Quadtrees

Alexander Fischer
14. August 2020

1 Einführung

In der nachfolgenden Arbeit werden Implementierungen für zwei verschiedene Arten von Quadtrees vorgestellt. Die Definitionen dafür folgen der Quelle[1]. Der erste Quadtree ist der Region Quadtree. Dieser ist in der Lage eine Region zu speichern. Ausgangspunkt dafür ist ein zweidimensionales Array mit Seitenlänge 2^k ($k \in \mathbb{N}$). Dieses ist mit Einsen und Nullen gefüllt. Die Einsen stellen hierbei die Region dar. Um nun eine Repräsentation als Baum zu erhalten, wird das Array rekursiv in vier gleich große Teile zerlegt, bis man auf der Ebene der einzelnen Elemente angelangt ist (siehe Abbildung 1). Diese 4 Quadranten bezeichnen wir mit den Himmelsrichtungen Nordost (NO), Südost (SO), Südwest (SW) und Nordwest (NW). Wenn 4 benachbarte Regionen den gleichen Wert haben, werden diese zu einem Knoten zusammengefasst. Angewendet auf Arrays mit vielen Blöcken an Nullen und Einsen, kann dies viel Platz einsparen. Ein Knoten speichert seine 4 Subquadranten und eine Farbe. Grau steht hierbei für Knoten, die keine Blätter sind. Blätter haben entweder die Farbe schwarz (in Abbildung 1 grün dargestellt) oder weiß. Schwarz steht hierbei für einen Knoten, der in der im Array definierten Region liegt. Weiß bedeutet, dass dieser Knoten nicht in der Region liegt. Ein möglicher Anwendungsfall des Region Quadtrees ist zum Beispiel die Bildkomprimierung.

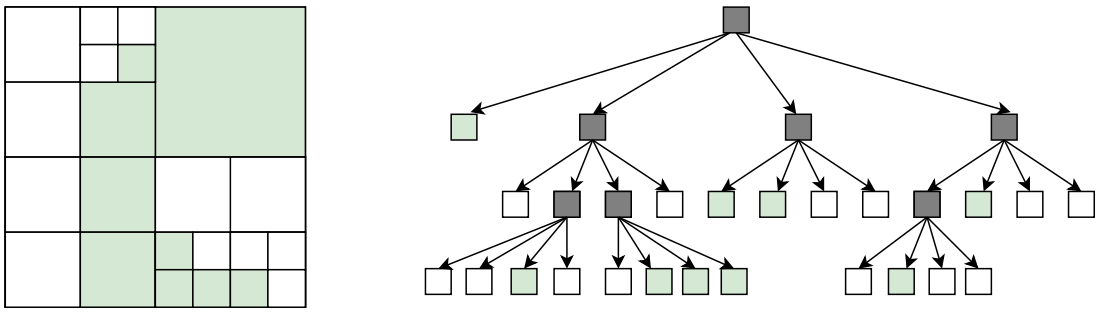

Abbildung 1: zweidimensionale Fläche mit zugehörigem Quadtree

Der Sogenannte Point Quadtree erfüllt einen anderen Zweck. Der Point Quadtree nimmt eine Liste an zweidimensionalen Punkten und speichert diese hierarchisch. Ein Punkt unterteilt eine

[1] H. Samet, „The Quadtree and Related Hierarchical Data Structures," *ACM Comput. Surv.*, Jg. 16, Nr. 2, S. 187–260, Juni 1984, ISSN: 0360-0300. DOI: 10.1145/356924.356930. Adresse: https://doi.org/10.1145/356924.356930

Fläche in 4 unterschiedlich große Rechtecke. Diese entsprechen wieder den gleichen Richtungen, wie beim Region Quadtree. Wenn nun Punkte hinzukommen, werden diese relativ ihrer Lage zum existierenden Punkt in das entsprechende Rechteck eingefügt (siehe Abbildung 2). Weiter Punkte werden ähnlich wie bei einem binären Suchbaum, passend an einen Knoten als Kind eingefügt. Ein Point Quadtree kann als Erweiterung eines binären Suchbaums auf 2 Dimensionen gesehen werden. Jeder Knoten speichert in diesen Baum Referenzen auf seine bis zu vier Kinder, seine Koordinaten und einen beliebigen Wert. Vorteil dieser Art von Speicherung ist, dass sich eine Vielzahl von Queries auf den Punkten effizient durchführen lassen. Die Suche nach einem Element braucht bei einer guten Balancierung des Baumes, gerade einmal $O(\log(N))$ Vergleiche [2], wobei N die Anzahl der Punkte ist. Auch die Suche nach allen Punkten in einem Bereich, der zum Beispiel durch ein Rechteck definiert wird, lässt sich effizient umsetzen. Das funktioniert, weil sich durch die hierarchische Struktur, viele Teilbäume bei solchen Queries ausschließen lassen.

