Dirk Praetorius,
Marcus Page,
Thomas Führer

Sommersemester 2013
07.05.2013

Übungen zur Vorlesung
Einführung in das Programmieren für TM

Serie 8

Aufgabe 8.1. Nicht jede Matrix $A \in \mathbb{R}^{n\times n}$ hat eine normalisierte LU-Zerlegung $A = LU$, d.h.

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ \ell_{21} & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ \ell_{n1} & \dots & \ell_{n,n-1} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & u_{n-1,n} \\ 0 & \dots & 0 & u_{nn} \end{pmatrix}.$$

Wenn aber A eine normalisierte LU-Zerlegung besitzt, so gilt

$$u_{ik} = a_{ik} - \sum_{j=1}^{i-1} \ell_{ij} u_{jk} \quad \text{für } i = 1, \dots, n, \quad k = i, \dots, n,$$

$$\ell_{ki} = \frac{1}{u_{ii}} \Big(a_{ki} - \sum_{j=1}^{i-1} \ell_{kj} u_{ji} \Big) \quad \text{für } i = 1, \dots, n, \quad k = i+1, \dots, n,$$

$$\ell_{ii} = 1 \quad \text{für } i = 1, \dots, n,$$

wie man leicht über die Formel für die Matrix-Matrix-Multiplikation zeigen kann. Alle übrigen Einträge von $L, U \in \mathbb{R}^{n\times n}$ sind Null. Schreiben Sie eine Funktion `computeLU`, die die LU-Zerlegung von A berechnet und zurückgibt. Dazu überlege man, in welcher Reihenfolge man die Einträge von L und U berechnen muss, damit die angegebenen Formeln wohldefiniert sind (d.h. alles was benötigt wird, ist bereits zuvor berechnet worden). Speichern Sie den Source-Code unter `computeLU.c` in das Verzeichnis `serie08`.

Aufgabe 8.2. Für eine Matrix $A \in \mathbb{R}^{m\times n}$ ist die transponierte Matrix $A^T \in \mathbb{R}^{n\times m}$ durch $(A^T)_{jk} = A_{kj}$ definiert. Schreiben Sie eine Funktion `transpose`, die die transponierte Matrix berechnet und zurückgibt. Speichern Sie den Source-Code unter `transpose.c` in das Verzeichnis `serie08`.

Aufgabe 8.3. In dieser Aufgabe soll der Vorteil von verketteten Listen gegenüber „normalen“ Arrays verdeutlicht werden: Die Einträge müssen nicht zusammenhängend im Speicher liegen. Legen Sie sich zu diesem Zweck im ersten Schritt einen großen ($\geq$ 1000000 Einträge) dynamischen `int`-Vektor `V` an. Im zweiten Schritt verlängern Sie den Vektor und fügen vorne(!) 10 neue Werte an. Messen Sie die Zeit die hierfür nötig ist. Orientieren Sie sich zur Zeitmessung an den Folien 85 ff.
Das Gleiche soll nun auch mittels verketteter Listen gemacht werden. Legen Sie sich auch hier eine Liste `L` gleicher Länge an und verlängern sie durch Einfügen von neuen Einträgen. Verwenden Sie hierfür die Struktur aus der Vorlesung. Messen Sie auch hier die CPU-Zeit. Was fällt Ihnen auf? Wie können Sie das Beobachtete erklären? Speichern Sie den Source-Code unter `ArrayListe.c` in das Verzeichnis `serie08`.
Hinweis: Eine `newList`-Funktion, die eine verkettete Liste beliebiger Länge mit konstanten Einträgen erzeugt kann hier hilfreich sein.

Aufgabe 8.4. In der letzten Aufgabe haben wir die Vorteile der verketteten Liste gegenüber dem C-Array kennengelernt. In dieser Aufgabe wollen wir uns nun Gedanken über die Nachteile machen. Eine Möglichkeit, einen Eintrag schnell in einem sortierten C-Array zu suchen, besteht in der binären Suche (vgl. Folie 83). Wieso lässt sich dieser Algorithmus nicht direkt bei sortierten (einfach) verketteten Listen anwenden? Welchen Aufwand hätte eine binäre Suche für verkettete Listen (worst-case)?

Aufgabe 8.5. Ein *binärer Baum* ist eine Datenstruktur, die dazu benutzt werden kann um effizient nach Daten zu Suchen. Solch ein Baum besteht dabei ähnlich wie eine verkettete Liste aus Knoten,

die einerseits Daten und andererseits Verbindungen zu anderen Knoten enthalten. Ein Binärbaum hat insbesondere höchstens zwei Verbindungen zu anderen Knoten, die man als *linkes* und *rechtes Kind* bezeichnet. Der Knoten, der keine *Eltern* besitzt wird als *Wurzel* bezeichnet.
Beispiel: In Abbildung 1 ist der Knoten mit Wert 15 das *linke Kind* der *Wurzel* und 20 das *rechte Kind* von 15.

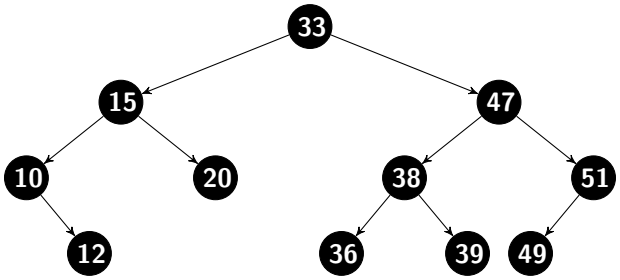

Abbildung 1: Ein Binärbaum, der die Suchbaumeigenschaft erfüllt.

Schreiben Sie einen Strukturdatentyp `node`, der einen `int`-Wert und zwei Pointer auf den linken und rechten Kindknoten speichern kann. Schreiben Sie weiters einen Strukturdatentyp `tree`, der einen Pointer auf die Wurzel des Baumes speichert. Schreiben Sie weiters Funktionen `node* newNode(int data, node* leftChild, node* rightChild)` und `tree* newTree()`, welche einen leeren Baum erzeugt, sowie die Zugriffsfunktionen `int getData(node* someNode)`, `node* getLeft(node* someNode)` und `node* getRight(node* someNode)`.
Anmerkung: Beachten Sie hier und in den weiteren Beispielen, dass wir `NULL` dazu verwenden um anzuzeigen, dass ein Knoten kein Kind an betroffener Stelle hat, oder ein Baum leer ist.

Aufgabe 8.6. Die Datenstruktur aus Aufgabe 8.5 stellt einen Binärbaum dar. Binäre *Suchbäume* sind Binärbäume, deren Knoten in einer gewissen Weise angeordnet sind. Konkret bedeutet das, dass für jeden Knoten K

- alle linken Kinder, sowie Enkelkinder usw. einen Wert speichern, der kleiner ist als der Wert von K selbst.
- alle rechten Kinder, sowie Enkelkinder usw. einen Wert speichern, der größergleich als der Wert von K selbst ist.

Schreiben Sie eine Funktion `void insert(tree* myTree, int content)`, die den übergebenen Wert `content` so in den Baum `myTree` einfügt, dass oben genannte *Suchbaumeigenschaft* erhalten bleibt. Man geht dabei wie folgt vor: Falls der Baum leer ist erstellen wir eine neue Wurzel mit `content` als Inhalt. Im anderen Fall überprüfen wir, ob der neue Knoten in den linken Teilbaum der Wurzel gehört. Trifft das zu, betrachten wir dieselbe Frage für den linken Teilbaum (rekursiv), ansonsten für den Rechten. Falls wir an dem Punkt angekommen sind, für den der entsprechende Teilbaum leer ist, so fügen wir einen neuen Knoten mit dem Inhalt `content` an dieser Stelle an.
Welchen Aufwand hat diese Operation in Abhängigkeit von der Tiefe des Baumes? Was bedeutet das für den Aufwand in Abhängigkeit von der Anzahl n der Elemente, wenn Sie davon ausgehen, dass der Baum balanciert ist? Speichern Sie den Source-Code unter `insert.c` in das Verzeichnis `serie08`.
Hinweis: Schreiben Sie eine rekursive Funktion `void insertAtNode(node* rootnode, int content)` und verwenden Sie:

```
void insert(tree* myTree, int content){
   if(myTree->root==NULL)
      myTree->root = newNode(content, NULL, NULL);
   else
      insertAtNode(myTree->root, content);
}
```

Aufgabe 8.7. Implementieren Sie eine binäre Suche für die Baumstruktur aus der Aufgabe 8.5. Orientieren Sie sich hierbei an Folie 83 der Vorlesung. Wie hoch ist der Aufwand in Abhängigkeit von der Tiefe des Baumes? Was bedeutet dies für den Aufwand in Abhängigkeit von der Anzahl der Einträge n wenn Sie davon ausgehen, dass der Baum balanciert ist.

Aufgabe 8.8. Ein binärer Suchbaum kann auch dazu verwendet werden um einen Vektor zu sortieren. Zuerst befüllt man den Baum schrittweise mit den Einträgen des Vektors. Dann kann der Baum rekursiv in der richtigen Reihenfolge durchlaufen werden. Die „richtige“ Reihenfolge ist dabei:
Für einen Knoten K:

- Gib auf die gleiche Weise (rekursiv!) den linken Teilbaum am Bildschirm aus.
- Gib den Wert des Knotens K am Bildschirm aus.
- Gib auf die gleiche Weise (rekursiv!) den rechten Teilbaum am Bildschirm aus.

Diese *Traversierung* wird auch *in-order* Traversierung genannt. Schreiben Sie eine Funktion `void fillTree(tree* T, Vector* v)`, welche alle Einträge des Vektors mittels `insert` in den Baum einfügt. Schreiben Sie ferner eine Funktion `void printSorted(tree* T)`, welche den erstellen Baum auf oben beschriebene Weise am Bildschirm ausgibt. Verwenden Sie die Struktur `Vector` aus der Vorlesung. Speichern Sie den Source-Code unter `treeSort.c` in das Verzeichnis `serie08`.
Hinweis: Auch hier ist es sinnvoll eine rekursive Funktion `void printSortedAtNode(node* rootnode)` zu schreiben. Verwenden Sie anschließend:

```
void printSorted(tree* myTree){
  printSortedAtNode(myTree->root);
}
```