Algorithmen und Datenstrukturen
Sommersemester 2022
Woche 4

Kevin Angele, Tobias Dick, Oskar Neuhuber,
Andrea Portscher, Monika Steidl, Laurin Wischounig

Abgabe bis 05.04.2022 23:59
Besprechung im PS am 07.04.2022

Aufgabe 1 (2 Punkte): Dynamisches Array

In der Vorlesung wurde gezeigt, dass man ein dynamisches Array vergrößern kann und dabei die Vergrößerungsstrategie so wählen kann, dass die amortisierte Laufzeit pro `add`-Aufruf konstant bleibt. Es kann jedoch auch Sinn machen, das Array unter gewissen Voraussetzungen wieder zu verkleinern um Speicher freizugeben.

Überlegen Sie sich, mit welcher Strategie Sie das Array bei einem `remove`-Aufruf verkleinern können, damit die amortisierte Laufzeit trotzdem konstant bleibt, aber sich das Array trotzdem effektiv verkleinert. Betrachten Sie dabei den schlechtesten Fall, insbesondere im Bezug auf die genaue Reihenfolge von `add`- und `remove`-Operationen. Gehen Sie davon aus, dass `add` sich bei Platzmangel die Größe des Arrays verdoppelt.

Argumentieren Sie analog zu der in der Vorlesung für `add` alleine gezeigten Methode, dass die Laufzeit mit Ihrer Strategie tatsächlich konstant bleibt.

Lösung:
Es reicht nicht, das Array um die Hälfte zu verkleinern wenn es halbleer ist, da dann ein `add`-Aufruf das Array sofort wieder vergrößern würde und diese Vergrößerung nicht ausreichend amortisiert werden könnte. Deshalb müssen wir sicherstellen, dass nach der Verkleinerung das Array noch immer nur zu einem definierten Faktor gefüllt ist. Damit können die Kosten für die Verdoppelung der Größe auf die nötigen `add`-Aufrufe aufgeteilt werden. Dieser Faktor kann z.B. mit der Hälfte gewählt werden. Um auch die Kosten für `remove` entsprechend aufteilen zu können, darf das auch nicht zu oft passieren. Daher muss auch dieser Abstand multiplikativ von der aktuellen Größe abhängen. Es kann daher z.B. immer, wenn das Array ein Viertel des Maximalfüllstandes erreicht, die Größe auf die Hälfte reduziert werden.

Aufgabe 2 (2 Punkte): Positionsbasierte Liste

Gegeben sind mehrere Operationen auf einer positionsbasierten Liste L. Geben Sie den vollständigen Inhalt der Liste nach jeder Operation an. Zu Beginn ist die Liste leer.

```
1 L = PositionalList()
2 L.addFirst(10)
3 L.addLast(1)
4 p = L.addAfter(L.addFirst(5), 8)
5 L.addAfter(L.first(), 2)
6 L.addBefore(L.after(p), 9)
7 L.remove(L.before(L.before(p)))
```

Lösung:

1. []
2. $[10]$
3. $[10, 1]$
4. $[5, 8, 10, 1]$
5. $[5, 2, 8, 10, 1]$
6. $[5, 2, 8, 9, 10, 1]$
7. $[2, 8, 9, 10, 1]$

Aufgabe 3 (2 Punkte): Iterator

Beantworten Sie die folgenden Fragen zu Iteratoren in eigenen Worten.

1. Wieso bietet sich die Verwendung eines Iterators für die Iteration über eine verkettete Liste im Gegensatz zu einer klassischen `for`-Schleife an, die durch alle möglichen Index-Werte geht? Beschreiben Sie die Laufzeitkomplexitäten beider Ansätze.
2. Sie brauchen für Ihre Anwendung einen Schnappschuss-Iterator. Leider stellt die Bibliothek, die Sie verwenden, keinen solchen zur Verfügung. Wie können Sie trotzdem den gleichen Effekt erlangen? Ist diese Umgehung mit einer höheren Laufzeitkomplexität versehen?
3. Stellen Sie sich vor, Sie haben einen faulen Iterator für ein dynamisches Array mit Verkleinerung wie in Aufgabe 1 implementiert. Was müssen Sie beim Aufruf von der `remove`-Methode des Iterators beachten?

