

Datenstrukturen und Algorithmen

Vorlesung 14

Überblick

- Vorige Woche:

- AVL Bäume
- Huffman Codierung
- Skip Listen
- Postfixnotation

- Heute betrachten wir:

- Zusammenfassung
- Schriftliche Prüfung
- Fragen und Antworten

Aufgaben – SLL

Wenn man den Listenkopf eines SLLs kennt, bestimme ob der Endknoten der Liste NIL in dem *next* Feld enthält, oder **ob die Liste einen Zyklus enthält** (der letzte Knoten enthält in dem *next* Feld die Adresse eines anderen Knotens).

- I. Methode – Man speichert die Adressen der Knoten in einer Hashtabelle. Falls es Duplikate gibt, dann gibt es einen Zyklus.
- II. Methode – Man kann einen Flag in dem Knoten speichern, um zu wissen ob der Knoten besucht wurde oder nicht.
- III. Methode - Floyd's Algorithmus: Man benutzt zwei Hilfsvariablen *node1* und *node2*, die mit dem Head initialisiert werden. Bei jedem Schritt geht *node1* je ein Knoten weiter und *node2* je zwei Knoten weiter. Falls *node1* und *node2* gleich werden, dann gibt es einen Zyklus.

Aufgaben – SLL

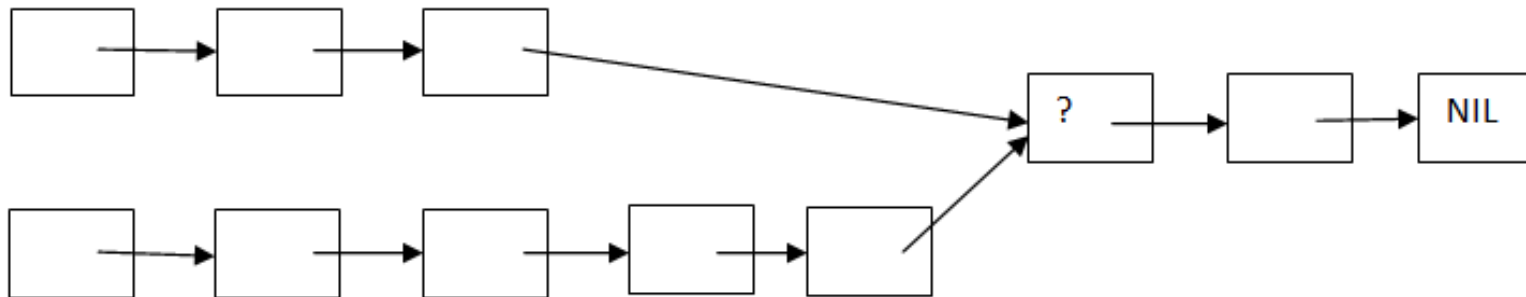
Falls die Liste in dem vorigen Problem einen Zyklus enthält, bestimme die **Länge des Zyklus**.

- I. Methode – benutze Floyd's Algorithmus um den Zyklus zu finden. Dann kann man mit einer anderen Variable den Zyklus bis zu dem gefundenen Knoten durchlaufen, um die Länge zu berechnen
- II. Methode – Recherchiere Brent's Algorithmus (ähnelt dem Floyd Algorithmus) – berechnet auch die Länge des Zyklus

Schnittpunkt zweier verketteten Listen

- **Schnittpunkt zweier verketteten Listen:**

- Man nimmt an, dass es zwei einfach verkettete **Listen** gibt, **die sich in einem Punkt treffen** und von da an gemeinsame Knoten haben. Die Anzahl der Knoten aus der zwei Listen vor dem gemeinsamen Teil ist nicht bekannt. Schreibe einen Algorithmus, der den ersten gemeinsamen Knoten findet (Tipp – benutze ein Stack)



Schnittpunkt zweier verketteten Listen

- Methode I: für jeden Knoten aus einer Liste überprüft man ob sich der Knoten in der anderen Liste befindet
 - Komplexität: $O(m \cdot n)$
- Methode II: man benutzt einen Flag um zu wissen ob ein Knoten schon besucht wurde. Nachdem die erste Liste ganz durchlaufen wird, beginnt man die zweite Liste zu durchlaufen. Falls ein Knoten schon besucht wurde, dann ist dieser der Schnittpunkt.
 - Komplexität: $O(m+n)$
- Methode III: man bestimmt die Anzahl der Elemente der zwei Listen $n1$ und $n2$. Man berechnet die positive Differenz $d = |n1 - n2|$ und man durchläuft die ersten d Elemente der längeren Liste. Nachher kann man die zwei Listen im parallel durchlaufen und die Knoten vergleichen um den Schnittpunkt zu finden.
 - Komplexität: $O(m+n)$
- **Denke an andere mögliche Lösungen und deren Komplexitäten**

Binärbaum aus den Präordnung- und Inordnungstraversierungen aufbauen

- Wenn es einen Binärbaum gibt, aber man kennt nichts darüber außer der Traversierungen in Präordnung und Inordnung
- Kann man den Binärbaum mithilfe der zwei Traversierungs-Methoden aufbauen?
- Zum Beispiel:
 - Präordnung: A B F G H E L M
 - Inordnung: B G F H A L E M

Binärbaum aus den Präordnung- und Inordnungstraversierungen aufbauen – Methode I

- Wir haben zwei Indexe mit denen wir arbeiten:
 - präIndex – Index um die Präordnung Traversierung zu durchlaufen
 - inIndex – Index eines Elementes in der Inordnung Traversierung
- Idee für einen rekursiven Algorithmus *buildTree*:
 1. Wähle ein Element aus der Präordnung Traversierung aus und inkrementiere den *präIndex*
 2. Erstelle einen neuen Knoten *node* mit dem ausgewählten Wert
 3. Finde den *inIndex* (Index in der Inordnung Traversierung) für den ausgewählten Wert
 4. Rufe *buildTree* für die Elementen **vor** dem *inIndex* auf und setze das Ergebnis als **das linke Kind** von *node*
 5. Rufe *buildTree* für die Elementen **nach** dem *inIndex* auf und setze das Ergebnis als **das rechte Kind** von *node*
 6. Gebe den Knoten *node* zurück

Binärbaum aus den Präordnung- und Inordnungstraversierungen aufbauen – Methode II

- Wenn bei jeder Suchoperation alle Elemente durchlaufen werden, dann ist die Komplexität von *buildTree* $O(n^2)$
- Wie kann das optimiert werden?
- Die Suche des Indexes in der Inordnung Traversierung:
 - Man kann eine Hilfsstruktur (z.B. ein Map) benutzen damit die Suche eine Zeitkomplexität von $\Theta(1)$ hat
 - Dann ist die Komplexität von *buildTree* $O(n)$

Binärbaum aus den Präordnung- und Inordnungstraversierungen aufbauen – Methode III


- Idee: in der Präordnung Traversierung ist der erste Knoten immer die Wurzel
- Benutze zwei Datenstrukturen:
 - ein Stack, um den Pfad zu speichern, der schon besucht wurde
 - ein Set, um der Knoten zu speichern, wo der nächste rechte Teilbaum erwartet wird

Zusammenfassung

- Während des Semesters haben wir die wichtigsten Container (ADT) und ihre wichtigsten Eigenschaften und Operationen besprochen: Bag, Set, Map, MultiMap, List und die sortierte Versionen, bzw. Stack, Queue, PriorityQueue
- Wir haben die wichtigsten Datenstrukturen besprochen, die zur Implementierung dieser Container verwendet werden können: Dynamisches Array, Verkettete Listen, Heap, Hashtabelle, Binärsuchbaum

Zusammenfassung

- Wenn ihr Container in verschiedenen Programmiersprachen verwendet, solltet ihr eine Vorstellung davon haben, wie sie implementiert sind und welche Komplexität die Operationen haben
- Ihr solltet in der Lage sein, den am besten geeigneten Container zu identifizieren, um ein bestimmtes Problem zu lösen



***If the only tool you
have is a hammer, to
treat everything as if it
were a nail***

Maslow, 1962

Schriftliche Prüfung - Struktur

- Aufgaben mit kurzen Antworten
- Einfügen bzw. Löschen von Werten in BST, AVL-Baum, Hashtabelle, usw.
- Verschmelzen (merging) zweier Binomial-Heaps
- "Was passiert, wenn man [...]?"
- Komplexität-Aufgaben
 - Berechne (vollständig!!!!) die Komplexität für eine gegebene Implementierung
 - Schreibe einen Algorithmus mit einer gegebenen Komplexität oder einen effizienten Algorithmus – denke an eine passende Repräsentierung
- Implementierungsaufgabe (benutze oder definiere eine Datenstruktur / ein Container und implementiere bestimmte Operationen)
 - Für einen bestimmten Container wähle die beste Repräsentierung, so dass die Operation X die Komplexität Y / minimale Komplexität hat und implementiere diese Operation
 - Implementiere eine bestimmte Operation für den gegebenen Container mit einer gegebenen Repräsentierung

Schriftliche Prüfung

- Insgesamt 8-9 Aufgaben
- Alle Antworten brauchen auch Erklärungen!
- Die Prüfung ist „closed book“, ihr dürft keine Ressourcen benutzen
- Versucht die Datenstrukturen und Algorithmen zu verstehen und nicht auswendig zu lernen!

Schriftliche Prüfung - komplexe Aufgabe - Beispiel

Wir wollen **ADT Quartiler** implementieren, wobei der ADT ganze Zahlen enthält und folgende Operationen mit den gegebenen Komplexitäten hat.

- `init(q)` - erstellt einen neuen, leeren Quartiler - $\Theta(1)$ - allgemeine Komplexität
- `add(q, elem)` - fügt ein neues Element in dem Quartiler `q` ein - $O(\log_2 n)$ - amortisierte Komplexität
- `get75Quartile(q)` - gibt das Element zurück, das am nächsten an dem 75sten Percentile ist. Falls es kein solches Element gibt, dann wird eine Ausnahme geworfen (throw Exception) - $\Theta(1)$ - allgemeine Komplexität
- `delete75Quartile(q)` - löscht das Element, das am nächsten an dem 75sten Percentile ist. Falls es kein solches Element gibt, dann wird eine Ausnahme geworfen (throw Exception) - $O(\log_2 n)$ - allgemeine Komplexität

Erläuterung. 75ste Percentile (oder 3rd quartile) einer Sequenz ist ein Wert aus der Sequenz mit folgender Eigenschaft: wenn die Sequenz sortiert wird, dann sind 75% der Werte kleiner als diesem Wert. Also, für die Werte 1 bis 100 (in jedwelcher Reihenfolge), ist die 75ste Percentile der Wert 75. Für die Werte 1,2,5,40, ist die 75ste Percentile der Wert 5. Im Falle eines Unentschiedens, zum Beispiel, für die Werte 1,2,3,4,5,6 können die Werte 4 oder 5 zurückgegeben werden.

1. Welche Datenstruktur oder Kombination von Datenstrukturen (aus denen, die in der Vorlesung besprochen wurden) würdet ihr benutzen um Quartiler zu repräsentieren?
2. Beschreibt kurz wie ihr jede Operation implementieren würdet und erklärt warum die Operation die angegebene Komplexität in dieser Implementierung haben würde.
3. Gebe die Repräsentierung für Quartiler und implementiere in Pseudocode die `delete75Quartile` Operation.

Bemerkungen: Ihr könnt die Komplexität der Speicherfreigabe (`deallocate`) insgesamt ignorieren, wenn ihr die Komplexität einer Operation berechnet. Ihr könnt davon ausgehen, dass ihr alles (einschließlich einer verketteten Liste mit dynamischer Allokation) in $\Theta(1)$ Komplexität freigeben könnt. Ihr müsst keine `resize` Operation implementieren. Ihr müsst alle Funktionen oder Unterprogramme, die ihr für die Implementierung bei Punkt 3 braucht, selber implementieren. Ihr dürft nicht annehmen, dass es zusätzliche Operationen implementiert gibt, außer den Operationen die gegeben wurden. Ihr dürft neue Hilfsfunktionen definieren, aber dann müsst ihr diese auch implementieren.