eine einzelne splay-Operation zur Wurzel des Virtual trees befördert. 33
3.8. Eine Einzelrotation im Virtual tree. 35
3.9. Eine splice-Operation im Virtual tree. Der Knoten w ist hier die Wurzel eines soliden Baumes. 37
3.10. Eine Einzelrotation im soliden Baum eines Virtual tree. 42
3.11. Der Levelgraph G_f^L und die Erstellung der zugehörigen Link/Cut trees. 47
3.12. Der erste gefundene Pfad und die Operationen auf den Link/Cut trees. 48
3.13. Der zweite gefundene Pfad und die Operationen auf den Link/Cut trees. 49
3.14. Der dritte gefundene Pfad und die Operationen auf den Link/Cut trees. 49

A. Literaturverzeichnis

[1] Althaus, E. : *Fortgeschrittene Algorithmen, Vorlesung 9: Splay-Trees und Link-Cut-Trees.* SoSe 2018. – Unveröffentlichte Folien der Vorlesung "Fortgeschrittene Algorithmen"

[2] Demaine, E. : *Advanced Data Structures, Lecture 19.* `https://courses.csail.mit.edu/6.851/spring12/scribe/L19.pdf`, 2012 (accessed July 7, 2018)

[3] Erickson, J. : *Algorithms, Lecture 16: Scapegoat and Splay Trees.* `http://jeffe.cs.illinois.edu/teaching/algorithms/notes/16-scapegoat-splay.pdf`, 2014 (accessed August 15, 2018)

[4] Karger, D. : *Lecture 04, 09/13: Splay Trees.* `https://www.youtube.com/watch?v=QnPl_Y6EqMo`, 2013 (accessed August 15, 2018)

[5] Mehlhorn, K. : Nearly Optimal Binary Search Trees. In: *Acta Inf.* 5 (1975), Dez., Nr. 4, 287–295. `http://dx.doi.org/10.1007/BF00264563`. – DOI 10.1007/BF00264563. – ISSN 0001–5903

[6] Mehta, D. P. ; Sahni, S. : *Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.).* Chapman & Hall/CRC, 2004. – ISBN 1584884355

[7] Sleator, D. ; Tarjan, R. : A Data Structure for Dynamic Trees. In: *Proceedings of the thirteenth annual ACM symposium on Theory of computing - STOC '81'* (1981), S. 114–122

[8] Sleator, D. ; Tarjan, R. : Self-Adjusting Binary Search Trees. In: *Journal of the Association for Computing Machinery* 32 (1985), Nr. 3, S. 652–686

[9] Turner, J. : *Advanced Data Structures and Algorithms, Lecture 19: Max Flow Problem - Dynamic Trees.* `https://www.arl.wustl.edu/~jst/cse/542/lec/lec19.pdf`, 2013 (accessed August 15, 2018)

[10] Wang, A. : *Seminar in Theoretical Computer Science: Blocking flows, Dinic's algorithm, and applications of dynamic trees.* `https://www.cc.gatech.edu/~rpeng/18434_S15/blockingFlows.pdf`, 2015 (accessed August 15, 2018)