Übungsaufgabe

Fortgeschrittene Datenstrukturen: Heaps, balancierte Suchbäume, Union-Find

Universität: Technische Universität Berlin
Kurs/Modul: Algorithmen und Datenstrukturen
Erstellungsdatum: September 6, 2025

Aufgabe 1: Fortgeschrittene Datenstrukturen – Heaps, balancierte Suchbäume, Union-Find

In dieser Aufgabe werden zentrale Konzepte zu Heaps, balancierten Suchbäumen (z. B. AVL- oder Rot-Schwarz-Bäume) sowie der Union-Find-Datenstruktur behandelt. Fassen Sie die Konzepte präzise zusammen, nennen Sie relevante Formeln und invarianten Bedingungen und diskutieren Sie Grenz- bzw. Amortisationsverhalten. Implementierungsdetails sind nicht vorgesehen.

a) Heaps

- Definieren Sie einen Min-Heap. Welche Heap-Eigenschaft muss für alle Knoten gelten?
- Beschreiben Sie Insert in einem Min-Heap und wie danach die Heap-Eigenschaft wiederhergestellt wird (Heapify-Up).
- Erläutern Sie Heapsort: Prinzip, Erzeugung der sortierten Folge und notwendige Schritte, um aus einem unsortierten Array eine sortierte Folge zu gewinnen.
- Geben Sie eine klare Notation der Laufzeitkomplexitäten an (Insert, Extract-Min) und deren Abhängigkeit von der Heap-Größe n.

b) Balancierte Suchbäume

- Skizzieren Sie die zentrale Idee eines balancierten Suchbaums. Welche Invariante(n) müssen eingehalten werden?
- Beschreiben Sie Rotationen als Rebalancing-Schritte (mindestens LL, RR, LR, RL) und geben Sie Beispiele für typische Auslösebedingungen.
- Warum garantiert ein AVL-Baum eine möglichst geringe Höhe? Nennen Sie eine obere Schranke der Höhe in Abhängigkeit von der Knotenzahl n.
- Diskutieren Sie Vor- und Nachteile balancierter Suchbäume gegenüber unbalancierten Bäumen hinsichtlich Worst-Case-Performance und Speicherverhalten.

c) Union-Find

- Beschreiben Sie die Union-Find-Ddatenstruktur mit den Operationen Find und Union. Welche Aufgabe erfüllen diese Operationen in Graphproblemen?
- Nennen Sie zwei Optimierungen, die typischerweise verwendet werden, um die Effizienz zu erhöhen: Pfadkompression und Union by Rank (oder Größe).
- Erklären Sie grob, wie diese Optimierungen die amortisierte Laufzeit pro Operation beeinflussen. Welche bekannte asymptotische Obergrenze ergibt sich für die Gesamtlaufzeit bei einer Folge von m Find- und n Union-Operationen?

Aufgabe 2: Implementierungssicht – praktische Aspekte von Heaps, balancierten Bäumen und Union-Find

- Diskutieren Sie, warum Heaps in Prioritätswarteschlangen häufig verwendet werden. Welche Vorteile bieten sie gegenüber alternativen Strukturen (z. B. sortierte Listen, Binärbäumen) in Bezug auf Insert und Remove-Min?
- Beschreiben Sie, wie man den Schlüsselwert in einem Min-Heap effizient aktualisiert (Key-Update) bzw. welche Operationen sinnvoll wären, um die Konsistenz der Heap-Eigenschaften zu wahren.
- Welche Eigenschaften von Union-Find-Strukturen sind relevant, wenn man sie in Multi-Threading-Szenarien einsetzen möchte? Welche Synchronisationsaspekte sind relevant?

Hinweis: Der Fokus liegt auf konzeptionellen, invarianten Begründungen und analytischen Einschätzungen; Implementierungsdetails bleiben offen.

Aufgabe 3: Vertiefung – Anwendungen und Analysen von Heaps, balancierten Bäumen und Union-Find

- Skizzieren Sie zwei typische Szenarien, in denen die drei Strukturen in Kombination sinnvoll eingesetzt werden (z. B. Graph- oder Netzwerkprobleme). Welche amortisierten bzw. worst-case Eigenschaften sind dabei relevant?
- Vergleichen Sie grob die Speicher- und Zeitkomplexität zentraler Operationen (Insert, Find/Union, Extract-Min) in diesen drei Strukturen.

Lösungen

Aufgabe 1: Fortgeschrittene Datenstrukturen – Heaps, balancierte Suchbäume, Union-Find

a) Heaps

- **Definition.** Ein Min-Heap ist eine 1-basierte Array-Repräsentation eines vollständigen Binärbaums, in dem die Heap-Eigenschaft gilt: Für jeden Knoten i mit $i > 1$ gilt

$$A[\lfloor i/2 \rfloor] \leq A[i].$$

  Dies implizit bedeutet, dass der Wurzelknoten $A[1]$ das minimale Element enthält.

- **Insert (Heapify-Up).** Füge das neue Element am Ende des Arrays hinzu ($A[n+1] \leftarrow x$, $n \leftarrow n+1$) und verschiebe es nach oben, solange $A[i] < A[\lfloor i/2 \rfloor]$ (Tauschen mit dem Elternelement). Stoppt, wenn die Heap-Eigenschaft wieder gilt. Laufzeit: $O(\log n)$.

- **Heapsort.** Prinzip: Zunächst Build-Heap durchlaufen

$$\text{for } i \leftarrow \lfloor n/2 \rfloor \text{ downto 1: heapify-down}(i).$$

  Dann wiederhole

$$\text{for } end \leftarrow n \text{ downto 2:} \quad \text{swap}(A[1], A[end]); \text{ heapify-down(1) mit n := end-1.}$$

  Aus der unsortierten Folge entsteht eine sortierte Folge in aufsteigender Reihenfolge. Laufzeit: Build-Heap $O(n)$; Heapsort insgesamt $O(n \log n)$.

- **Laufzeitnotationen.** Insert: $O(\log n)$. Extract-Min (bzw. Remove-Min): $O(\log n)$. Heapify-Up/Down jeweils $O(\log n)$. Build-Heap: $O(n)$. Heapsort: $O(n \log n)$.

b) Balancierte Suchbäume

- **Zentrale Idee.** Balancierte Suchbäume sichern durch entsprechende Rebalancing-Regeln eine Baumhöhe $H(n)$ von $O(\log n)$, sodass Such-, Einfüge- und Löschoperationen im Diskriminatorbereich bleiben.

- **Rotationen (LL, RR, LR, RL).** Rotationen dienen als Rebalancing-Schritte:

  - LL (single rotation rechts): Knoten mit zu schmalem linken Unterbaum wird rotiert, um die Balance wiederherzustellen.
  - RR (single rotation links): Spiegelung von LL.
  - LR (double Rotation: links-then-rechts): Erst Linksdrehung der linken Kindknoten, dann Rechtsdrehung des Wurzelknotens.
  - RL (double Rotation: rechts-then-links): Erst Rechtrotation der rechten Kindkante, dann Linkrotation des Wurzelknotens.

  Beispiele: Insertion in den linken linken Teilbaum löst LL-Überbalance aus; LR entsteht, wenn Einfügen in den rechten Teilbaum des linken Kindes passiert; ähnliche Muster gelten für RL/RR.

- **Geltende Invariante(n).** BST-Eigenschaft bleibt erhalten: Alle Schlüssel im linken Teilbaum eines Knotens sind kleiner als der Knoten, alle Schlüssel im rechten Teilbaum größer. Zusätzlich gilt eine Balance-Eigenschaft, z. B. beim AVL-Baum der Balancing-Faktor

$$BF(v) = \text{height}(left(v)) - \text{height}(right(v)), \quad \text{mit } |BF(v)| \leq 1 \text{ für alle } v.$$

- **Höhe von AVL-Bäumen.** Die Höhe ist $h(n)$ beschränkt durch

$$h(n) \leq \log_{\phi}(n+2) - 2, \quad \text{mit } \phi = \frac{1+\sqrt{5}}{2} \text{ (Goldener Schnitt)},$$

  da die minimale Knotenzahl eines AVL-Baums der Höhe h durch die Rekursion $N(h) = 1 + N(h-1) + N(h-2)$ mit $N(0) = 1$, $N(1) = 2$ gegeben ist. Folglich gilt $h(n) = O(\log n)$.

- **Vor- und Nachteile.** Vorteile: garantierte logarithmische Tiefe, damit Worst-Case-Operationen $O(\log n)$ bleiben. Nachteile: zusätzliche Balancier-Logik, Konstante Faktoren durch Rotationen, ggf. höherer Speicherbedarf (z. B. für Height-Labels oder Parent-Zuweisungen) im Vergleich zu unbalancierten Bäumen. Gegenunüber unbalancierten Bäumen (z. B. ungebohene BST) erreicht man im Worst-Case nur $O(n)$ Suchzeit, Speicherverhalten kann je nach Implementierung variieren.

c) Union-Find

- **Kernidee.** Die Union-Find-Datenstruktur modelliert eine Zerlegung in disjunkte Mengen. Die Operationen Find und Union dienen der Bestimmung von Repräsentanten bzw. dem Zusammenführen von Mengen, was in Graphproblemen z. B. zur Ermittlung von Zusammenhangskomponenten genutzt wird.

- **Optimierungen.**
  - Pfadkompression (Path Compression): Bei Find wird der Pfad von Knoten zur Wurzel flach gemacht, indem jeder Knoten auf die Wurzel zeigt.
  - Union by Rank (oder Union by Size): Beim Vereinen zweier Mengen wird der Wurzelknoten der schmaleren Menge an die Wurzel der breiteren Menge angehängt; der „Rank“ dient als heuristische Höhe.

- **Amortisierte Laufzeit.** Für eine Folge von m Find- und n Union-Operationen gilt

$$O\big((m+n)\,\alpha(n)\big),$$

  wobei $\alpha(n)$ die inverse Ackermann-Funktion ist. $\alpha(n)$ wächst extrem langsam (praktisch Konstante); selbst für reale Größen bleibt $\alpha(n) \leq 4$.

Hinweis: Die obigen Aussagen sind konzeptionell, invariantenbasiert und analytisch begründet; Implementierungsdetails bleiben offen.

Aufgabe 2: Implementierungssicht – praktische Aspekte von Heaps, balancierten Bäumen und Union-Find

- **Wieso Heaps in Prioritätswarteschlangen?** Heaps ermöglichen effiziente Einfügung und das Entfernen des aktuell kleinsten Elements im amortisierten bzw. worst-case Bereich $O(\log n)$. Gegenüber sortierten Listen: Einfügen ist effizienter (kein vollständiges Drehen oder erneute Sortierung nötig); gegenüber Binärbäumen bietet der Heap garantierte Worst-Case-$\log n$ für Insert/Extract-Min, während BSTs je nach Balancierung $O(\log n)$ im Durchschnitt kosten können, aber im Worst-Case auch $O(n)$ erreichen können. Speicherseitig sind Heaps oft kompakt als Arrays implementiert, was Speicher-Layout und Cache-Lokalität verbessert.

- **Key-Update in Min-Heap.** Zur Aktualisierung eines Schlüssels an Position i:

  – Falls der neue Schlüssel kleiner ist (Decrease-Key): Setze $A[i] \leftarrow newKey$ und führe Heapify-Up ab Index i durch (swappe nach oben, solange $A[i] < A[\lfloor i/2 \rfloor]$).

  – Falls der neue Schlüssel größer ist (Increase-Key): Setze $A[i] \leftarrow newKey$ und führe gegebenenfalls Heapify-Down ab Index i durch (swappe mit dem kleineren Kind, solange $A[i] > \min\{A[2i], A[2i+1]\}$).

  Dies sorgt dafür, dass die Heap-Eigenschaft wiederhergestellt wird. Zeitkomplexität: $O(\log n)$ im Worst-Case.

- **Union-Find in Multi-Threading.** Relevante Synchronisationsaspekte:

  – Die Grundoperationen Find und Union sind nicht thread-sicher, wenn mehrere Threads gleichzeitig auf dieselben Eltern-/Rank-Arrays zugreifen.

  – Möglichkeiten zur Synchronisation: (i) grob-granulare Sperren um die gesamte Struktur pro Operation; (ii) feingranulare Sperren pro Baumknoten (oder pro Wurzel) mit konsistentem Sperrmechanismus, um Deadlocks zu vermeiden; (iii) Einsatz von sperrfreien/konkurrierenden Varianten oder spezialisierter Parallel-Union-Find-Implementierungen; (iv) in vielen Szenarien genügt eine serielle Monitorsynchronisation vor Find/Union-Aufrufen.

Hinweis: Der Fokus liegt auf invarianten Begründungen und analytischen Einschätzungen; konkrete Parallelisierungs- oder Thread-Sicherheits-Implementierungen hängen von der Zielplattform ab.

Aufgabe 3: Vertiefung – Anwendungen und Analysen von Heaps, balancierten Bäumen und Union-Find

- **Two typische Szenarien, in denen die Strukturen kombiniert werden.**

  1. Kruskal's Algorithmus zur Bestimmung des kleinsten Spannbaums (MST). Hier nutzt man eine Sortierung der Kanten, anschließend eine Union-Find-Struktur, um zu prüfen, ob zwei Endpunkte bereits im gleichen Teilbaum liegen. Amortisiert kostet die Folge von Union- und Find-Operationen $O(m\,\alpha(n))$, bei welcher echte Kosten durch die Sortierung der Kanten dominiert werden (typisch $O(m \log m)$).
  2. Dijkstra's Algorithmus zur Kürzesten-Pfad-Bestimmung. Hier kommt eine Min-Heap-Prioritätswarteschlange zum Einsatz, um jedes Mal die noch nicht verarbeitete Knoten mit dem aktuell geringsten Distanzwert zu wählen. Die Komplexität beträgt $O((n + m) \log n)$ bei einem Binär-Heap; mit fortgeschrittenen Heaps (z. B. Fibonacci-Heap) kann man theoretisch besser, in der Praxis oft nicht signifikant besser, werden.

- **Speicher- und Zeitvergleiche zentraler Operationen.**

  – Insert: Min-Heap $O(\log n)$; AVL-Baum/Rotationsbasierte Bäume in $O(\log n)$; Union-Find-Operationen stehen hier nicht im direkten Zusammenhang.

  – Find/Union (Union-Find): Find mit Pfadkompression und Union durch Rang führt zu amortisierten Kosten $O(\alpha(n))$, also nahezu konstanter Zeit pro Operation.

  – Extract-Min (oder Remove-Min) in Heap: $O(\log n)$.