Algorithmen I
Übungsklausur
Musterlösung
11.07.2012

Name:

Aufgabe 1: Wissensfragen (6 Punkte)

(1) Nennen Sie zwei in der Vorlesung vorgestellte Algorithmen, die nach dem Greedy-Prinzip arbeiten. Begründen Sie kurz, warum. (1P)
mögliche Lösungen:

- Kruskal: es wird immer die Kante mit dem niedrigsten Gewicht ausgewählt, die Teil eines minimalen Spannbaums sein kann.
- Prim: es wird immer die leichteste Kante, die den bisherigen MST mit einem neuen Knoten verbindet, ausgewählt.
- Dijkstra: Durch die Verwendung einer Prioritätswarteschlange werden immer die ausgehenden Kanten des Knotens mit der geringsten Distanz relaxiert.

(2) Nennen Sie einen Vorteil und einen Nachteil von Mergesort gegenüber Quicksort. (1P)
mögliche Lösungen:

(+) besseres Verhalten im Worst Case

(+) stabil

(-) in der Regel langsamer als Quicksort

(-) zusätzlicher Speicher benötigt (nicht in-place)

(3) Was bedeutet Stabilität bei Sortieralgorithmen? Nennen Sie eine Situation, in der man die Stabilität ausnutzen kann. (2P)
Lösung:
Stabilität bedeutet, dass sich die Reihenfolge von gleichen Elementen beim Sortieren nicht ändert. Anwenden lässt sich dies z.B. bei Tupeln, die lexikographisch geordnet werden (d.h. zuerst nach dem ersten Wert, bei Gleichheit nach dem zweiten). Ein Beispiel hierfür wären Tupel der Form (Monat, Tag) für einen Tag innerhalb eines Jahres. Man sortiert dabei zuerst nach dem zweiten Wert und danach mit einem stabilen Sortieralgorithmus nach dem ersten Wert. Bei Tupeln mit gleicher erster Komponente wird dann nämlich die ursprüngliche Reihenfolge beibehalten, welche durch Sortieren nach der zweiten Komponente erreicht wurde.

(4) Zeigen oder widerlegen Sie: Binäre Suche hat auf jeder Datenstruktur eine bessere Laufzeit als Lineare Suche. (2P)
Lösung:
Die Behauptung ist falsch! Beispielsweise bringt binäre Suche keinen Geschwindigkeitsvorteil bei einer verketteten Liste, da diese keinen wahlfreien Zugriff in O(1) erlaubt.

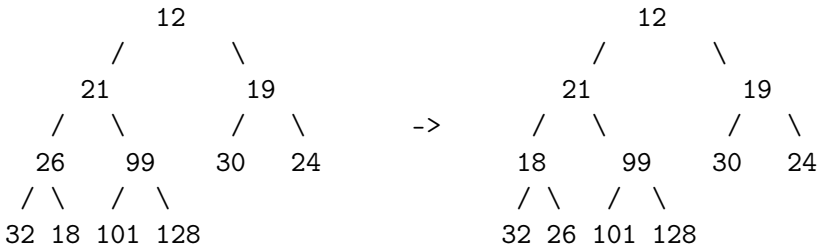
Aufgabe 2: Heaps (12 Punkte)

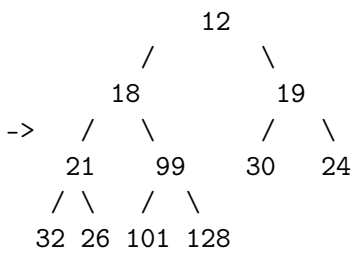
(1) Gegeben sei das folgende Feld:

12	21	19	26	99	30	24	32	18	101	128

(a) Erläutern Sie, warum das Feld keinen gültigen Min-Heap darstellt. (1P)
Lösung:
Das Element mit dem Schlüssel 18 verletzt die Heap-Eigenschaft, da es kleiner als 26 und 21 ist, aber weiter unten als diese im gleichen Ast des Heaps steht.

(b) Reparieren Sie den Min-Heap mit dem aus der Vorlesung bekannten Verfahren. Geben Sie dabei den Heap nach jeder Änderung an (als Feld oder als Baum). (3P)
Lösung:





(2) (a) Warum lässt sich in einem Heap, egal wie gut er implementiert ist, das Minimum nicht in $\Theta(1)$ entfernen, sodass die entstehende Struktur immer noch ein Heap ist? (4P)
Hinweis: Betrachten Sie die Laufzeit einer Heapsort-Implementierung, wenn dies

möglich wäre. Welcher Widerspruch ergibt sich daraus?
Lösung:
Der Heap lässt sich in $\Theta(n)$ aufbauen. Wenn das Extrahieren des Minimums jedes Mal $\Theta(1)$ braucht, dauern alle Extrahierungsaktionen zusammen $\Theta(n)$, da man diesen Schritt während des Sortiervorgangs n Mal macht. Es ergibt sich eine Gesamtlaufzeit von $\Theta(n)$, was der unteren Schranke des Sortierproblems ($\Theta(nlog(n))$ widerspricht.

