Technische Universität München
Fakultät für Informatik
Lehrstuhl für Effiziente Algorithmen
Dr. Hanjo Täubig
Tobias Lieber

Sommersemester 2011
Übungsblatt 4
11. September 2011

Grundlagen: Algorithmen und Datenstrukturen

Abgabetermin: 11. September 2011

Hausaufgabe 1

Implementieren Sie den QuickSelect-Algorithmus.
Implementieren Sie in der Klasse `UIsqsArray` in der Funktion `quickSelect` den QuickSelect-Algorithmus der das i-te Elemente in dem unsortierten Feld `A` liefert.
Verwenden Sie für Ihre Implementierung die auf der Übungswebseite bereitgestellten Klassen und verändern Sie für Ihre Implementierung *ausschließlich* die Klasse `UIsqsArray`.

Lösungsvorschlag

Siehe Übungswebseite.

Hausaufgabe 2

Implementieren Sie in der Klasse `UIaHeap` einen adressierbaren binären Heap, der neben den Basisoperationen auch die Operationen `inserth`, `remove` und `decreaseKey` implementiert.
Verwenden Sie für Ihre Implementierung die auf der Übungswebseite bereitgestellten Klassen und verändern Sie für Ihre Implementierung *ausschließlich* die Klasse `UIaHeap`.
Hinweis: Hier macht die Benutzung eines Dynamischen Arrays Sinn. Falls Sie die früher benutzte Klasse `UIsArray` *benutzen wollen, so fügen Sie den Quellcode Ihrer Abgabe hinzu. Sie können aber auch das in Java implemntierte Dynamische Array benutzen, welches deutlich flexibler ist.*

Aufgabe 1

Seien A und B nicht leere Mengen und $\leq_A$ und $\leq_B$ totale Ordnungen auf A bzw. B.
Definieren Sie eine totale Ordnung auf $A \times B$.
Wozu brauchen wir den Begriff der totalen Ordnung im Zusammenhang der aktuellen Vorlesungsthemen Sortieren und Selektieren?
Hinweis: Eine lineare Ordnung $\leq$ auf einer Menge X ist

- transitiv: $\forall x, y, z \in X: \quad x \leq y \ \wedge \ y \leq z \ \Rightarrow \ x \leq z$
- antisymmetrisch: $\forall x, y \in X: \quad x \leq y \ \wedge \ y \leq x \ \Rightarrow \ x = y$
- linear (total): $\forall x, y \in X: \quad x \leq y \ \vee \ y \leq x$
- reflexiv: $\forall x \in X: \quad x \leq x$ (folgt aus der Linearität)

Lösungsvorschlag

Seien $(a_1, b_1), (a_2, b_2) \in A \times B$. Dann ist die wie folgt definierte Ordnung $\leq_{A\times B}$

$$(a_1, b_1) \leq_{A\times B} (a_2, b_2) \quad \Leftrightarrow \quad (a_1 \leq_A a_2 \wedge \neg(a_2 \leq_A a_1)) \vee (a_1 \leq_A a_2 \wedge a_2 \leq_A a_1 \wedge b_1 \leq_B b_2)$$

eine lineare Ordnung auf der Menge $A \times B$.
Sinngemäß übersetzt bedeutet das soviel wie:

$$(a_1, b_1) \leq_{A\times B} (a_2, b_2) \quad \Leftrightarrow \quad (a_1 <_A a_2) \ \vee \ (a_1 =_A a_2 \ \wedge \ b_1 \leq_B b_2).$$

Der Begriff der totalen Ordnung ist essentiell für jeden vergleichsbasierten Sortieralgorithmus (oder auch Selektionsalgorithmus). Ist es nicht Möglich eine Menge total zu ordnen, so ist nicht für alle Paare x, y von Elemente klar, welches der beiden Elemente größer oder kleiner ist. Somit ist nicht klar wie diese Elemente anzuordnen sind.
Dies ist eine grundsätzliche Voraussetzung für die Eingabemenge für vergleichsbasierte Sortieralgorithmen. Diese Defintion stellt eine Verallgemeinerung der $\leq$-Relation für einelementige Schlüßel aus der Vorlesung dar.