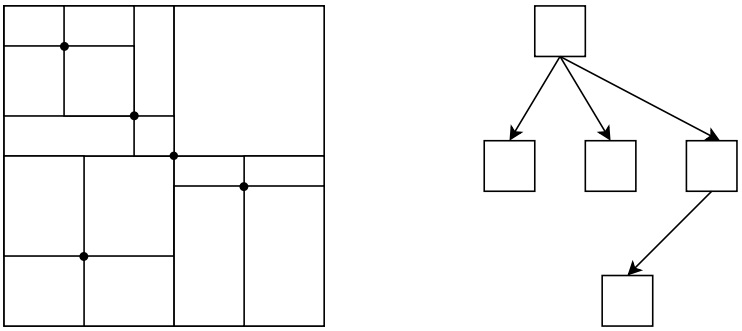

Abbildung 2: Räumliche Anordnung der Punkte mit zugehörigem Point Quadtree

2 Entwurf

In diesem Abschnitt werden wir uns spezifische Eigenschaften der Quadtrees anschauen, die eine Auswirkung auf die Implementierung haben. Die definierte Reihenfolge der Himmelsrichtungen ist Nordosten, Südosten, Südwesten und dann Nordwesten. In dieser Reihenfolge wird in allen Methoden der Implementierung über die Blätter der Knoten iteriert. Im nachfolgenden wird von einer höheren Himmelsrichtung gesprochen, wenn diese weiter hinten in der obigen Aufzählung auftritt. Beide Quadtrees sollen jeweils eine eigene Übersetzungseinheit. Die kommt daher, dass es ein paar Unterschiede zwischen den beiden Bäumen gibt. Ein Unterschied ist die Konstruktion. Bei einem Region Quadtree zerlegen wir das Array rekursiv, bis wir die maximale Tiefe erreicht haben und fassen dann Blätter eines Knotens zusammen, wenn sie die gleiche Farbe besitzen. Einen Point Quadtree konstruieren wir, indem wir den ersten Punkt einer Liste als Wurzel einfügen und die restlichen dann in Ihre jeweiligen Quadranten. Ein weiterer Unterschied ist die Iteration. Bei einem Region Quadtree sind vor allem die Blätter von Interesse. Deshalb wollen wir einen Iterator, der nur auf den Blättern iteriert. Bei einem Point Quadtree wollen wir einen Iterator, der über alle Knoten iteriert, da auch diese für Punkte stehen und relevante Informationen beinhalten. Auch strukturelle Unterschiede müssen bei der

[2] R. A. Finkel und J. L. Bentley, „Quad trees a data structure for retrieval on composite keys," *Acta Informatica*, Jg. 4, Nr. 1, S. 1–9, 1974, ISSN: 1432-0525. DOI: 10.1007/BF00288933. Adresse: https://doi.org/10.1007/BF00288933

Implementierung berücksichtigt werden. Während der Knoten eines Region Quadtrees entweder 4 Kinder besitzt oder ein Blatt ist, kann jeder Knoten eines Point Quadtrees 0 bis 4 Kinder besitzen.

Beide Klassen sollen ähnlich aufgebaut werden. Beide sollen jeweils eine Subklasse für einen Iterator und ein struct, dass einen Knoten darstellt, besitzen. Die Hauptklasse soll jeweils Informationen über das Objekt, wie die Größe verwalten und soll einen Zeiger auf die Wurzel des Baums besitzen. Über das gesamte Projekt sollen RAII-Zeiger für die Speicherverwaltung benutzt werden. Außerdem sollen beide Klassen einen Kopierkonstruktor, einen Move-Konstruktor und den Zuweisungsoperator implementieren. Dafür soll dem Copy and Swap Idiom gefolgt werden. Ein explizizer Destruktor soll nicht notwendig sein, da wir Speicher auf dem Heap mit RAII-Zeigern verwalten. In den beiden nachfolgenden Kapiteln werde ich den Entwurf der beiden einzelnen Quadtrees detaillierter besprechen.

