Interval trees

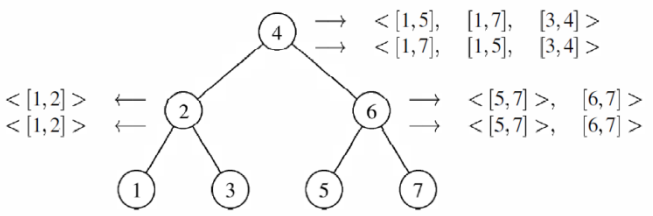
1 Einleitung

Wie Segmentbäume dienen auch diese Bäume der Speicherung von Intervallen, und unterstützen ebenfalls Aufspiessanfragen. Diese Struktur kommt jedoch mit linearen Speicherplatz aus, verglichen mit den *O(n log n)* der Segmentbäume.

Da eine Menge von n Intervallen höchstens s verschiedene Endpunkte haben kann, kann ohne Einschränkung die Menge {1,..,s} als Menge der Endpunkte verwendet werden, wenn s ≤ 2n.

2 Struktur und Operationen

Das Skelett besteht aus einem vollständigen Binärbaum mit den Schlüsseln {1,..,s}. An jedem Knoten gibt es zwei sortierte Listen *u* und *o*. In *u* sind die Intervalle aufsteigend sortiert nach den unteren Endpunkte und in *o* absteigend nach den oberen Endpunkten. Ein Intervall befindet sich nun bei nur genau einem Knoten des Baumes, nämlich beim der Wurzel am nächsten befindliche Knoten, welcher im Intervall liegt.



2.1 Einfügen und Entfernen

Für ein Intervall [a,b] geschieht das Einfügen also folgendermassen: Es geschieht rekursiv und für jeden Knoten wird überprüft, ob der Schlüssel in [a,b] liegt. Ist dies der Fall, wird das Intervall in die beiden Listen eingetragen, und andernfalls wird bei einem der Kinder weitergesucht: Ist a kleiner als der Schlüssel des gerade betrachteten Knotens, so wird links weitergemacht, sonst rechts. Werden die beiden Listen als balancierte Bäume organisiert, benötigt das Einfügen lediglich logarithmische Zeit.

Das Entfernen geschieht natürlich genau invers zum Einfügen und ist ebenfalls in logarithmischer Zeit machbar.

2.2 Aufspiessanfrage

Eine Anfrage, welche Intervalle von einem gegebenen Punkt *x* aufgespiesst werden, wird nun folgendermassen beantwortet: Man folgt dem Suchpfad im Baum zum Knoten mit Schlüssel x. Auf dem Weg dorthin wird bei jedem Knoten eine der beiden Listen inspiziert. Ist x kleiner als der Schlüssel des Knotens, wird die u-Liste durchgesehen und alle Intervalle ausgegeben, solange das linke Ende kleiner als x ist. Analog dazu wenn x grösser als der Schüssel ist: Die o-Liste wird durchgegangen und alle Intervalle ausgegeben, solange das rechte Ende grösser als x ist. Ist der Schlüssel gleich x, so wird eine der beiden Listen gewählt und alle Elemente ausgegeben.

Da das durchgehen der Listen in einer Zeit linear in den auszugebenden Elementen ist, dauert eine Aufspiessanfrage *O(log n + k)*.

3 Vergleich mit ähnlichen Datenstrukturen

3.1 Interval trees vs. segment trees

Beide Strukturen können Einfügen und Entfernen von Intervallen sowie Aufspiessanfragen in *O(log n)* bzw. *O(log n + k)* bearbeiten. Dennoch haben beide Strukturen verschiedene Vorteile. Für Segmentbäume gilt:

- Segmentbäume können in **beliebige Dimensionen** verallgemeinert werden, wobei sich ihre Laufzeit jeweils um einen logarithmischen Faktor verschlechter pro zusätzliche Dimension. Für Intervallbäume gibt es keine Verallgemeinerung, jedoch werden die beiden Datenstrukturen oftmals gemeinsam verwendet: Als Grundstruktur dienen Segmentbäume, und auf der letzen Stufe kommen dann Intervallbäume zum Einsatz, um etwas Speicher zu sparen.
- Die Liste der Intervalle bei jedem Knoten kann bei Segmentbäumen beliebig gespeichert werden, wodurch sie **flexibler** werden (was genau das hilft, ist mir jedoch nicht klar).

Intervallbäume haben auch verschiedene Vorteile, nämlich:

- Es wird **weniger Speicherplatz** benötigt, nämlich *O(n)* im Vergleich zu *O(n log n)*.

3.2 Interval and range trees

Intervallbäume sind in einem gewissen Sinne invers zu Bereichbäumen: Ist es bei ersterem möglich, Intervalle zu speichern und für einen gegebenen Punkt effizient entscheiden, welche Intervalle diesen Punkt enthalten, erlauben letztere Punkte zu speichern und für ein Intervall alle Punkte in diesem anzugeben.