Aufgabe 2

Veranschaulichen Sie das Vorgehen des RadixSort-Algorithmus anhand der folgenden Wörter:

`alt, hort, kalt, bar, sport, spalt, ort, stall, spart, speer, bart`

Benutzen Sie die lexikographische Sortierung (Dabei sei das Leerzeichen ␣ kleiner als jeder andere Buchstabe). Es reicht aus, wenn sie als Alphabet die Buchstaben ␣, a, b, e, h, k, l, o, p, r, s, t verwenden und daher die Schlüssel 0,...,11 verwenden.

Lösungsvorschlag

- Ergänzen wir die Wörter, wie Zahlen, vorne mit dem kleinsten lexikoraphischen Zeichen erhalten wir folgendes Sortierung:

  Wir Sortieren nach dem letzten Zeichen.

␣	a	b	e	h	k	l	o	p	r	s	t
						stall			bar		alt
									speer		hort
											kalt
											sport
											spalt
											ort
											spart
											bart

  Wir erhalten daraus das Zwischenresultat: `stall, bar, speer, alt, hort, kalt, sport, spalt, ort, spart, bart`.

Sortieren nach dem vorletzten Zeichen:

␣	a	b	e	h	k	l	o	p	r	s	t
	bar		speer			stall			hort		
						alt			sport		
						kalt			ort		
						spalt			spart		
									bart		

Nach dem zweiten Zeichen: `bar, speer, stall, alt, kalt, spalt, hort, sport, ort, spart, bart`.

Sortieren nach dem drittletzten Zeichen:

␣	a	b	e	h	k	l	o	p	r	s	t
	stall	bar	speer				hort				
	alt						sport				
	kalt						ort				
	spalt										
	spart										
	bart										

Nach dem dritten Zeichen: `stall, alt, kalt, spalt, spart, bart, bar, speer, hort, sport, ort`.

Sortieren nach den 2. Zeichen:

␣	a	b	e	h	k	l	o	p	r	s	t
alt	bart			hort	kalt			spalt			stall
bar								spart			
ort								speer			
								sport			

Vor dem letzten Sortieren: `alt, bar, ort, bart, hort, kalt, spalt, spart, speer, sport, stall`.

Sortieren nach dem ersten Zeichen:

␣	a	b	e	h	k	l	o	p	r	s	t
alt										spalt	
bar										spart	
ort										speer	
bart										sport	
hort										stall	
kalt											

Ergibt: `alt, bar, ort, bart, hort, kalt, spalt, spart, speer, sport, stall`.

- Ergänzen wir die Wörter hinten mit dem kleinsten Zeichen, erhalten wir die korrekte lexikographische Sortierung:

  Sortieren nach dem letzten Zeichen:

␣	a	b	e	h	k	l	o	p	r	s	t
alt						stall			speer		sport
hort											spalt
kalt											spart
bar											
ort											
bart											

  Wir erhalten das Zwischenergebnis `alt, hort, kalt, bar, ort, bart, stall, speer, sport, spalt, spart`.

  Sortieren nach dem vorletzten Zeichen:

␣	a	b	e	h	k	l	o	p	r	s	t
alt			speer			stall			sport		hort
bar						spalt			spart		kalt
ort											bart

  Daraus erhalten wir das Zwischenresultat: `alt, bar, ort, speer, stall, spalt, sport, spart, hort, kalt, bart`.

  Sortieren nach dem dritten Zeichen:

␣	a	b	e	h	k	l	o	p	r	s	t
stall			speer			kalt	sport		bar		alt
spalt									hort		ort
spart									bart		

  Daraus erhalten wir das Zwischenresultat: `stall, spalt, spart, speer, kalt, sport, bar, hort, bart, alt, ort`.

  Sortieren nach dem zweiten Zeichen:

␣	a	b	e	h	k	l	o	p	r	s	t
	kalt					alt	hort	spalt	ort		stall
	bar							spart			
	bart							speer			
								sport			

  Daraus erhalten wir das Zwischenresultat: `kalt, bar, bart, alt, hort, spalt, spart, speer, sport ,ort, stall`.

Sortieren nach dem ersten Zeichen:

␣	a	b	e	h	k	l	o	p	r	s	t
	alt	bar		hort	kalt		ort			spalt	
		bart								spart	
										speer	
										sport	
										stall	

Daraus erhalten wir das Ergebnis: `alt, bar, bart, hort, kalt, ort, spalt, spart, speer, sport, stall`.

Aufgabe 3

Wir wollen in dieser Aufgabe den `QuickSort` Algorithmus näher analysieren.

a) Geben Sie eine Familie von Eingabearrays an, so dass der `QuickSort` Algorithmus für jedes n Laufzeit $\Omega(n^2)$ benötigt.

b) Zeigen Sie: Wird in dem Algorithmus als Pivot-Element das k-größte Element ausgewählt (wobei $k \in \mathbb{N}$ eine beliebige aber feste Konstante ist), so liegt die Worst-Case Laufzeit des modifizierten Algorithmus immer noch bei $\Omega(n^2)$.

Ist $|A| < k$, so wird das Pivot-Element wie im Algorithmus der Vorlesung gewählt.

c) Beweisen Sie: Wählen wir als Pivot-Element das $\lfloor \frac{9}{10} n \rfloor$-größte Element aus `A` ($|A| = n$), so ist die Worst-Case Laufzeit $\mathcal{O}(n \log n)$ (wenn die Selektion des Pivot Elements in $\mathcal{O}(n)$ realisiert ist).

Lösungsvorschlag

a) Eine Familie von Eingabearrays, für die der QuickSort Algorithmus für jedes $n \in \mathbb{N}$ $\Omega(n^2)$ Zeit benötigt, ist die Familie der sortierten Arrays.

b) Da als Pivot-Element immer das k-größte Element ausgewählt wird, wird der rekursive Aufruf `quickSort(l,i-1)` genau $\lfloor \frac{n}{k} \rfloor$-Mal durchgeführt. In jedem rekursiven Aufruf müssen alle $r - l$ Elemente mit dem Pivot-Element verglichen werden. Es müssen also schon durch den ersten rekursiven Aufruf mindestens

$$\sum_{j=1}^{\lfloor \frac{n}{k} \rfloor} kj \geq k \sum_{j=1}^{\frac{n}{k}-1} j = \frac{n(\frac{n}{k} - 1)}{2} = \frac{n^2}{2k} - \frac{n}{2} \in \Omega(n^2)$$

Vergleiche bzw. Rechenoperationen durchgeführt werden.

c) In jedem rekursiven Schritt ist die Anzahl der Rechenoperationen durch das Vergleichen des gesamten Arrays mit dem, bzw. den Pivot-Element/en beschränkt (es gibt mehrere Pivot-Elemente ab der zweiten Stufe der Rekursion). Die Anzahl der Rechenoperationen ist also durch die Tiefe i der rekursiven Aufrufe und durch die Anzahl der zu sortierenden Elemente beschränkt ($\mathcal{O}(ni)$).

Sei n_i die Anzahl der Elemente die noch zu sortieren sind nach dem i-ten (ersten) rekursiven Aufruf (bzw. r_i und l_i die rechten und linken Grenzen). Die Selektion des Pivot-Elementes wird so oft durchgeführt, bis sich die Rekursion trivialisiert, also $r_i \leq l_i + 1$ gilt. Wir sind also an dem größten i interessiert so dass $r_i \geq l_i + 1$ gilt(denn so lange diese Gleichung noch gilt muss noch rekursiv abgestiegen werden).