(b) Geben Sie eine Datenstruktur an, die die entsprechende Eigenschaft aus (a) hat. (3P)
Lösung:
Eine aufsteigend sortierte verkettete Liste erfüllt genau das gewünschte (Extrahieren des Minimums ist dann einfach „remove head“).

(c) Warum ist dies kein Widerspruch zur Überlegung aus (a)? (1P)
Lösung:
Beim Aufbau dieser Datenstruktur muss die Liste sortiert werden. Dies braucht wegen der unteren Schranke mindestens $\Theta(n \log n)$. Für ein Sortierverfahren ist der Aufbau der Datenstruktur aber nötig. Also widerspricht dies nicht den Überlegungen aus (a).

Aufgabe 3: Sortieren (13 Punkte)

(1) Gegebenen sei der Sortieralgorithmus *Cocktail-Sort*. Um ein Array zu sortieren, durchschreitet der Algorithmus das Array zunächst von vorne nach hinten und vertauscht dabei zwei benachbarte Werte, wenn sie in der falschen Reihenfolge stehen. Ist der Algorithmus am Ende des Arrays angekommen, durchschreitet er das Array jetzt in umgekehrter Reihenfolge von hinten nach vorne (bewegt sich also „wieder zurück“). Dies wird so lange fortgesetzt, bis in einem Durchlauf keine Elemente mehr vertauscht werden.

(a) Wenden Sie Cocktail-Sort auf folgendes Array an, um die Elemente **aufsteigend** zu sortieren. Geben Sie dabei das Array nach jedem Schritt an. (3P)

4	2	1	3	7	3

Lösung:

4	2	1	3	7	3
2	4	1	3	7	3
2	1	4	3	7	3
2	1	3	4	7	3
2	1	3	4	7	3
2	1	3	4	3	7
2	1	3	4	3	7
2	1	3	3	4	7
2	1	3	3	4	7
2	1	3	3	4	7
1	2	3	3	4	7

(b) Geben Sie eine naive Implementierung von *Cocktail-Sort* in Pseudocode an. (7P)
Lösung:

```
COCKTAIL-SORT(A)
01  do
02      vertauscht = false;
03      for i = 1 to A.länge - 1
04          if A[i]) > A[i + 1]
05              swap(A[i], A[i + 1]);
06              vertauscht = true;
07      if vertauscht == false
08          break;
09      vertauscht = false;
10      for i = A.länge - 1 downto 1
11          if A[i] > A[i + 1]
12              swap(A[i], A[i + 1]);
13              vertauscht = true;
14  while vertauscht == true;
```

(c) Geben Sie eine *Familie* von Zahlenarrays an, die von Bubblesort in $O(n^2)$ sortiert werden, aber von Cocktail-Sort in $O(n)$. (3P)
Lösung:
Beispielsweise die Arrays, die bereits sortiert sind bis auf das kleinste Element, welches ganz rechts steht:

2	...	n	1

Aufgabe 4: Dynamische Programmierung (5 Punkte)

(1) Was ist die grundlegende Idee hinter der Methode der dynamischen Programmierung? (1P)
Lösung:
Mit dynamischer Programmierung wird verhindert, dass überlappende Teilprobleme eines Optimierungsproblems unnötig mehrfach gelöst werden. Dafür werden die Teillösungen zwischengespeichert, um sie wiederverwenden zu können.

(2) Die Fakultätsfunktion $n!$ lässt sich folgendermaßen rekursiv definieren:
$0! = 1$
$n! = n \cdot (n-1)!$ für $n \geq 1$

(a) Geben Sie ein Programm in Pseudocode an, welches $n!$ mittels dynamischer Programmierung und Bottom-Up-Ansatz berechnet. (3P)
Lösung:

```
FACTORIAL(n)
01  //sei f[0..n] ein neues Feld
02  f[0] = 1;
03  for i = 1 to n
04        f[i] = f[i – 1] * i;
05  return f[n];
```

(b) Das Programm soll nun so modifiziert werden, dass es nach einmaligem Berechnen von $n!$ jeden Aufruf $k!$ mit $k \leq n$ in $O(1)$ bearbeiten kann. Beschreiben Sie eine Möglichkeit dafür. (1P)
Lösung:
Da das dynamische Programm während der Bearbeitung alle Zwischenergebnisse $k!$ $(k \leq n)$ bereits in der Tabelle f speichert, muss nur dafür gesorgt werden, dass diese auch nach dem Ablauf der Methode erhalten bleibt. In jedem weiteren Aufruf für $k!$ $(k \leq n)$ kann nun durch einfaches Auslesen des Werts f[k] das Ergebnis geliefert werden.