2.1 Region Quadtree

Der Region Quadtree soll für die Konstruktion ein zweidimensionales, quadratisches Array mit Seitenlänge 2^k ($k \in \mathbb{N}$) annehmen können. Auch die Konstruktion mittels Iteratorpaar soll möglich sein. Dieser Iterator muss ein RandomAccessIterator sein. Jedes Mitglied davon soll wiederum auf eine Liste von Werten zeigen, auf die mit dem operator [] zugegriffen werden kann. Die Daten, die sowohl das Array, als auch der Iterator beinhalten, sollen konvertierbar zu boolean sein. True steht hier für einen Wert innerhalb der Region, dem Black zugewiesen wird und False für einen Wert außerhalb der Region, dem White zugewiesen wird. Die Hauptklasse soll außerdem die ursprüngliche Seitenlänge des Arrays speichern. Dies ist notwendig, um aus dem Baum das entsprechende Array zu rekonstruieren. Die Subklasse Node soll einen Farbwert und Referenzen auf 4 oder 0 Kinder speichern. Ein ForwardIterator soll angeboten werden. Dieser soll nur auf den Blättern des Baumes iterieren. Außerdem soll es eine Methode geben, die die Anzahl der Blätter zählt.

2.2 Point Quadtree

Der Point Quadtree soll für die Konstruktion einen Iterator nutzen, der eine eindimensionale Liste an Punkten mit zugehörigen Werten repräsentiert. Diese Punkte sollen mithilfe von std::pair dargestellt werden als std::pair<std::pair<x,y>, Wert> . Der Typ der Koordinaten und des Wertes, werden über Templates festgesetzt. Die Werte für Koordinaten sollen auf numerische Werte beschränkt sein. Der nordöstliche und der südwestliche Quadrant sollen ein offener Quadrant sein. Das heißt Punkte, die auf der den Quadrant definierenden Linie sind, zählen als Punkte in diesem Quadranten. Dies folgt der Defition aus [2]. Die Hauptklasse soll die Anzahl der Elemente und einen Zeiger auf die Wurzel speichern. Die Subklasse Node repräsentiert einen Knoten. Dieser speichert jeweils einen Zeiger für seine bis zu 4 Kinder, seine Koordinaten und seinen Wert. Es soll ein Forwarditerator angeboten werden, der in preorder über alle Knoten iteriert. Hierbei soll die oben definierte Reihenfolge der Himmelsrichtungen benutzt werden. Außerdem sollen ein paar Methoden angeboten werden. Für die Konstruktion und die spätere Erweiterung soll die Methode insert(Knoten) angeboten werden. Diese fügt einen Punkt an den entsprechenden Knoten im Baum ein und liefert ein std::pair<bool, Iterator> zurück. Der Boolean Wert gibt an, ob das Einfügen Erfolg hat und der Iterator zeigt auf das eingefügte Objekt (oder ist leer, wenn insert() fehlschlägt). Insert soll fehlschlagen, falls ein Punkt an derselben Stelle eines bereits im Point Quadtree existierenden Punktes eingefügt wird. Auch die Methode find(Knoten) soll angeboten werden. Diese soll nach einem Knoten im Baum su-

chen und einen Iterator auf ihn zurückliefern, wenn dieser dort enthalten ist. Ansonsten wird ein end iterator zurückgeliefert. Zusätzlich soll noch eine Methode find_in_rectangle(Rechteck) angeboten werden. Diese liefert alle im Rechteck enthaltenen Punkte. Punkte auf dem Rand des Rechtecks sollen auch dazu gezählt werden. Das Rechteck soll durch zwei Koordinatenpaare definiert werden. Das Erste soll die untere, linke Ecke und das zweite die obere, rechte Ecke darstellen

3 Umsetzung

In diesem Kapitel werden wir uns anschauen, ob und wie die Anforderungen aus Absatz 2 umgesetzt sind. Außerdem wird geschaut, wo noch Erweiterungsmöglichkeiten für die Quadtrees möglich sind. Der Anspruch war es, sich vor allem an den Übungen zu orientieren und das dort erworbene Wissen anzuwenden. Die dort vermittelten Techniken sollen sinnvoll genutzt werden, um eine gute Implementierung zu bekommen. Ich habe mich für die Nutzung von Concepts aus dem kommenden C++20 Standard entschieden. Grund dafür ist die deutlich erhöhte Lesbarkeit, im Vergleich zu SFINAE-Techniken. Fangen wir wieder mit dem Region Quadtree an.

