ADP Aufgabe 2 Entwurf, Abgabe 1

Team 1
Hugo Protsch, Justin Hoffmann

3. Dezember 2020

Inhaltsverzeichnis

1 Formales

Aufgabenaufteilung

Der Entwurf für Insertion Sort wurde zusammen entwickelt.
Der Entwurf für Quick Sort wurde von Hugo Protsch entwickelt.
Der Entwurf für Heap Sort wurde von Justin Hoffmann entwickelt.

Quellenangaben

Es wurden lediglich Vorlesungsmaterialien verwendet.

Bearbeitungszeitraum

Der gesamte Arbeitsaufwand für den Entwurf belief sich auf ca. 16 Stunden.

Aktueller Stand

Der Entwurf steht zur Implementation zur Verfuegung.

Änderungen des Entwurfes

– nicht zutreffend –

2 Insertion Sort

2.1 Algorithmus

Siehe Abbildung 1. Bei Insertion Sort wird eine Liste durch das Einfügen von Elementen aus einem unsortierten Bereich (zunächst die komplette Liste) in einen sortieren Bereich (zunächst leer, <N1>) sortiert.

Bei dem Einfügen eines Elements E in den sortierten Bereich muss dabei jeweils der Bereich bis zu dem Element durchlaufen werden, hinter das das Element E eingefügt werden muss (Siehe »Insert into sorted list«Subgraph).

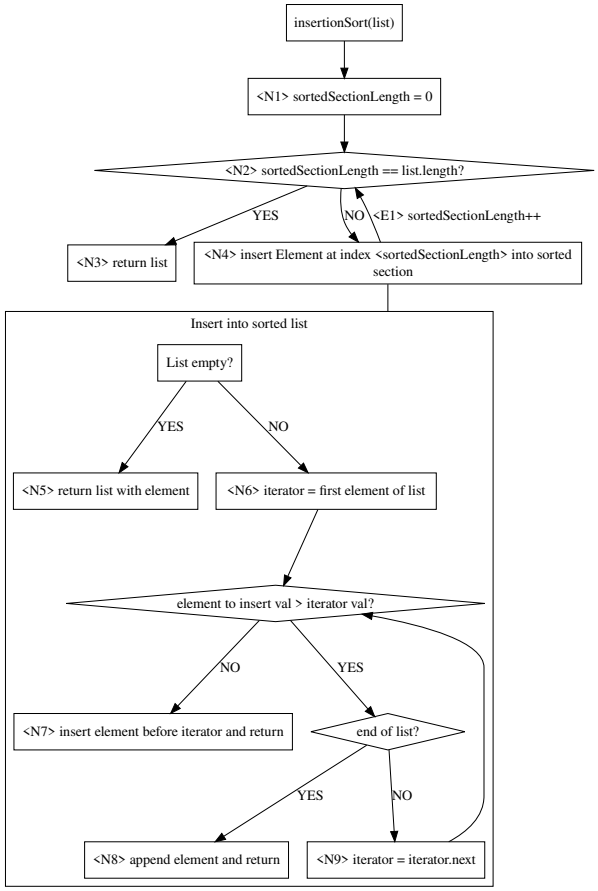
Somit wird der sortierte Bereich mit jeder Iteration um 1 erhöht <E1>. Sobald der sortierte Bereich alle Elemente enthält, wird die Liste zurückgegeben. Bei der Implementation des Algorithmus auf einfach verkettete Listen kann der sortierte und unsortierte Bereich getrennt werden.

2.2 Laufzeit

Die erwartete Laufzeit beträgt $O(n^2)$, da bei jeder Iteration das einzufügende Element mit allen noch nicht sortierten Elementen verglichen wird. Im Fall einer bereits sortierten Liste tritt der Best-Case ein. Hierbei müssen nur n Vergleiche durchgeführt werden. Somit beträgt die Laufzeit $\Theta(n)$.

Bei der Laufzeitmessung überprüfen wir, ob der gemessene Zusammenhang mit dem erwarteten übereinstimmt. Dafür verwenden wir für den Best-Case zufällige Listen, im Worst-Case bereits sortierte Listen.

Abbildung 1: Insertion Sort



3 Quick Sort

3.1 Algorithmus

Siehe Abbildung 2. Bei Quick Sort wird zunächst willkürlich ein Pivot-Element aus der Liste ausgewählt.

Anschließend wird die Liste in zwei Teile aufgeteilt: Die Liste L erhält alle Elemente, die kleiner als das Pivot-Element sind, die Liste R alle Elemente, die größer als das Pivot-Element sind. Dafür wird die Liste elementweise durchlaufen und jedes Element mit dem Pivot-Element verglichen. Die Reihenfolge der Elemente in den Listen spielt keine Rolle. Das Pivot-Element selber kommt nicht in den Listen vor.

Da alle Elemente, die kleiner als das Pivot-Element sind und alle Elemente, die größer als das Pivot-Element sind nun getrennt vorliegen, ist die Position des Pivots eindeutig als 'zwischen den Listen L und R' bestimmt. Somit kann der Algorithmus rekursiv auf die jeweiligen Listen erneut angewandt werden.

