Aufgabe 1: AVL-Bäume (10 Punkte)

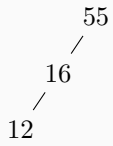
Fügen Sie die folgenden Zahlen nacheinander in einen *AVL-Baum* ein:

$$55 \quad 16 \quad 12 \quad 19 \quad 38 \quad 42$$

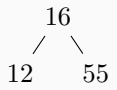
Zeichnen Sie den Baum vor und nach jeder durchgeführten Rotation. Geben Sie auch jeweils an, was für Rotationen Sie durchführen.

Lösung

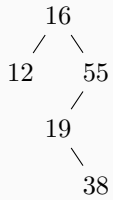
Einfügen von $55, 16, 12$



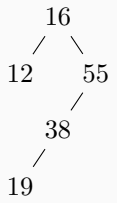
R-Rotation um 55



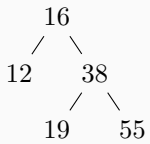
Einfügen von $19, 38$



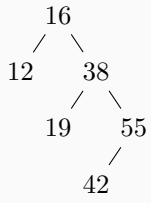
LR-Rotation um 55 (1)



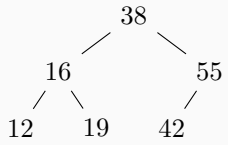
LR-Rotation um 55 (2)



Einfügen von 42



L-Rotation um 16



Aufgabe 2: Datenstrukturen (Entwurf) (10 Punkte)

Für ein Logging-System soll eine Datenstruktur entworfen werden, die es ermöglicht, eine große Menge an Strings zu speichern und von dort aus in Dateien zu schreiben. die Datenstruktur soll sozusagen als Puffer fungieren. Die Datenstruktur soll dabei Platz für eine fest Anzahl an Strings bieten, z.B. 1000. Werden die Meldungen nicht schnell genug abgearbeitet, darf die älteste Meldung gelöscht werden.
Eine einfache Lösung wäre, ein Array mit einer festen Größe zu verwenden. Dies wäre zwar leicht umsetzbar, hat aber einige Nachteile bgzl. Performance und Erweiterbarkeit. Analysieren Sie diese Lösung und machen Sie Verbesserungsvorschläge:

- Welche Performance-Probleme können sich durch den o.g. einfachen Ansatz ergeben?
- Wie können diese Probleme gelöst werden?
- Diskutieren, inwiefern Ihre Datenstruktur die Probleme löst und welche neuen Nachteile sie möglicherweise hat.

Hinweis: Sie müssen keinen Code für die Datenstruktur angeben, eine Erklärung der Idee genügt.

Lösung

Nachteile:
Der Ansatz, ein Array mit fester Größe zu verwenden, hat den Nachteil, dass beim Löschen eines Elements alle folgenden Elemente verschoben werden müssen. Hierbei wird davon ausgegangen, dass das Array vom Anfang her gefüllt wird. Ist es erstmal voll, müssen dann Elemente am Anfang gelöscht werden, um Platz für neue Elemente zu schaffen.

Lösung 1: Dynamisches Array
Ein erster Ansatz wäre, ein dynamisches Array zu verwenden, das bei Bedarf vergrößert wird. Dies würde das Problem des Verschiebens der Elemente beim Löschen lösen, weil das Array niemals voll ist. Man könnte dann einfach das erste Element markieren, das noch dazugehört, statt es zu löschen. Etwas weiter gedacht, bräuchte man dann gar keine Elemente mehr löschen, sondern könnte einfach immer weiter am Ende des Arrays anhängen. Allerdings hat dies den Nachteil, dass bei länger laufenden Systemen das Log immer weiter anwächst und irgendwann den verfügbaren Speicher übersteigt.

Lösung 2: Verkettete Liste
Bei einer verketteten Liste enthält jeder Knoten einen Wert und einen Zeiger auf den nächsten Knoten. Die Liste kann dynamisch wachsen und schrumpfen, ohne dass Elemente verschoben werden müssen. Dies löst die Probleme der Basislösung wie auch des dynamischen Arrays. Ein Nachteil ist, dass die verkettete Liste nicht zusammenhängend im Speicher liegt, was zu schlechterer Cache-Lokalität führen kann.

Lösung 3: Ringpuffer
Eine weitere Möglichkeit ist, einen Ringpuffer zu verwenden. Genau genommen ist dies sogar der Name der gesuchten/beschriebenen Datenstruktur. Ein Rungpuffer kann mittels einer verketteten Liste (wie oben) oder auch mittels eines Arrays implementiert werden. Dabei verwendet man zwei Zeiger, die den Anfang und das Ende des Puffers markieren. Beim Anhängen wird das Ende des Puffers aktualisiert und beim Löschen wird der Anfang des Puffers aktualisiert. Dies hat den Vorteil, dass die Elemente nicht verschoben werden müssen und der Puffer immer eine feste Größe behält. Die Positionen der Zeiger werden dabei zyklisch aktualisiert, d.h. wenn das Ende des Puffers erreicht ist, wird es wieder auf den Anfang gesetzt.

Aufgabe 3: Komplexität (10 Punkte)

Betrachten Sie die folgende Funktion `SameElements()`. Die Funktion erwartet zwei `int`-Listen und prüft, ob diese beiden Listen die gleiche Menge an Elementen enthalten. D.h. ob jedes Element aus der einen Liste auch in der anderen enthalten ist. Dabei spielt es keine Rolle, ob die Länge der Listen gleich ist bzw. ob die Elemente in der gleichen Anzahl vorkommen.

```
1  bool same_elements(std::vector<int> list1, std::vector<int> list2) {
2    for (int el : list1) {
3      bool contained = false;
4      for (int el2 : list2) {
5        if (el == el2) {
6          contained = true;
7        }
8      }
9      if (!contained) {
10       return false;
11     }
12   }
13
14   for (int el : list2) {
15     bool contained = false;
16     for (int el2 : list1) {
17       if (el == el2) {
18         contained = true;
19       }
20     }
21     if (!contained) {
22       return false;
23     }
24   }
25   return true;
26 }
```

a) Bestimmen Sie die Komplexität dieser Funktion. Geben Sie in O-Notation an, wie oft die Vergleiche `if v1 == v2` durchgeführt werden. Begründen Sie Ihr Ergebnis.

b) Machen Sie einen Vorschlag, wie diese Funktion optimiert werden kann. Erläutern Sie, inwiefern dieser eine bessere Komplexität hat.

Hinweis: Sie müssen keinen konkreten, vollständigen Algorithmus angeben.

Lösung

a) Die Funktion liegt in $O(n \cdot m)$, wobei n und m die Längen der beiden Listen sind. Es gibt zwei geschachtelte Schleifen, wo jeweils innerhalb eines Durchlaufs durch eine Liste für jedes Element ein Durchlauf durch die andere Liste passiert. In jeder dieser Schleifen werden daher $n \cdot m$ Vergleiche durchgeführt.

b) Eine Optimierungsmöglichkeit wäre, die beiden Listen zu sortieren, Duplikate zu entfernen und die Listen dann auf Gleichheit zu testen. Das Sortieren liegt in $O(n \cdot \log n + m \cdot \log m)$, Duplikate entfernen und der exakte Vergleich von Listen haben jeweils lineare Laufzeit. Da diese Aktionen hintereinander geschehen und nicht geschachtelt werden, bleibt es bei der Komplexität des Sortierens, also insgesamt bei $O(n \cdot \log n)$, wobei n die Länge der größeren Liste ist.

Aufgabe 4: Datenstrukturen (Funktionsweise) (10 Punkte)

Erläutern Sie die Idee und Funktionsweise des folgenden Datentyps. Diskutieren Sie kurz die Vor- und Nachteile gegenüber ähnlichen Datentypen aus der Vorlesung.

```
1  struct FooType {
2    std::vector<int> values;
3    std::vector<FooType *> children;
4
5    void AddToChild(int i, int value) {
6      while (children.size() <= i) {
7        children.push_back(new FooType());
8      }
9      children[i]->Add(value);
10   }
11
12   void Add(int value) {
13     if (values.size() < 2) {
14       values.push_back(value);
15       return;
16     }
17     if (value < values[0]) {
18       AddToChild(0, value);
19       return;
20     }
21     if (value < values[1]) {
22       AddToChild(1, value);
23       return;
24     }
25     AddToChild(2, value);
26   }
27 };
```

Lösung

Der Datentyp `Footype` ist ein *ternärer Suchbaum*. In jedem Knoten des Baums werden bis zu zwei Werte gespeichert und es gibt bis zu drei Kindknoten. Die Werte im linken Kindknoten sind kleiner als der erste Wert im Elternknoten, die Werte im mittleren Kindknoten sind größer als der erste Wert im Elternknoten und kleiner als der zweite Wert im Elternknoten, und die Werte im rechten Kindknoten sind größer als der zweite Wert im Elternknoten.

Die Funktion `Add` fügt einen Wert in den Baum ein. Dazu benutzt sie die Funktion `AddToChild`, die einen Wert in einen Kindknoten einfügt und diesen ggf. vorher erzeugt. Die Funktion `AddToChild` ruft dann wieder `Add` auf, um den Wert in den Kindknoten einzufügen.

Die Datenstruktur ist sehr ähnlich zu einem binären Suchbaum, allerdings wird der Baum breiter und flacher, da jeder Knoten bis zu zwei Werte speichert. Dadurch ergibt sich ein minimal besseres Laufzeitverhalten.

Die Komplexitätsklasse der Operationen ist die gleiche wie bei einem binären Suchbaum. Der Nachteil ist, dass die Datenstruktur wesentlich unübersichtlicher ist. Sobald bspw. eine Funktion zum Löschen hinzu kommt, kann es passieren, dass ein Knoten nur noch einen Wert enthält, dieser aber nicht der linke der beiden Werte ist. Dann muss der Baum umorganisiert werden oder beim Einfügen muss darauf geachtet werden, dass auch Werte ungültig sein können.

Aufgabe 5: Algorithmen (Entwurf) (10 Punkte)

Gegeben eine Liste von Zahlen, die ausschließlich die Werte $0, 1$ und 2 enthalten, entwerfen Sie einen Algoritzmus, der diese Liste aufsteigend sortiert.

Anmerkung: Ihre Lösung wird sowohl anhand der Korrektheit als auch der Laufzeitkomplexität bewertet. Eine einfache Lösung, bei der einer der aus der Vorlesung bekannten Sortieralgorithmen angewendet wird, oder bei der auf einen Bibliotheksalgorithmus (z.B. `std::sort`) zurückgegriffen wird, wird nicht akzeptiert.

Hinweis: Sie müssen keinen Code für den Algorithmus angeben, eine Erklärung der Idee genügt. Eine optimale Lösung sollte in $\mathcal{O}(n)$ Zeit laufen. Genauer ist es sogar möglich, die Liste mit einem einzigen Durchlauf zu sortieren.

Lösung

Lösung: Neues Sortierverfahren
Eine mögliche Lösung ist, ein Sortierverfahren zu beschreiben, das in der Vorlesung nicht behandelt wurde. Die Komplexität wäre dabei je nach Verfahren $O(n \log n)$ oder größer. Derartige Lösungen werden hier nicht weiter behandelt, würden aber (für Teilpunkte) akzeptiert werden. Volle Punktzahl kann man damit nicht erreichen, da die Komplexität nicht optimal ist.

Bessere Lösung: Vorkommen zählen
Eine einfache Lösung in $O(n)$ wäre, die Vorkommen der Zahlen 0, 1 und 2 zu zählen. Dies funktioniert generell, solange der Wertebereich der Liste vorab bekannt ist. Es wird dafür ein Hilfsarray mit sovielen Elementen benötigt, wie der Wertebereich hat. Dieses kann in einem ersten Durchlauf befüllt und anschließend in einem zweiten Durchlauf in der richtigen Reihenfolge ausgelesen werden.
Bei der vorgegebenen Aufgabe mit drei Elementen ist dieser Wertebereich konstant und sehr klein. Generell ist die Qualität dieser Lösung aber stark von der Größe des Wertebereichs abhängig. Bei einem vorab unbekannten Wertebereich muss dieser erst ermittelt werden und die Größe des Hilfsarrays ist dann ausschlaggebend für die Laufzeitkomplexität.

Optimale Lösung:
Eine optimale Lösung in $O(n)$ ist, die Liste in einem einzigen Durchlauf zu sortieren. Dazu werden drei Zeiger benötigt:

- `low` : Alles links von diesem Zeiger ist 0.
- `mid` : Dieser Zeiger wandert durch die Liste.
- `high` : Alles rechts von diesem Zeiger ist 2.

Die Idee ist, die Liste von links nach rechts zu durchlaufen und je nach Wert des aktuellen Elements (`mid`) Elemente zu vertauschen und die Zeiger zu aktualisieren.

- Ist `mid` gleich 0, wird es mit `low` vertauscht. `low` und `mid` werden um eins erhöht.
- Ist `mid` gleich 1, wird `mid` erhöht.
- Ist `mid` gleich 2, wird es mit `high` vertauscht. `high` wird um eins verringert, aber `mid` bleibt unverändert.

Anmerkung: Dieser Algorithmus ist auch als *Dutch National Flag Problem* bekannt. Der Name stammt von einem Problem, das in den 1970er Jahren von Edsger W. Dijkstra formuliert wurde.