3.1 Region Quadtree

Der Region Quadtree enthält zwei Enums, die die verschiedenen Farben und die Himmelsrichtungen repräsentieren. Er enthält ein paar Konstruktoren. Der wichtigste ist der, der einen begin und einen end random acces iterator als Parameter hat. Der Iterator zeigt hierbei auf ein Objekt, auf das mit dem Operator [ ] zugegriffen werden kann und dessen Mitglieder in einen boolean konvertierbar ist. Eingeschränkt wird dies mit Concepts wie folgt:

```
template<std::random_access_iterator Iterator>
    requires requires (Iterator it) {
        {**it} -> std::convertible_to<bool>;
        (*it)[];
}
```

Dieser Konstruktor ruft nun einen Weiteren auf, der die Wurzeln und rekursiv den Rest des Baums aufbaut. Dazu wird das Array immer weiter in 4 gleiche Teile geteilt. Sobald eine Fläche nur noch einen Wert enthält, wird ihr basierend auf diesem eine Farbe zugeteilt. Dies passiert mit:

```
colour = it_begin[y][x] ? BLACK : WHITE;
```

Hier sieht man, wofür wir die obigen Einschränkungen des Template-Parameters benötigt haben. Wenn beim rekursiven Aufstieg nun festgestellt wird, dass 4 Kinder die gleiche Farbe besitzen, werden diese nicht an deren Vater angehängt und er bekommt stattdessen deren Farbe. Ist dies nicht der Fall, werden diese an den Vater angehängt und er bekommt die Farbe GREY. Es wird außerdem ein Kopierkonstruktor, ein Move-Konstruktor und der Operator = angeboten. Die Implementierung folgt dem Copy and Swap Idiom. Wir brauchen keinen expliziten Dekonstruktor, da Speicher auf dem Heap mit RAII-Zeigern verwaltet wird. Da wir für die Knoten nicht den Vater speichern, erschwert dies die Umsetzung eines Forward-Iterators. Diesen können wir aber mithilfe eines Stacks implementieren. Der Stack speichert den Weg von

der Wurzel zum aktuellen Knoten. Er für jeden Knoten auf dem Weg einen std::weak_ptr auf den Knoten und dessen zugehörige Richtung von dessen Vater aus. Auf dem Stack liegt dann immer eine Referenz auf das dem Iterator zugehörige Objekt. Wenn wir das nächste Objekt finden wollen, gehen wir zum Vaterknoten und schauen, ob dieser noch Knoten besitzt, die eine höhere Himmelsrichtung als das der aktuelle Knoten besitzen. Wenn der aktuelle Knoten der nordwestlichste Knoten des Vaters war, wird weiter oben im Baum nach nächsten Knoten gesucht. Dazu wird das oberste Element des Stacks entfernt. Wenn der Stack leer wird, war der aktuelle Knoten der letzte im Baum und der End Iterator wird zurückgegeben. Dieser wird durch einen leeren Stack repräsentiert. Die Methode "get_array()" soll aus einem Quadtree, ein entsprechendes, zweidimensionales boolean Array konstruieren. Eine Sache, die hier schwieriger als erwartet war, war es ein zweidimensionales Array zurückzugeben. Da bei std::Array die Größe zur Compilezeit feststehen muss, war die Verwendung davon nicht möglich. Einen Rohzeiger zurückzuliefern, war keine Option, da dann die Freigabe des Speichers an den Nutzer übergeben wird. Ein Blättern im Standard hat ergeben, dass es ein für solche Fälle geschaffene std::dynarray nicht in die Sprache geschafft hat. Letztendlich wurde ein zweidimensionaler std::vector als Rückgabetyp verwendet.

Zu der Implementierung muss gesagt werden, dass diese eine recht statische Implementierung eines Region Quadtree ist, die vor allem die Funktion erfüllt, Speicherplatz zu sparen. Von sich aus unterstützt die Implementierung, keine den Baum manipulierenden Methoden. Zwar kann man den Baum mit Iteratoren manipulieren, allerdings werden Blätter nicht nach der Konstruktion zu einem neuen Blatt zusammengefasst, selbst wenn diese den gleichen Wert besitzen. Hier könnte man diese Klasse noch erweitern und das ganze zum Beispiel mit einer update Methode umsetzen, die über den gesamten Baum iteriert und schaut, ob sich Blätter zusammenfassen lassen. Auch für die Exception - Safeness und ausführlichere Tests war leider keine Zeit mehr übrig, weshalb sich im Code lediglich ein paar Assertions finden, die zum Beispiel schauen, ob das den Konstruktor aufrufende Array eine Seitenlänge von 2^k besitzt.