Aufgabe 5: Datenstrukturen (12 Punkte)

(1) Für die folgenden Anwendungsfälle soll jeweils eine geeignete Datenstruktur ausgewählt werden. Geben Sie jeweils eine passende Datenstruktur an und begründen Sie Ihre Wahl:

(a) In einer Musiksammlung sollen häufig neue Musikstücke hinzugefügt, gelöscht und gesucht werden. Über die zukünftige Größe der Sammlung kann beim Anlegen noch keine Aussage gemacht werden. (1P)

Lösung:
Hashtabelle mit Verkettung - Operationen laufen bei hinreichender Größe des Feldes in $O(1)$, Speicherplatz ist nicht beschränkt (außer durch Größe des Speichermediums).

(b) An einen Datenbankserver werden zu manchen Zeitpunkten so viele Anfragen geschickt, dass er sie nicht sofort bearbeiten kann. Daher soll er die Möglichkeit bekommen, die Anfragen zwischenspeichern zu können, bis er wieder genügend freie Ressourcen besitzt. (1P)
Lösung:
Warteschlange (Queue) - durch die FIFO-Strategie ändert sich die Reihenfolge der Anfragen beim Puffern nicht, außerdem werden ENQUEUE und DEQUEUE in jeweils konstantem Zeitaufwand ausgeführt.

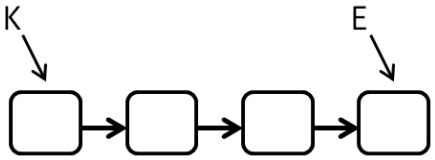
(c) Ein Prozess-Scheduler in einem Betriebssystem arbeitet mit unterschiedlich hohen Prioritäten. Bei jedem Aufruf soll jeweils der Prozess mit der höchsten Priorität ausgeführt werden, wobei die Auswahlgeschwindigkeit für die Leistungsfähigkeit des Betriebssystems eine entscheidende Rolle spielt. (1P)
Lösung:
Max-Heap, da hier sowohl die Auswahl des Elements mit dem größten Schlüssel als auch das Einfügen neuer Prozesse hier besonders schnell ($O(log(n))$ erfolgen kann.

(2) Zeigen Sie, wie eine Warteschlange mit einer **einfach verketteten** Liste sowie zwei Zeigern K (ältestes Element) und E (neuestes Element) implementiert werden kann. Hierbei sollen sowohl ENQUEUE(X) als auch DEQUEUE() die Laufzeit O(1) besitzen. (4P)
Gehen Sie dafür folgendermaßen vor:

- Stellen Sie mit einer Skizze dar, wie die beiden Zeiger in der Liste positioniert werden müssen.
- Geben Sie den Pseudocode für beide Warteschlangenoperationen an. Stellen Sie dabei sicher, dass Ihre Implementierung auch korrekt mit einer leeren Warteschlange umgehen kann.

Lösung:

- Skizze:



Anm: K muss auf den Listenanfang zeigen, da nur hier das Löschen in konstanter Zeit möglich ist.

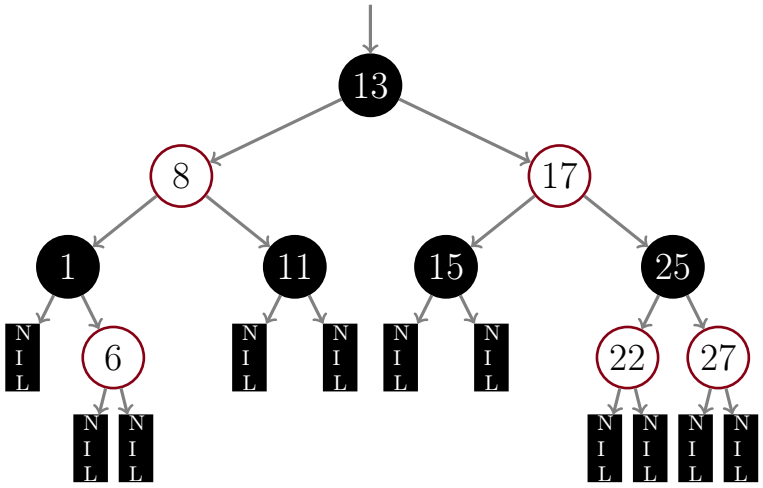
- *ENQUEUE(x)*

```
01   if (E == NIL)
02         E = x;
03   else
04         E.next = x;
05         E = E.next;
```