Dies ist äquivalent mit der Forderung

$$r_i - l_i \geq 1 \Leftrightarrow n_i \geq 1. \tag{1}$$

Da in jedem Rekursionsschritt (wir betrachten nur den ersten Rekursionsschritt) die Eingabegröße um $\frac{9}{10}$ verringert wird, ist Ungleichung (1) äquivalent zu der Frage, für welches i die Ungleichung $\left(\frac{9}{10}\right)^i n \geq 1$ erfüllt ist. Daraus ergibt sich:

$$\left(\frac{9}{10}\right)^i n \geq 1 \Leftrightarrow n \geq \left(\frac{10}{9}\right)^i \Leftrightarrow i \leq \log_{\frac{10}{9}} n \Leftrightarrow i \in \mathcal{O}(\log n)$$

Insgesamt ergibt sich also für diese Variante des QuickSort Algorithmus eine Laufzeitschranke von: $\mathcal{O}(n \log n)$.

Aufgabe 4

Beweisen Sie folgende in der Vorlesung benötigte Aussage: Ein fast vollständiger Binärbaum der Höhe h hat $2^{h-1} \leq n \leq 2^h - 1$ Knoten (Die Wurzel hat Höhe 1).
Definition; Ein fast vollständiger Binärbaum der Höhe h ist ein vollständiger Binärbaum der Höhe $h-1$ dessen Knoten so angeordnet werden können, dass es auf dem Level h einen Knoten e mit dem Vaterknoten v gibt so dass gilt:

- Alle Knoten auf dem Level $h-1$ die „links“ von v stehen zwei Kinder haben
- und alle Knoten auf dem Level $h-1$ die „rechts“ von v stehen keine Kinder haben
- oder e ist die Wurzel.

Lösungsvorschlag

Wir betrachten zwei Fälle:

- Ein vollständiger Binärbaum der Höhe h gibt immer eine obere Schranke für die Anzahl der Knoten in einem fast vollständigen Binärbaum an.
- Ein vollständiger Binärbaum der Höhe $h-1$ mit einem zusätzlichem Knoten (ganz links) gibt eine untere Schranke für die Anzahl der Knoten in einem fast vollständigen Binärbaum an.

Somit ergibt sich: $1 + \sum_{j=1}^{h-1} 2^{j-1} \leq n \leq \sum_{k=1}^{h} 2^{k-1}$. Mit Indextransformationen erhalten wir: $1 + \sum_{j=0}^{h-2} 2^j \leq n \leq \sum_{k=0}^{h-1} 2^k$. Mit $\sum_{k=0}^{n} 2^k = 2^{n+1} - 1$ (kann auch mit Induktion bewiesen werden) erhalten wir das angestrebte Resultat: $2^{h-1} \leq n \leq 2^h - 1$.

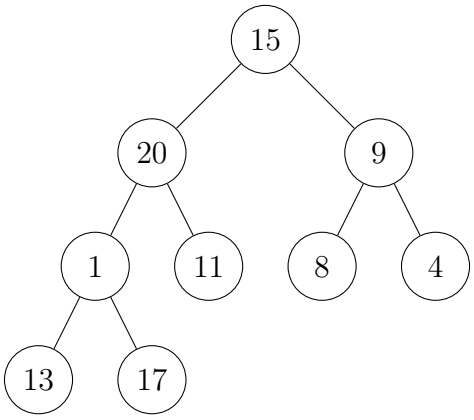
Aufgabe 5

Führen Sie auf einem anfangs leeren Binären Heap folgende Operationen aus und stellen Sie die Zwischenergebnisse graphisch dar:

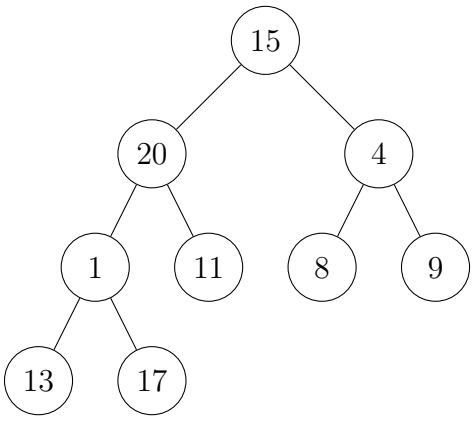
- build({15, 20, 9, 1, 11, 8, 4, 13, 17}),
- insert(7),
- delMin(),
- decreaseKey(20, 3),
- delete(8).

Lösungsvorschlag

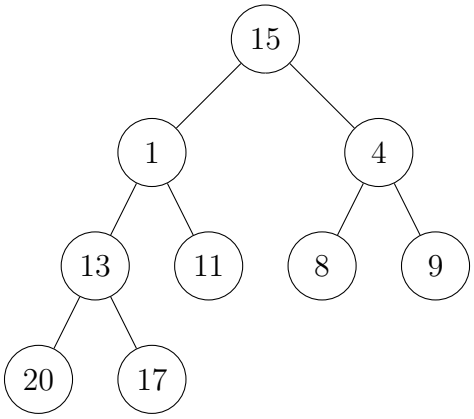
Anfang von Build:



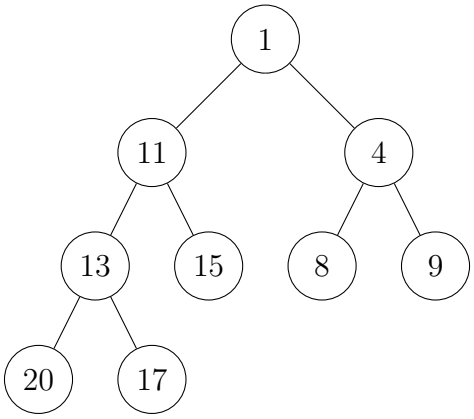
Bei siftDown von 1 passiert nichts. Daher gleich die 9:



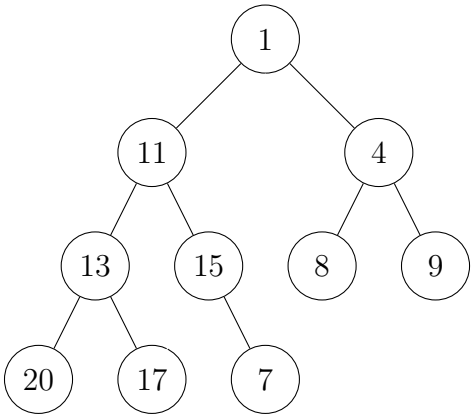
SiftDown auf der 20:



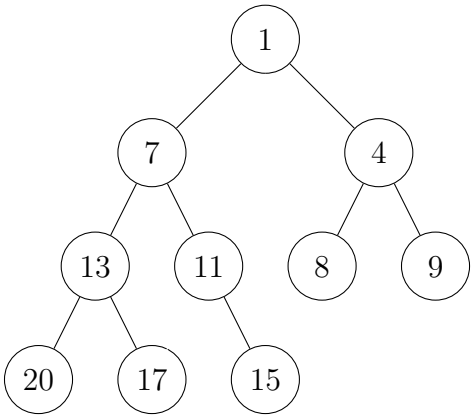
SiftDown auf der 15 und damit Endresultat für Build:



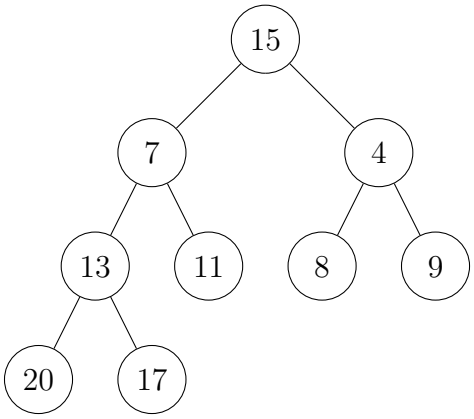
Einfügen der 7. Zuerst hinten Anhängen.



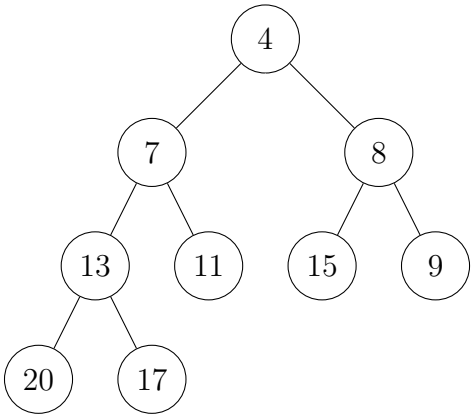
SiftUp auf die 7:



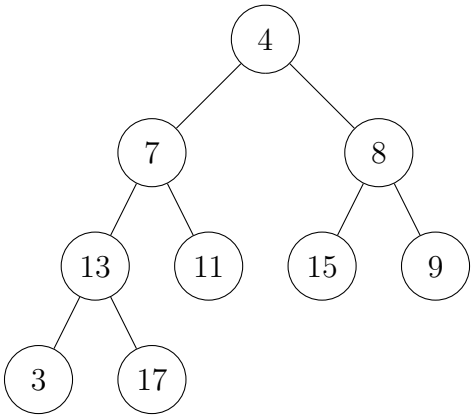
Vorbereitung für deleteMin:



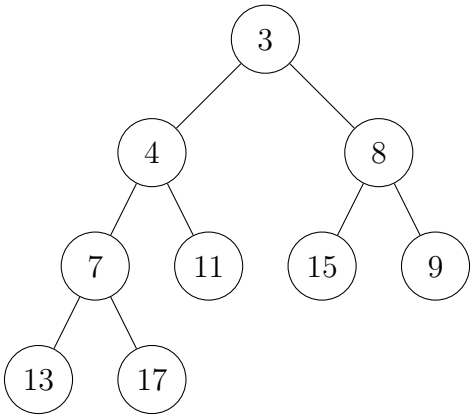
SiftDown auf die 15:



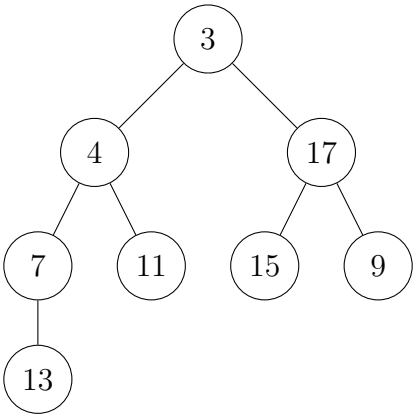
Wertänderung der 20 auf 3:



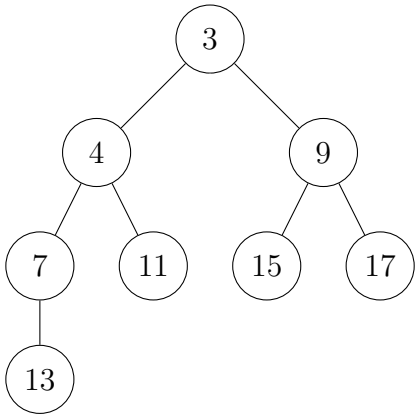
SiftUp der 3:



Delete von 8 vorbereiten:



SiftDown der 17.



Aufgabe 6

Analysieren Sie die Anzahl der Element-Vergleiche, die der in der Vorlesung angegebene `siftDown`-Algorithmus im Worst-Case benötigt.
Geben Sie ausgehend von dem Algorithmus in der Vorlesung einen Algorithmus an, der mit $\log n + \mathcal{O}(\log \log n)$ Vergleichen auskommt.

Lösungsvorschlag

Der in der Vorlesung angegebene Algorithmus führt in jedem Durchlauf der while-Schleife gerade 2 Elementvergleiche aus – Vergleich der Kinder und Test ob das kleinere Kind kleiner als das Element ist. Somit werden gerade $2 \log n$ Vergleiche benötigt.
Zur Reduzierung der Vergleiche benutzen wir zusätzlichen Speicherplatz um uns den Pfad der kleinsten Kinderelemente zu zpeichern. Dieser Pfad hat höchstens Länge $\lceil \log n \rceil$. Zum Finden der korrekten Position führen wir nun eine binäre Suche auf dem Pfad aus. Diese benötigt $\mathcal{O}(\log \log n)$ Vergleiche.
Somit erhalten wir als Gesamtanzahl von Vergleichen: $\log n + \mathcal{O}(\log \log n)$.