3.2 Point Quadtree

Der Aufbau der Klasse ist des Point Quadtrees ist dem des Region Quadtrees ähnlich. Der Point Quadtree bekommt bei der Initialisierung zwei Template Parameter, die den Typ der Koordinaten und den Typ des Wertes festlegen, der mit den Koordinaten an den Knoten gespeichert wird. Der Typ für die Koordinaten wird auf Zahlen beschränkt. Dies geschieht durch das definierte Concept arithmetical, das alle Typen, die das Prädikat std::integral<T> oder das Prädikat std::floating_point<T> erfüllen, beinhaltet. Der Point Quadtree ist dann so groß, wie es der Wertebereich des Coordtype zulässt. Diesen nicht zu überschreiten, muss vom Nutzer sichergestellt werden. Es wird ein Konstruktor für einen Forward-Iterator und ein Konstruktor für einen Random-Acces Iterator bereitgestellt. Der Typ des Iterators ist hier mittels concepts auf std::pair<std::pair<CoordType, CoordType>, ValueType> beschränkt. Dies wird erreicht durch:

```
template<std::random_access_iterator Iterator>
    requires requires (Iterator it1) {
        {*it} -> std::same_as<std::pair<std::pair<CoordType, CoordType>, ValueType>;
}
```

Die Konstruktion erfolgt dadurch, dass der erste Punkt des Iterators, als Wurzel des Point Quadtrees eingefügt wird. Alle weiteren werden dann mit der Insert - Methode eingefügt. Beim

erfolgreichen Einfügen erhöht sich ein Zähler in der Hauptklasse, der die Anzahl der Elemente speichert. Das Einfügen schlägt fehl, wenn sich der Punkt bereits im Point Quadtree befindet. Wie beim Region Quadtree werden mithilfe des Copy and Swap Idioms ein Kopierkonstruktor, ein Move-Konstruktor und der Zuweisungsoperator unterstützt. Auch hier verzichten wir auf einen expliziten Destruktor, da die Verwaltung des Speichers auf dem Heap mit RAII-Zeigern realisiert wird Der Iterator ist wie beim Region Quadtree mit einem Stack aus einem std::pair mit Referenz auf Objekt und Richtung des Vaters zum Kind umgesetzt. Ein paar Unterschiede gibt es aber, da uns beim Point Quadtree alle Knoten interessieren. Deshalb werden alle Knoten des Baums in Preorder traversiert. Ein Iterator zeigt im Point Quadtree auf eine Node. Von dieser kann nun auf deren Koordinaten und deren Wert zugegriffen werde. Ein Iterator auf einen Punkt kann mithilfe eines Stacks, der den Weg zu diesem repräsentiert, konstruiert werden. Dies sorgt in den Methoden, die einen Iterator zurückliefern (also find(), find_in_rectangle() und insert()) für zusätzlichen Code, der über die gesamten rekursiven Aufrufe einen entsprechenden Stack verwaltet, um am Ende einen Iterator konstruieren zu können. Der implementierte Point_Quadtree ist etwas flexibler als der Region Quadtree und lässt sich für die Suche nach einem Punkt oder den Punkten in einem Rechteck nutzen. Es wäre denkbar dies noch um mehr Methoden zu erweitern, die auch die Struktur dieses Baums ausnützen können. Auch wäre eine delete Methode hier denkbar, die einen Knoten im Baum findet und löscht, und den Baum danach wieder passend zusammen zu fügen. Bei diesem Quadtree hat mir leider auch die Zeit für Exception Safeness und ausführliche Tests gefehlt. Es gibt nur stichprobenartige Tests und ein paar Assertions. Auch beim Verändern der Punkte über Iteratoren werden diese leider nicht im Baum umgehängt, was diesen leider ungültig machen würde. Für die Unterbindung der Änderung von Punkten über den Iterator mittels ausschließlicher Vergabe von konstanten Referenzen blieb mir leider keine Zeit mehr.

3.3 Schluss

Abschließend muss gesagt werden, dass ich für viele Dinge bei der Implementierung länger Zeit, als dafür geplant war, benötigt habe. Dies hat leider dazu geführt, dass ich meinem Anspruch, alles über die Übung vermittelte Wissen umzusetzen, nicht ganz gerecht werden konnte. Insbesondere umfangreichere Tests und Exception Safeness, würden mein Projekt noch sehr gut ergänzen und wären noch eine gute Gelegenheit gewesen, eventuelle Fehler zu finden und zu beheben. Trotzdem denke ich, dass die beiden Klassen eine gute Basis für die beiden verwendeten Quadtrees darstellt. Die Unterstützung von Iteratoren bei der Konstruktion und das Anbieten eines eigenen Iterators, machen diese Klassen flexibel. Der Region Quadtree erfüllt seine Funktion, speichersparend eine Fläche zu repräsentieren und der Point Quadtree bietet die wichtigsten Methoden für eine Suche auf ihm an.