Universität des Saarlandes
FR 6.2 Informatik
Prof. Dr. Hans-Peter Lenhof
Dipl. Inform. Jan Küntzer
Dipl. Inform. Alexander Rurainski

Programmierung II, SS 2004
Musterlösung zur 12. Übung

Aufgabe 1: Wissenstest (0 Punkte)

(a) Eine *Template-Klasse* kann in C++ dazu benutzt werden, Datenstrukturen und Algorithmen so unabhängig wie möglich vom Typ der Eingabedaten u.Ä. zu implementieren. So können z.B. in einem `vector<T>` Elemente eines beliebigen Typs `T` gespeichert werden, z.B. `int` - Variablen in einem `vector<int>`.

(b) Sei $F := \{f | f : \mathbb{N} \to \mathbb{R}_0^+\}$. Dann definieren wir für $g \in F$:

$$\Theta(g) := \{f \in F | \exists c_1, c_2 > 0 \wedge \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

(c) **Doppelt verkettete sortierte Liste:**

- *Suche* geht bei der doppelt verketteten sortierten Liste in $\mathcal{O}(n)$, denn wir müssen uns von vorne oder von hinten an durchhangeln bis wir das Element gefunden haben und können dabei die Ordnungseigenschaft nicht ausnutzen
- wenn *vorne anfügen* die Sortierung zerstören darf, dann geht es offenbar in $\mathcal{O}(1)$. Wenn wir die Eigenschaft erhalten wollen, dann müssen wir zunächst den richtigen Einfügeort suchen ($\mathcal{O}(n)$, wenn n die Länge der Liste bezeichnet) und dann an der gefunden Stelle in $\mathcal{O}(1)$ einfügen $\Rightarrow \mathcal{O}(n)$.

Unsortiertes Array:

- Da wir keine Informationen über die gespeicherten Elemente haben, müssen wir sie bei der *Suche* u.U. alle überprüfen bis wir das gesuchte Element gefunden haben $\Rightarrow \mathcal{O}(n)$
- Da wir das gesamte bisher gespeicherte Array nach hinten verschieben müssen um Platz für das neue Element zu schaffen, läuft hier *vorne anfügen* in $\mathcal{O}(n)$.

(d) (i) linear Probing: wenn eine Kollision auftritt, wird so lange ein neuer Hashwert mit einer Linearkombination des alten Hashwertes und einem Laufindex i berechnet, bis man auf ein freies Feld stößt. $h(k,i) = (h_{\text{alt}}(k) + i) \mod m$.

(ii) quadratic Probing: wie linear Probing, allerdings mit Polynom zweiten Grades: $h(k,i) = (h_{\text{alt}}(k) + c_1 i + c_2 i^2) \mod m$.

(iii) double hashing: hier wird eine zweite Hashfunktion verwendet, um neue Hashwerte zu berechnen bis ein freies Feld gefunden wird $h(k,i) = (h_{\text{alt}}(k) + i h_2(k)) \mod m$

(e) Eine endliche Menge $\mathcal{H}$ von Hashfunktionen $h : \mathcal{U} \to \{0, 1, \ldots, m-1\}$ heißt *universelle Klasse von Hashfunktionen*, wenn für jedes Paar k_i, k_j von verschiedenen Schlüsseln gilt: die Zahl der Funktionen $h \in \mathcal{H}$ mit $h(k_i) = h(k_j)$ ist höchstens $\frac{|\mathcal{H}|}{m}$.

(f) Ein *Circuit* oder *Rundgang* in einem Graphen ist ein Pfad, bei dem der erste und der letzte Knoten identisch sind. Ein *Euler-Circuit* ist ein solcher Rundgang, der *jede Kante* des Graphen *genau einmal* enthält.

Aufgabe 2: Programmieraufgabe (10 Punkte)

```
template <class T> class liste


  // Struktur des Knotens einer doppelt verketteten Liste
  struct knoten

    // Element des Listenknotens
    T element;

    // Zeiger auf linken Nachbarn
    knoten* links;

    // Zeiger auf rechten Nachbarn
    knoten* rechts;
  ;

public:

  // Defaultkonstruktor
  liste();

  // Copykonstruktor
  liste(const liste<T>& b);

  // Destruktor
  ~liste();

  // Zuweisungsoperator
  liste operator=(const liste<T>& op2);

  // Elementanzahl
  int elementAnzahl();

  // Einfuegen eines Elements am linken Ende
  void einfuegen_links(const T& i);
```

```
  // Loeschen aller Vorkommen von i in der Liste
  void loesche(const T& i);

  // gibt zurueck, ob i in der Liste enthalten ist
  bool suche(const T& i);

protected:

  // Zeiger auf das linke Ende der Liste,
  // 0 wenn Liste leer, wird im Konstruktor und 'loesche' gesetzt
  knoten* linkes_ende_;

  // Zeiger auf das rechte Ende der Liste,
  // 0 wenn Liste leer, wird im Konstruktor und 'loesche' gesetzt
  knoten* rechtes_ende_;
;

// Implementierung von 'loesche'
template <class T> void liste<T>::loesche(const T& i)

  // Wenn die Liste leer ist...
  if (linkes_ende_ == 0)

    // ...kann sie auch nicht 'i' enthalten
    return;


  // Laufzeiger, um durch die Liste zu gehen
  knoten* laufzeiger = linkes_ende_;

  // Hilfszeiger, um Element zu loeschen
  knoten* delzeiger;

  // Gehe die Liste durch...
  while(laufzeiger != rechtes_ende_)

    // ...vergleiche mit 'i'...
    if (laufzeiger->element == i)

      // ...und loesche im Erfolgsfall.

      // Befindet sich das Element am linken Rand?
      if(laufzeiger->links != 0)

        // nein, also loesche "normal"
        delzeiger = laufzeiger;
        laufzeiger = laufzeiger->rechts;
```

```
      delzeiger->links->rechts = laufzeiger;
      laufzeiger->links = delzeiger->links;
      delete delzeiger;

    else

      // ja, also mache naechstes Element zum ersten Element
      linkes_ende_ = laufzeiger->rechts;
      linkes_ende_->links = 0;
      delete laufzeiger;
      laufzeiger = linkes_ende_;


  else

    // anderenfalls weiter zum naechsten Listenelement
    laufzeiger = laufzeiger->rechts;



// Wenn 'laufzeiger' also 'rechtes_ende_' ist, muss noch dieses
// Element ueberprueft werden...
if (laufzeiger->element == i)

  // ...und wenn noetig geloescht werden

  // Ist nur ein Element vorhanden, d.h. linkes_ende_ == rechtes_ende_?
  if (laufzeiger->links != 0)

    // nein, also wird vorletztes Element zum letzten Element
    rechtes_ende_ = laufzeiger->links;
    rechtes_ende_->rechts = 0;
    delete laufzeiger;

  else

    // ja, also loesche das Element...
    delete laufzeiger;

    // ...und setze nach Konvention die Endzeiger auf 0
    linkes_ende_ = 0;
    rechtes_ende_ = 0;
```

Aufgabe 3. Klassendeklaration (10 Bonuspunkte)

```
class Quadrat: public Form

public:

  // Default-Konstruktor
  Quadrat();

  // Konstruktor, der die Seitenlaenge und den Farbwert setzt
  Quadrat(double seitenlaenge, Farbe farbwert);

  // Destruktor
  ~Quadrat();

  // liefert den Flaecheninhalt
  double flaecheninhalt();

  // liefert den Umfang
  double umfang();

protected:

  // speichert die Seitenlaenge
  double seitenlaenge;
;

//------------------- Implementierungen -------------------

Quadrat::Quadrat()

  // setze Seitenlaenge und Farbwert auf etwas Sinnvolles
  seitenlaenge = 0.0;
  farbwert = blau;


Quadrat::Quadrat(double seitenlaenge, Farbe farbwert)

  // setzen der Seitenlaenge und des Farbwertes
  Quadrat::seitenlaenge = seitenlaenge;
  Quadrat::farbwert = farbwert;


Quadrat::~Quadrat()

  // tue nichts
```

```
double Quadrat::flaecheninhalt()

  // Formel fuer den Flaecheninhalt
  return seitenlaenge*seitenlaenge;


double Quadrat::umfang()

  // Formel fuer die Seitenlaenge
  return 4.0*seitenlaenge;
```

Aufgabe 4. $\mathcal{O}$-Notation (5 Bonuspunkte)
Finden Sie eine Anordnung der folgenden Funktionen, so dass gilt $f_i \in \mathcal{O}(f_{i+1}) \forall i$ und geben Sie für jedes Paar f_i, f_{i+1} eine kurze Begründung Ihrer Anordnung an.

$$n\ln(n), 4^{\log_2(n+1)}, e^n, n!$$

Begründungen: (i) Zuerst mal formen wir $4^{\log_2(n+1)}$ um: $4^{\log_2(n+1)} = (2^2)^{\log_2(n+1)} = 2^{\log_2((n+1)^2)} = (n+1)^2$

$$\begin{aligned}\lim_{n\to\infty} \frac{n\ln(n)}{(n+1)^2} &= \lim_{n\to\infty} \frac{n\ln(n)}{n^2+2n+1} \\ &\underset{=}{\text{L'H}} \lim_{n\to\infty} \frac{1+\ln(n)}{2n+2} \\ &\underset{=}{\text{L'H}} \lim_{n\to\infty} \frac{\frac{1}{n}}{2} = 0\end{aligned}$$

(ii)

$$\begin{aligned}\lim_{n\to\infty} \frac{(n+1)^2}{e^n} &= \lim_{n\to\infty} \frac{n^2+2n+1}{e^n} \\ &\underset{=}{\text{L'H}} \lim_{n\to\infty} \frac{2n+2}{e^n} \\ &\underset{=}{\text{L'H}} \lim_{n\to\infty} \frac{2}{e^n} = 0\end{aligned}$$

(iii)

$$\begin{aligned}\lim_{n\to\infty} \frac{e^n}{n!} &= \lim_{n\to\infty} \frac{e\times e\times\cdots\times e}{n\times(n-1)\times\cdots\times 1} \\ &= 0\end{aligned}$$

Aufgabe 5. Laplace Wahrscheinlichkeit (5 Bonuspunkte)

Man betrachtet den Laplace-Wahrscheinlichkeitsraum für die nächsten 4 Spiele. Denn spätestens nach 4 Spielen ist ein Gewinner festgestellt! $\Rightarrow |\Omega| = 2^4$
Sei A das Ereignis, dass Spieler 2 gewinnt. Spieler 2 gewinnt genau dann, wenn er 3 der nächsten 4 Spiele oder alle 4 Spiele gewinnt.
$\Rightarrow |A| = (\binom{4}{3}) + (\binom{4}{4}) = 5$
$\Rightarrow P(A) = \frac{5}{16}$
Damit erhält Spieler 1 $\frac{11}{16}$ und Spieler 2 $\frac{5}{16}$ des Betrags.

Aufgabe 6. Substitutionsmethode (5 Bonuspunkte)
Lösen Sie die folgende Rekurrenzgleichung mit Hilfe der Substitutionsmethode, d.h., geben Sie ein $f(n)$ an mit: $T(n) \in \Theta(f(n))$. Beweisen Sie: $T(n) \in \mathcal{O}(f(n))$. Das auch gilt $T(n) \in \Omega(f(n))$, müssen Sie hier ausnahmsweise *nicht* beweisen!

$$T(n) = 2T\left(\frac{n}{3}\right) + n^3$$

Z.B. der Rekursionsbaum führt auf die Behauptung:

$$T(n) \in \Theta(n^3)$$

Wir zeigen $T(n) \leq cn^3 \Rightarrow T(n) \in \mathcal{O}(n^3)$ für ein geeignetes $c > 0$:
Behauptung: $T(n) \leq cn^3$
Beweis: I.A.: $n = 1 : T(1) = 1 \leq c1^3 \forall c \geq 1$
I.S.:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{3}\right) + n^3 \\ &\underset{\text{I.A.}}{\leq} 2\left(\frac{cn^3}{27}\right) + n^3 \\ &= c\frac{2n^3}{27} + n^3 \\ &= cn^3\left(\frac{1}{c} + \frac{2}{27}\right) \\ &\leq cn^3 \forall c \geq \frac{27}{25} = 1 + \frac{2}{25} \end{aligned}$$

Aufgabe 7. Mastertheorem (5 Bonuspunkte)
Zeigen Sie für jede der folgenden Rekurrenzen ob das Mastertheorem zur Lösung angewandt werden kann. Bestimmen Sie in den Fällen, in denen sich das Theorem anwenden lässt, die Lösung der Rekurrenz.

- $T(n) = 64T(\frac{n}{8}) + 1042n^2 + 1000000n$ Wir bestimmen zunächst $\log_b(a) = \log_8(64) = 2$. Daraus folgt: $f(n) := 1042n^2 + 1000000n \in \Theta(n^{\log_b(a)}) = \Theta(n^2)$ und daher mit dem 2. Fall des Mastertheorems: $T(n) \in \Theta(\log_2(n)n^2)$
- $T(n) = 27T(\frac{n}{3}) + n^3\ln(n) + 42$ Hier gilt: $\log_b(a) = \log_3(27) = 3$, und damit: $f(n) = n^3\ln(n) + 42 \in \Omega(n^{\log_b(a)})$. Allerdings gilt auch:

$$\begin{aligned}\lim_{n\to\infty} \frac{f(n)}{n^3 n^{varepsilon}} &= \lim_{n\to\infty} \left(\frac{\ln(n)}{n^\varepsilon} + \underbrace{\frac{42}{n^{3+\varepsilon}}}_{0} \right) \\ &\underset{\text{L'H}}{=} \lim_{n\to\infty} \frac{\frac{1}{n}}{\varepsilon \frac{1}{n} n^\varepsilon} \\ &= \frac{1}{\varepsilon n^\varepsilon} \\ &= 0\end{aligned}$$

  Und damit gilt: $f(n) \in o(n^{3+\varepsilon}) \forall \varepsilon > 0$. Also ist das Mastertheorem hier *nicht* anwendbar!

Aufgabe 8: Bäume (10 Punkte)
Zwei Mengen A und B seien in den zugehörigen 2–4 - Bäumen T_A und T_B gespeichert. Beschreiben Sie einen Algorithmus, der den 2–4 - Baum $T_{A\cup B}$ der vereinigten Menge $A\cup B$ in Zeit $\mathcal{O}(|A|+|B|)$ berechnet. (Es reicht aus, wenn Sie Ihren Algorithmus in Pseudocode formulieren!) Begründen Sie Korrektheit und Laufzeit Ihres Algorithmus!
Hinweis: Zeigen Sie dazu zuerst, dass Sie den 2–4 - Baum einer schon sortierten Sequenz $a_1, \ldots, a_{n-1}$ in Zeit $\mathcal{O}(n)$ aufbauen können.

Zunächst überlegen wir uns – dem Hinweis entsprechend – wie wir schnell 2–4 - Bäume aus der sortierten Sequenz aufbauen können. Dazu stellen wir uns vor, dass wir von der Endsequenz aus einfach immer für je 2 Knoten einen Vaterknoten erstellen und mit diesen verbinden (ein möglicher überstehender Randknoten wird einfach mit an den zuletzt erstellten Vater gehängt, der dann 3 Kinder besitzt). Dies geht pro neu zu erzeugendem Vaterknoten offenbar in $\Theta(1)$, und da wir $\frac{n}{2}$ neue Väter erzeugen erhalten wir eine Laufzeit pro Ebene von $\Theta(n)$. Danach fahren wir rekursiv mit den eben erzeugten Vaterknoten fort, bis wir schliesslich an der neuen Wurzel ankommen. Da wir die Anzahl noch zu bearbeitender Knoten jedesmal offenbar halbieren, erhalten wir die Rekurrenz:

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n)$$

Diese Rekurrenz löst man leicht mit dem Mastertheorem (Fall 3), und es folgt: $T(n) \in \Theta(n)$.

Damit können wir den folgenden Algorithmus verwenden:

(a) Ausgeben der Sequenzen A und B in sortierter Reihenfolge (dies geht z.B. durch eine Breitensuche (Laufzeit —A— + —B—), bei der nur die letzte Ebene ausgegeben wird. Die Sequenzen A und B werden dann in linearer Zeit zu einer sortierten Sequenz $A \cup B$ zusammengefasst.)

(b) Nun bauen wir mit dem oben beschriebenen Verfahren den 2–4 - Baum der beschriebenen Sequenz auf. Da die Sequenz $|A|+|B|$ Elemente hat, läuft auch dieser Schritt in $\mathcal{O}(|A| + |B|)$.

Die Korrektheit des Algorithmus ist offensichtlich (denn wir erzeugen zunächst die korrekt sortierte Sequenz und bauen daraus mit dem oben beschriebenen Verfahren einen korrekten 2–4 - Baum), und die Laufzeit ergibt sich zu den verlangten $\mathcal{O}(|A| + |B|)$.

Aufgabe 9: Graphen (10 Punkte)
Zeigen Sie, dass jeder ungerichtete zusammenhängende Graph $G = (V, E)$ mindestens einen Knoten enthält, bei dessen Entfernung aus G der Graph zusammenhängend bleibt (wenn Sie den Knoten entfernen, entfernen Sie natürlich auch die zu ihm gehörigen Kanten aus dem Graphen). Entwickeln Sie einen Algorithmus, der einen solchen Knoten bestimmt und eine worst-case Laufzeit von $\mathcal{O}(|V| + |E|)$ aufweist (es reicht aus, wenn Sie diesen in Pseudocode formulieren!). Beweisen Sie die Korrektheit Ihres Verfahrens und begründen Sie kurz dessen Laufzeit.
Man macht sich leicht klar, dass die Blätter im DFS-Baum (wir haben hier nur einen Baum im Wald, da ja der Graph zusammenhängend ist) die Erreichbarkeit der anderen Knoten von G nicht beeinflussen können, denn beim Aufruf von DFS wurden *alle* Knoten des Graphen besucht, wobei keine Knoten von den Blättern des Predecessor-Graphen aus entdeckt wurden (genau das macht sie ja zu Blättern in diesem Graphen). Man kann also von allen anderen Knoten aus alle anderen Knoten erreichen, ohne auf die Blätter des Predecessor-Graphen zurückzugreifen. Damit können wir einfach DFS ausführen ($\mathcal{O}(|E|+|V|)$) und uns aus dem entstehenden Predecessor-Baum ein Blatt herausgreifen und aus dem Graphen löschen (wir erreichen dieses Blatt in in $\mathcal{O}(h)$, wobei h die Höhe des Predecessor-Graphen sei. Diese Höhe ist offenbar in $\mathcal{O}(|E|+|V|)$, und damit erhalten wir die gewünschte Laufzeit.