DEQUEUE()

```
01   if (K == NIL)
02         error("Underflow!");
03   else
04         x = K;
05         K = K.next;
06         return x;
```

(3) Löschen Sie in folgendem Rot-Schwarz-Baum das Element mit dem Schlüssel 17 und zeichenen Sie den Baum nach Ablauf der Operation. Sie können dabei die schwarzen Knoten durch einen doppelten Kreis kennzeichnen, die roten durch einen einfachen. (5P)



Lösung:

Knoten 17 ist kein Blattknoten, kann also nicht direkt entfernt werden. Er muss also zuerst mit einem Blattknoten getauscht werden. Dabei stehen zwei Kandidaten zur Auswahl: Die 15 und die 22. Es ist offensichtlich einfacher, die 22 zu tauschen, denn rote Knoten sind unproblematisch. Man tauscht also die 17 mit der 22 und entfernt die 17 anschließend. Dann ist man sofort fertig.

Für Gourmets: *(die unbedingt die 15 tauschen wollen)*
Nach dem Tauschen und Entfernen des Knotens ist das linke NIL-Element (unter der 15) der Knoten x. Der Bruder ist also die 25, die selber schwarz ist und zwei rote Kinder hat. Laut Vorlesungsfolien tritt also Fall 4 auf. Es muss rotiert (nach links um die 15) und umgefärbt werden (25 rot, 27 schwarz, 15 schwarz). Damit ist die Reparatur abgeschlossen.

Aufgabe 6: Graphen (12 Punkte)

(1) Ein Graph G = (V,E) sei durch die folgende Adjazenzfelddarstellung gegeben:

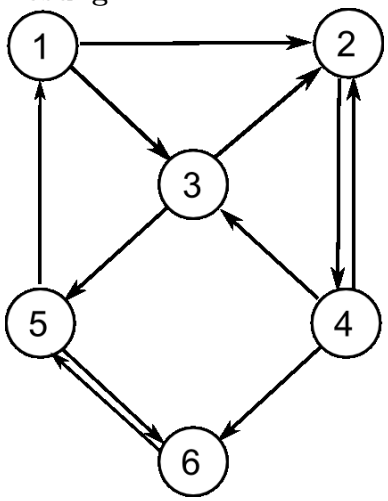
V :=

1	3	4	6	9	11

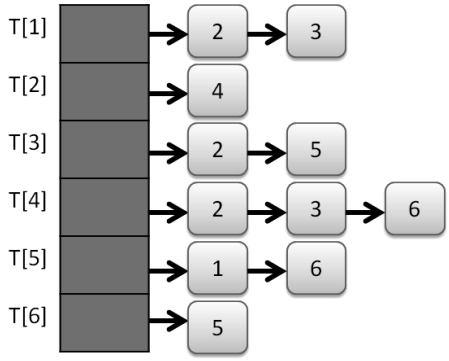
E :=

2	3	4	2	5	2	3	6	1	6	5

(a) Zeichnen Sie den Graphen. (2P)
Lösung:



(b) Geben Sie den Graphen in Adjazenzlistendarstellung an. (1P)
Lösung:



(c) Welche dritte Möglichkeit zur Darstellung von Graphen wurde in der Vorlesung vorgestellt? Beschreiben Sie sie kurz und nennen Sie sowohl einen Vor- als auch einen Nachteil dieser Methode. (2P)

Lösung

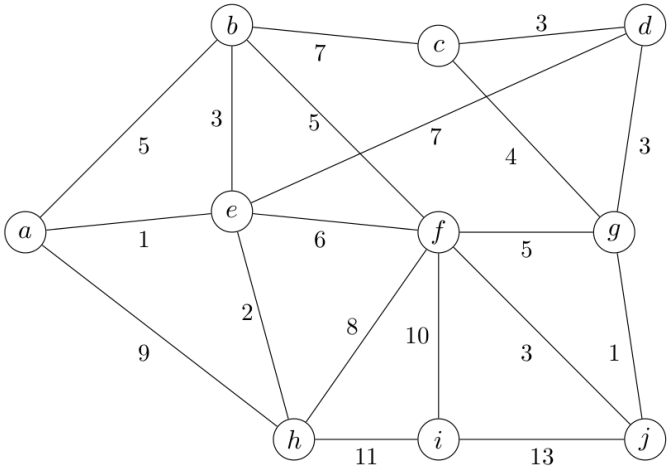
Die dritte vorgestellte Möglichkeit ist die Adjazenzmatrixdarstellung. In einer $|V| \times |V|$-Matrix A wird an der Stelle $A_{i,j}$ eine 1 gesetzt, wenn zwischen den Knoten i und j eine Kante existiert, andernfalls eine 0.
Vorteile:

- Problem "Existiert Kante von i nach j?" in konstanter Zeit lösbar
- Einfaches Hinzufügen und Löschen von Kanten möglich
- manche Graphenoperationen direkt durch Matrixoperationen durchführbar (z.B. "Existiert ein Pfad von A nach B?" durch Potenzieren der Adjazenzmatrix)

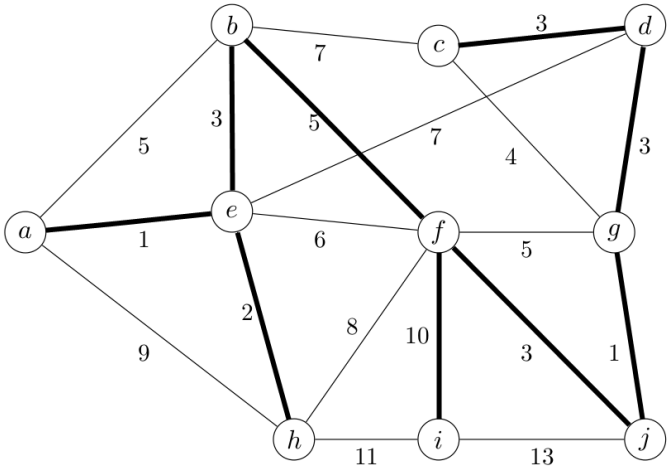
Nachteile:

- verhältnismäßig hoher Speicherverbrauch bei dünnen Graphen

(2) Wenden Sie den Algorithmus von Prim auf dem folgenden Graphen an. Der Startknoten sei **a**. Markieren Sie die Kanten des resultierenden MSTs eindeutig (z.B. durch dicke Linien) und geben Sie die Reihenfolge der hinzugefügten Kanten an. (2P)

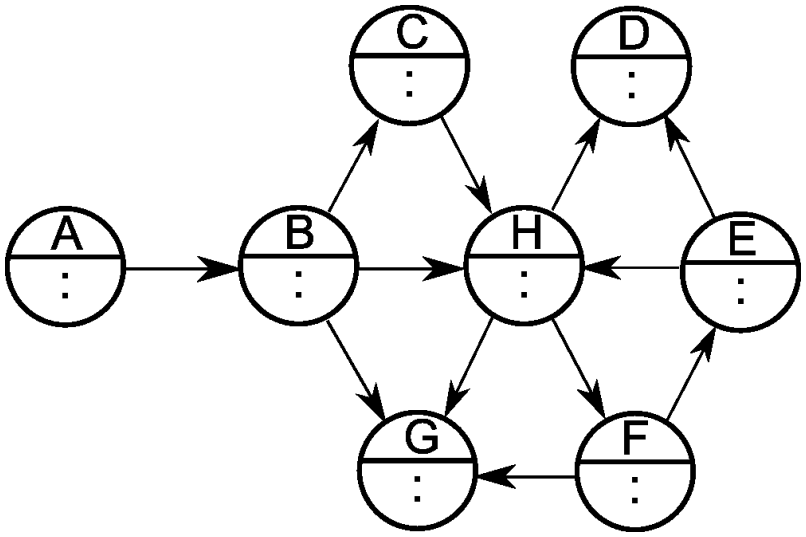


Lösung:



Reihenfolge der Kanten: (a, e); (e, h); (e, b); (b, f); (f, j); (j, g); (g, d); (d, c); (f, i).

(3) Gegeben sei folgender Graph:



Führen Sie auf diesem Graphen eine Tiefensuche aus und gehen Sie dabei wie folgt vor:

- Beginnen Sie bei Knoten A.
- Betrachten Sie die ausgehenden Kanten eines Knotens bei der Suche jeweils im Uhrzeigersinn.
- Schreiben Sie die *discovered*-Zeiten der Knoten links neben die Doppelpunkte, die *finalized*-Zeiten jeweils rechts daneben.
- Klassifzieren Sie die Kanten. Kennzeichnen Sie dabei:
  - jede *Baumkante* durch eine dicke Linie
  - jede *Rückwärtskante* mit einem "B"
  - jede *Vorwärtskante* mit einem "F"
  - jede *Querkante* mit einem "C"

(5P)

Lösung: