Sortieren

Übersicht Sortieralgorithmen

Name	Best Case	Average Case	Worst Case	Stabil?	In-Place?	Vergleich?
Bubblesort	$O(n)$	$O(n^2)$		ja	ja	ja
Insertionsort	$\Theta(n)$	$\Theta(n^2)$		ja	ja	ja
Selectionsort	$O(n^2)$			nein	ja	ja
Mergesort	$\Theta(n \cdot \lg n)$			ja	nein	ja
Quicksort	$\Theta(n \cdot \lg n)$		$\Theta(n^2)$	nein	nein $(+\log n)$	ja
Heapsort	$\Theta(n \cdot \lg n)$			nein	ja	ja
Radixsort	$\Theta(d(n+k))$			ja	nein $(+n)$	nein
Countingsort	$\Theta(n+k)$			ja	nein $(+(n+k))$	nein

Tabelle 1. Übersicht der Sortierverfahren. n = Anzahl zu sortierender Elemente, k = Anzahl diskreter Werte, die von einem Schlüssel (Countingsort) bzw. einer Stelle des Schlüssels (Radixsort) angenommen werden können, d = Anzahl Stellen des Schlüssels.

In der Praxis ist Quicksort meistens schneller als Heapsort, benötigt aber asymptotisch mehr zusätzlichen Speicher.

Graphen

Darstellung

	Vorteile	Nachteile
Adjazenzliste	kompakt, Kantenoperationen	Kantensuche, Cache-Misses
Adjazenzmatrix	Operationen in $O(1)$	Speicherverbrauch, Navigation
Adjazenzfeld	Platzsparend, Navigation	Änderungen aufwendig

Tabelle 2. Darstellungsmöglichkeiten von Graphen

Laufzeiten

Heaps

GetMax/Min	$O(1)$
ExtractMax/Min	$O(\lg n)$
Insert	$O(\lg n)$
Erzeugen durch Einfügen	$\Theta(n \lg n)$
Erzeugen durch „Heapify“	$O(n)$

Tabelle 3. Laufzeiten von Heap-Operationen

Felder

Operation	Liste (doppelt)	Liste (einfach)	Stack	Queue	Array	unbeschr. Feld
Search	$O(n)$	$O(n)$			$O(n)$	$O(n)$
Select(k)					erwartet $O(n)$	erwartet $O(n)$
Insert-Front	$O(1)$	$O(1)$				
Delete	$O(1)$	$O(n)$				
Push/Pop			$O(1)$			amortisiert $O(1)$
Enqueue/Dequeue				$O(1)$		
isEmpty	$O(1)$	$O(1)$	$O(1)$	$O(1)$		

Tabelle 4. Laufzeiten von Feldoperationen. Select(k) wählt das Element mit Rang k aus (Quickselect!)

Hashtabellen

Falls

- einfaches gleichmässiges Hashing verwendet wird
- eine doppelt verkettete Liste zur Kollisionsauflösung verwendet wird
- die Anzahl der Slots proportional zur Anzahl der gespeicherten Elemente ist

dann ist die Anzahl der Kollisionen *erwartet* in $O(1)$ und die Laufzeiten betragen:

Operation	Best Case	Average Case	Worst Case
SEARCH	$O(1)$	$O(1)$	$\Theta(n)$
INSERT		$O(1)$	
DELETE		$O(1)$	

Tabelle 5. Laufzeiten von Hashoperationen mit Verkettung

Annahme: Hashwertberechnung in $\Theta(1)$.

Belegungsfaktor $\alpha = \frac{m}{n}$ gibt die mittlere Länge einer Liste an. Erwartete Anzahl Sondierungen bei erfolgloser Suche beträgt also α.

Die erwartete Anzahl Sondierungen für eine erfolgreiche Suche beträgt $1 + \frac{\alpha}{2} + \frac{\alpha}{2\,m}$. $(\Theta(1+\alpha))$

Einfügen eines Elements findet am Anfang der verketteten Liste statt!

Offene Adressierung

Bei Verwendung von offener Adressierung, Belegungsfaktor α:

Operation	Anzahl Sondierungen im Mittel
Suche (erfolglos)	$\frac{1}{1-\alpha}$
Suche (erfolgreich)	$\frac{1}{\alpha} \cdot \ln(1-\alpha)$
Einfügen	$\frac{1}{1-\alpha}$

Tabelle 6. Laufzeiten von Hashoperationen mit offener Adressierung. Es wird angenommen, dass nach jedem Schlüssel mit gleicher Wahrscheinlichkeit gesucht wird.

Offene Adressierung bietet Speichervorteile (Platz für Zeiger bei verketteten Listen kann für zusätzlichen Speicher genutzt werden) und ist Cache-effizienter.

Hashfunktionen

Divisionsmethode. $h(k) = k \bmod m$

Geeignete Wahl für m (Hashtabellengrösse): Primzahl, die nicht nahe an einer Zweierpotenz liegt.