Lösung:

1. Da die verkettete Liste keinen effizienten Zugriff über Index hat, muss sie bei einer klassischen Index-basierten `for`-Schleife in jedem Schleifendurchlauf wiederholt traversiert werden. Es ergibt sich eine Laufzeitkomplexität von $\mathcal{O}(n^2)$, statt der mit Iterator möglichen $\mathcal{O}(n)$.
2. Die Liste kann im Vorfeld einfach kopiert werden. Die Laufzeitkomplexitäten unterscheiden sich nicht, da ein Schnappschuss-Iterator immer eine Kopie anfertigen muss.
3. Der interne Zustand muss entsprechend so angepasst werden, dass das nächste Element vor dem Entfernen auch weiterhin das nächste bleibt (z.B. den internen Iterationsindex um 1 verringern). Die Verkleinerung/Vergrößerung ist für den Iterator nicht von Bedeutung, da Sie bereits vom dynamischen Array abstrahiert wird.

Aufgabe 4 (4 Punkte): Speicherverwaltung

Es kommt sehr oft vor, dass ein gemeinsam genutzter Speicherbereich auf mehrere unabhängige Programmteile dynamisch aufgeteilt werden muss. Wenn z.B. 1000 Bytes zur Verfügung stehen und 5 Funktionen gleichzeitig jeweils 150 Byte brauchen, muss irgendwie entschieden werden, wer welchen Speicherbereich verwendet. In C übernimmt z.B. die Funktion `malloc` diese Aufgabe für die Aufteilung des Heap-Speichers.

Wenn z.B. ein Speicherbereich von Adresse 0 bis 1000 zur Verfügung steht und ein Programm 50 Bytes davon anfordert, soll z.B. ein Bereich von $5-54$ zurückgegeben werden. Die Stellen $0-4$ können im Beispiel von der Speicherverwaltung intern verwendet werden, der Bereich von $55-1000$ wäre noch frei. Eine darauf folgende weitere Anfrage um 150 Bytes bekäme z.B. den Bereich von $60-210$ und der Bereich von $211-1000$ wäre als frei markiert. Wird der zweite Bereich nun freigegeben, muss der gesamte Bereich von $55-1000$ wieder als frei markiert werden.

Eine Möglichkeit, eine solche Verwaltung zu implementieren, ist mithilfe einer verketteten Liste. Dabei stellen Blöcke freien Speichers Elemente der Liste dar. Blöcke belegten Speichers sind nicht Teil der Liste, eventuell ist es aber trotzdem notwendig Zusatzinformationen außer der Nutzdaten zu speichern (z.B. die Größe). Für die Speichervergabe wird die Liste nach einem passenden Block durchsucht und eine Strategie zur Vergabe angewandt. Es muss entschieden werden, welcher Block verwendet wird und wie dieser unterteilt wird. Überschüssiger Speicher wird in einem neuen, freien Block verpackt und wieder an die richtige Stelle in die Liste eingefügt. Bei der Freigabe wird der zuvor belegte Block wieder in die Liste eingefügt und eventuell mit anderen, direkt angrenzenden freien Blöcken zusammengefasst.

1. Implementieren Sie eine Speicherverwaltung auf Basis einer verketteten Liste. Dazu steht Ihnen ein Array der Größe N zur Verfügung, das Sie anfangs selbst erstellen. Die einzelnen Plätze beinhalten Ganzzahlen. Wählen Sie die maximale Größe so, dass es für Ihre Implementierung zu keinen Problemen kommt und Sie nie einen Wert aufgrund seiner Größe auf zwei Plätze aufteilen müssen. Die Implementierung muss nicht für beliebig große N funktionieren, nehmen Sie für die Laufzeitanalyse aber an, dass sie das tut (also dass alle Ganzzahlen beliebig groß sein können). Verwenden Sie sowohl für Ihre Datenstruktur, als auch für den Speicher, den Sie vergeben, nur das anfangs erstellte Array.

   Sie müssen die Methoden der ADTen positionsbasierte Liste und Position nicht explizit implementieren, aber sollten die entsprechenden Konzepte in Ihrer Implementierung umsetzen.

   Die folgende Operationen müssen unterstützt werden:

   (a) Eine Methode `malloc(size)`, die den Startindex der Nutzdaten von einem Block, der mindestens `size` Plätze ununterbrochen beinhalten kann, zurückgibt. Nutzdaten verschiedener `malloc`-Aufrufe, die nicht freigegeben wurden, dürfen sich nicht überlappen, müssen aber nicht direkt aneinander liegen (d.h. Sie können Teile ihrer Datenstruktur zwischen den Nutzdaten der Blöcke platzieren).

   (b) Eine Methode `free(index)`, die einen Block, dessen Nutzdaten bei `index` beginnen, wieder freigibt. `index` ist dabei zwingend ein vorher von `malloc` zurückgegebener Wert. Anschließend muss der freigegebene Speicher mit `malloc` wieder neu vergeben werden können.

   Folgende Abläufe dürfen keine Fehler oder unerfüllbare `malloc`-Anfragen verursachen:

(a)
```
p = Pool(N=200)
a = p.malloc(50)
p.free(a)
b = p.malloc(190)
```

(b)
```
p = Pool(N=200)
a = p.malloc(50)
b = p.malloc(50)
p.free(a)
p.free(b)
p.malloc(190)
```

(c)
```
p = Pool(N=200)
a = p.malloc(50)
b = p.malloc(50)
c = p.malloc(50)
p.free(b)
p.free(a)
p.free(c)
p.malloc(190)
```

(d)
```
p = Pool(N=200)
a = p.malloc(50)
b = p.malloc(50)
c = p.malloc(50)
p.free(a)
p.malloc(50)
p.free(b)
p.malloc(50)
p.free(c)
p.malloc(50)
```

(e)
```
p = Pool(N=200)
a = p.malloc(50)
b = p.malloc(50)
c = p.malloc(50)
p.free(c)
p.free(b)
p.malloc(50)
p.malloc(50)
p.free(a)
p.malloc(50)
```

Hinweis: Platzieren Sie Informationen wie die Blockgröße unmittelbar vor dem von `malloc` zurückgegeben Index. So können Sie in `free` einfach „zurückschauen“, um die entsprechenden Informationen im Array zu finden.

Hinweis: Es ist keine Strategie vorgegeben. Wählen Sie eine sehr einfach zu implementierende Strategie, die die Testfälle aber trotzdem schafft.

2. Gehen Sie auf Ihre Strategie und mögliche Problemfälle ein. Fallen Ihnen mögliche Verbesserungsvorschläge Ihrer Strategie ein? Kann es z.B. sein, dass k Plätze frei sind aber ein `malloc`-Aufruf mit `size` deutlich unter k nicht erfüllt werden kann? Lässt sich das in jedem Fall verhindern, ohne die Signatur der Methoden anzupassen (also ohne einer Möglichkeit belegte Blöcke im Array zu verschieben und ohne alle Anfragen im Vorfeld zu kennen)?

3. Was ist die Laufzeitkomplexität im schlechtesten Fall in Groß-O Notation von `malloc` und `free` in Abhängigkeit von der Anzahl der Arrayplätze (nehmen Sie an, alles andere sei konstant)? Was ist sie in Abhängigkeit von der Anzahl der bereits erstellten Blöcke? Begründen Sie Ihre Antworten kurz.

Lösung:

1. Siehe `memory_management.py`.

2. Der erste Block, der gefunden wird und groß genug ist, wird zurückgegeben. Die Blockteilung erfolgt immer am Anfang des freien Blockes.

   Diese Strategie verursacht sehr viel Fragementierung, da große Blöcke von kleinen Anfragen „zerfressen“ werden, auch wenn später eventuell kleinere Lücken bereit stehen würden. Man könnte z.B. immer die ganze Liste durchsuchen und den am besten passenden Block wählen. Bei der Art, wie Blöcke geteilt werden, gibt es auch Optimierungspotential. Wenn man kleine Allokationen z.B. eher an eine angrenzende, größere Allokation als eine kleinere legt, hat man mehr Möglichkeiten durch Freigabe größere zusammenhängende Freistellen zu erzeugen.

Durch Fragmentierung ist es gut möglich, dass viele kleine freie Blöcke vorhanden sind, die nicht zusammenhängen und damit rein von der Betrachtung vom insgesamt freien Speicher her eine Allokation möglich erscheint, die es nicht ist. Allgemein lässt sich dieses Problem nicht verhindern, da man immer eine Allokations- und Freigabekette finden wird, welche die Strategie „austrickst“ und eine hohe Fragmentierung verursacht.

3. $\mathcal{O}(1)$ im Bezug auf die Speichergröße, da Datenbereiche durch die Verkettung einfach übersprungen werden. $\mathcal{O}(n)$ im Bezug auf die Anzahl der bereits vergebenen Blöcke, da diese linear durchsucht werden.