Das Pivot-Element wird an die Liste R vorangestellt. Die Liste L wird der Liste R, inklusive Pivot, vorangestellt. Die Liste ist nun sortiert.

3.2 Laufzeit

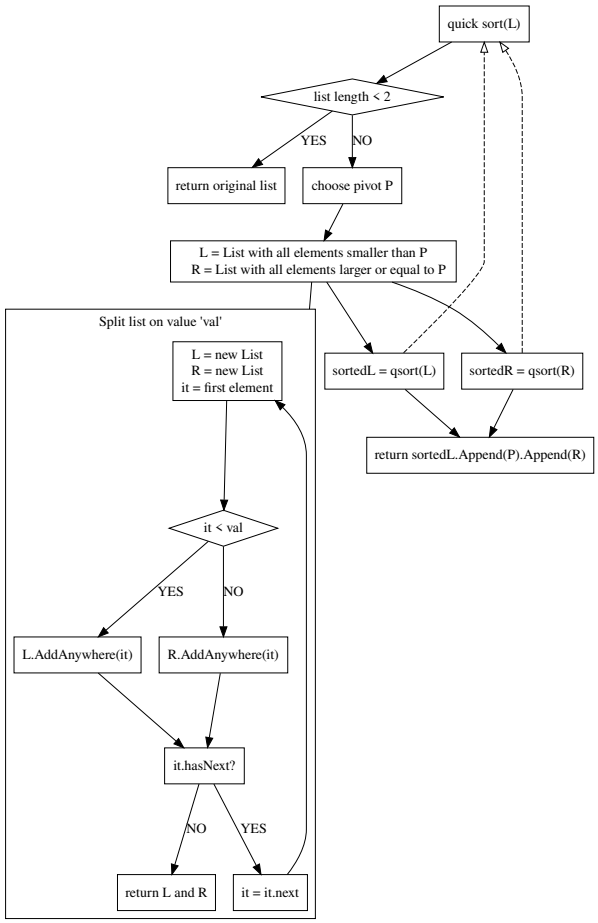
Die Laufzeit beträgt im Best-Case $\Omega(n \cdot \log(n))$, da bei gut gewähltem Pivot-Elementen die Anzahl an Vergleichen mit jedem Rekursionsschritt halbiert wird, wobei die Teillisten trotzdem durchlaufen werden müssen. Wenn das Pivot-Element bei jeder Iteration schlecht gewählt ist (kleinstes oder größtes Element in der List), beträgt die Laufzeit $O(n^2)$, da jeweils eine Partition die Größe 1 hat und die Anzahl an Vergleichen somit nicht reduziert wird.

Bei der Laufzeitmessung testen wir, ob die Komplexität des implementierten Algorithmus mit der erwarteten übereinstimmt. Um den Worst-Case zu überprüfen, nutzen wir bereits sortierte Listen und wählen das letzte bzw. erste Element als Pivot-Element. Für den Best-Case nutzten wir zufällig generierte Listen.

Wir ermitteln des Weiteren die Elementanzahl, ab der Quicksort schneller als Insertion Sort ist und wie sich die unterschiedlich Methoden, das Pivot-Element auszuwählen, auf die Laufzeit auswirken.

Außerdem wird die Implementation mit Erlang List Comprehension der Eigenen gegenübergestellt. Wir erwarten bei der List Comprehension eine leicht langsamere Laufzeit als bei der eigenen Implementierung, da bei der List Comprehension die Liste zwei Mal durchlaufen wird: Das erste Mal um

Abbildung 2: Quicksort



alle Elemente, die größer – das zweite Mal um alle Elemente, die kleiner als das Pivot-Element sind, zu filtern. Bei der eigenen Implementation wird die Liste lediglich einmal durchlaufen. Hierbei ist es interessant zu sehen, inwiefern sich die Anzahl an Durchläufen durch die Liste in der Laufzeit wieder spiegelt.

4 Heap Sort

4.1 Algorithmus

4.1.1 Allgemeiner Aufbau

Betrachte Abbildung 3. Im Allgemeinen besteht Heap Sort im Wesentlichen aus zwei Teilen: Dem Bauen eines Max-Heaps aus der zu sortierenden Liste und dem eigentlichen Sortieren mithilfe dieses Heaps. Der sogenannte Max-Heap ist eine binäre Baum-Datenstruktur, in der jeder Kindwert kleiner dem Elternwert ist, die entscheidende Eigenschaft für den Heap-Sort.

Strukturell haben wir einen "Divide and ConquerAnsatz gewählt und den Algorithmus in mehrere Subroutinen unterteilt.

4.1.2 Bauen des Max-Heaps

Der Bau-Algorithmus des Max-Heaps beginnt mit dem Einsetzen des ersten Elements der Liste als Wurzel des Baumes. Jedes nachfolgende Element wird unten eingefügt. Der Baum wird von links nach rechts aufgebaut. Ist das nun eingefügte Element größer als sein Elternknoten, werden die Knoten vertauscht, wodurch das eingefügte Element nach oben wandert. Dieser Vergleich wird solange durchgeführt, bis das Element eine passende Position im Max-Heap gefunden hat. In Abbildung 5 spiegelt sich dieser Aufsteig-Vorgang in der unteren Schleife wieder.