Multiplikationsmethode. $h(k) = \lfloor m\,(k \cdot A \bmod 1) \rfloor$

wobei A eine Konstante zwischen 0 und 1 ist und „mod 1“ den gebrochenen Rest bezeichnet.

Offene Adressierung

$h'(k)$ ist hier eine Hilfshashfunktion.

Alle drei Varianten bieten kein universelles Hashing, da sie maximal m^2 (statt $m!$) Sondierungssequenzen liefern.

Lineares Sondieren. $h(k,i) = (h'(k) + i) \bmod m$

Quadratisches Sondieren. $h(k,i) = (h'(k) + a_1 \cdot i + a_2 \cdot i^2) \bmod m$

Doppeltes Hashing. $h(k,i) = (h'(k) + i \cdot h''(k))$

mit zusätzlicher Hashfunktion $h''(k)$, deren Wert teilerfremd zu $h'(k)$ sein muss.

Wenn m Primzahl: $h'(k) = k \bmod m$ (Divisionsmethode), $h''(k) = 1 + (k \bmod (m-1))$ erfüllen diese Bedingung.

Suchbäume

Datenstruktur	Operation	Best Case	Average Case	Worst Case
Binärer Suchbaum	Search		$O(\lg n)$ [balanciert]	$O(n)$
	Insert		$O(\lg n)$ [balanciert]	$O(n)$
	Delete		$O(\lg n)$ [balanciert]	$O(n)$
Rot-Schwarz-Baum	Search		$O(\lg n)$	$O(\lg n)$
	Insert		$O(\lg n)$	$O(\lg n)$
	Delete		$O(\lg n)$	$O(\lg n)$
B-Baum	Search		$O(\lg m \cdot \lg n)$	
(max. m Einträge/Knoten)	Insert			$O(m \cdot \lg n)$
	Delete			$O(m \cdot \lg n)$

Tabelle 7.

Graphenalgorithmen

Tiefensuche	$\Theta(\|V\| + \|E\|)$	
Topologische Sortierung	$\Theta(\|V\| + \|E\|)$	benutzt Tiefensuche
Breitensuche	$O(\|V\| + \|E\|)$	
Bellman-Ford	$O(\|V\| \cdot \|E\|)$	
DAG_SHORTEST_PATHS	$\Theta(\|V\| + \|E\|)$	benutzt top. Sortierung
Dijkstra mit binärem Min-Heap	$O((\|V\| + \|E\|) \cdot \lg \|V\|)$	
Dijkstra mit Fibonacci-Heap	$O(\|V\| \cdot \lg \|V\| + \|E\|)$	
MST-Kruskal	$O(\|E\| \cdot \lg \|V\|)$	
MST-Prim	$O(\|E\| \cdot \lg \|V\|)$	mit binärem Min-Heap

Tabelle 8. Laufzeiten von Graphenalgorithmen

In *zusammenhängenden* Graphen laufen Tiefensuche, Breitensuche etc. in $O(|E|)$, da für die Anzahl der Kanten gilt $|E| \geqslant |V| + 1$, somit $|V| \in O(|E|)$.

Algorithmus von Kruskal

Benutzt Union-Find-Datenstrukturen (Strukturen zur Verwaltung disjunkter Mengen), um den MST aufzubauen: Zu Anfang wird für jeden Knoten eine eigene Komponente erzeugt (n mal MAKE_SET), dann wird in aufsteigender Reihenfolge der Kantengewichte jede Kante, die zwei Knoten aus unterschiedlichen Mengen verbindet, zum MST hinzugefügt. Diese Mengen werden dann vereinigt.

Vorteile: Gut für Graphen mit $|E| \in O(|V|)$

Algorithmus von Prim – MST von einem Knoten aus bilden

Benutzt eine Prioritätswarteschlange, in der die Knoten des Graphen gespeichert sind. Der Schlüssel ist dabei die minimale Distanz zum aktuellen MST. In jedem Schritt wird die Kante zum Knoten mit minimaler Distanz zum MST in den Spannbaum eingefügt.

Vorteile: Gut für Graphen mit vielen Kanten, asymptotisch gut

Zeugs

Logarithmengesetze

- $\log_a (x \cdot y) = \log_a x + \log_a y$
- $\log_a \frac{x}{y} = \log_a x - \log_a y$
- $\log_a x^r = r \cdot \log_a x$
- $\log_b r = \frac{\log_a r}{\log_a b}$
- $\log_x y = \frac{1}{\log_y x}$

Dynamische Programmierung

Zwei Ansätze:

Bottom-Up-Methode. Kleinste Teilprobleme zuerst lösen, Ergebnisse in Tabelle speichern. Beim Lösen eines Teilproblems stehen die Lösungen aller Unterprobleme zur Verfügung.

Top-Down-Memoisation. Problem rekursiv lösen, dabei jedoch Zwischenergebnisse in Tabelle speichern und vor dem Lösen eines Teilproblems nachschauen, ob das Zwischenergebnis schon einmal vorberechnet wurde.