4.1.3 Heapify

In Heapify machen wir uns die Tatsache zunutze, dass eigentlich ein Max-Heap vorliegt und sich nur ein einziges Element an der falschen Stelle befindet. Dieses Element wird, falls nötig, zu einer passenden Position nach unten "durchsickern".

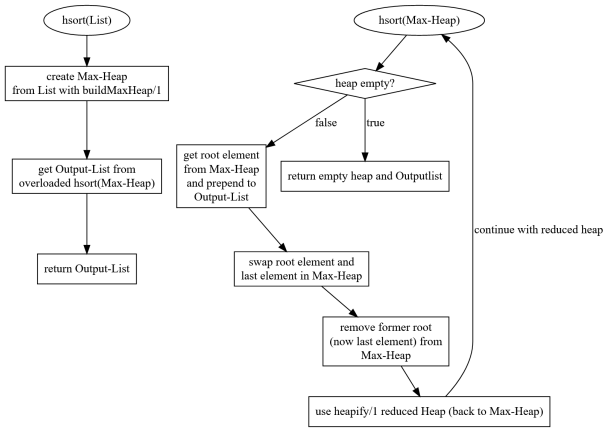
Zunächst wird überprüft, ob das Wurzelelement größer als das linke Kind ist (Aufbau von links nach rechts). Ist dies der Fall müssen die Positionen getauscht werden und das Element wandert hinab. Gibt es kein linkes Kind geschieht der gleiche Prozess analog mit dem rechten Kind. Nach dem Tauschen fragt man erneut ab und tauscht wenn nötig, bis eine der folgenden Bedingungen erfüllt ist. Existiert auch kein rechtes Kind, hat das Element eine passende Position gefunden und der Heap ist wieder ein Max-Heap. Sind beide Kinder kleiner als das Elternelement, hat das Element eine passende Position gefunden und der Heap ist wieder ein Max-Heap.

So wird der Heap für die weitere Sortierung "heapified".

4.1.4 Sortierung

Auf das Umwandeln der Liste in einen Max-Heap wurde bereits eingegangen. Der erstellte Max-Heap wird dann zum Sortieren benutzt. Wie oben bereits erwähnt ist die bestimmende Eigenschaft eines Max-Heaps der ausschlaggebende Aspekt dieses Sortier-Algorithmus. Wir fügen also das Wurzelelement des Heaps in die Output-Liste ein. Dieses Element gilt nun als sortiert (alle Elemente müssen vorne angefügt werden). Nun wird das letzte Element des Heaps mit der Wurzel getauscht und die ehemalige Wurzel entfernt. Die Struktur des Max-Heaps wird, wie oben beschrieben, durch Heapify wieder hergestellt. Dieser Vorgang wiederholt sich nachfolgend mit dem jeweils reduzierten Max-Heap, bis keine Elemente mehr im Heap vorhanden sind. Abschließend kann eine vollständige sortierte Liste zurückgegeben werden.

Abbildung 3: Heap Sort



4.2 Laufzeit

Die Laufzeit beträgt im Worst-Case $O(n \cdot log\ n)$. Dieser tritt ein, wenn eine bereits weitgehend vorsortierte Liste vorliegt, da die Liste beim Heap-Aufbau gewissermaßen invertiert wird und somit viele Vergleiche pro Element erforderlich sind. Wünschenswert wäre demnach eine invertiert-sortierte Liste, wodurch die Anzahl der Vergleiche auf 1 pro Element minimiert ist. Es gilt mit den Laufzeittests herauszufinden, ob unsere Komplexitäts–Annahmen realistisch sind. Um Heap-Sort hinreichend zu testen, binden wir den Algorithmus erweiternd in den Quick-Sort-Test mit ein und stellen beide gegenüber. Diese drei Fälle werden untersucht und verglichen:

- Vorsortierte Liste
- Invertiert sortierte Liste
- Zufällig generierte Liste

Abbildung 4: Heapify

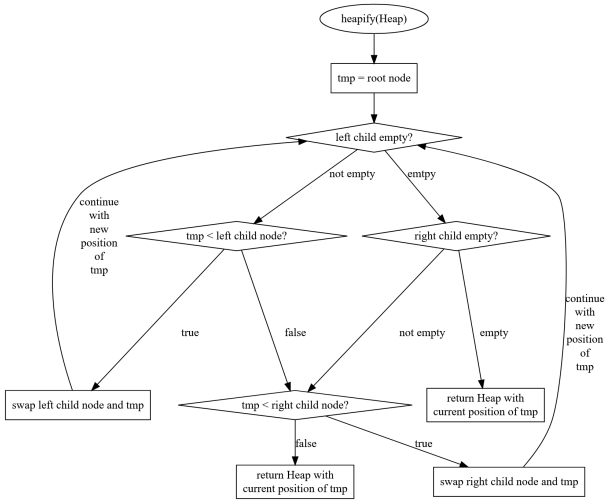


Abbildung 5: Build Max-Heap