Dirk Praetorius,
Michele Ruggeri

Wintersemester 2017/18
14.12./15.12.2017

Übungen zur Vorlesung
Einführung in das Programmieren für TM

Serie 8

Aufgabe 8.1. Schreiben Sie eine Struktur `cMatrix`, zur Speicherung von $(m \times n)$-Matrizen $A \in \mathbb{C}^{m\times n}$ mit komplexwertigen Koeffizienten. Benutzen Sie hierzu den Strukturdatentyp `cDouble` aus Aufgabe 7.1 und Aufgabe 7.2. Ferner schreibe man die zugehörigen Funktionen `newCMatrix`, `delCMatrix`, `getCMatrixM`, `getCMatrixN`, `getCMatrixCoeff`, `setCMatrixCoeff`. Testen Sie Ihre Implementierung entsprechend! Speichern Sie den Source-Code, aufgeteilt in Header-Datei `cmatrix.h` und `cmatrix.c`, in das Verzeichnis `serie08`.

Aufgabe 8.2. Eine komplexe quadratische Matrix $A \in \mathbb{C}^{n\times n}$ ist hermitesch, falls $A_{jk} = \overline{A_{kj}}$ für alle $j, k = 1, \dots, n$ gilt. Schreiben Sie eine Funktion `int isHermitian(cMatrix* A)`, die die Struktur einer Matrix $A \in \mathbb{C}^{n\times n}$ überprüft. Der Rückgabewert der Funktion ist entweder `1` (hermitesche Matrix) oder `0` (sonst). Benutzen Sie die Strukturdatentypen und die Funktionalitäten aus Aufgabe 7.1, Aufgabe 7.2und Aufgabe 8.1. Testen Sie Ihre Implementierung entsprechend! Welche Eigenschaft haben die Diagonaleinträge einer hermiteschen Matrix? Warum? Speichern Sie den Source-Code unter `ishermitian.c` in das Verzeichnis `serie08`.

Aufgabe 8.3. 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.

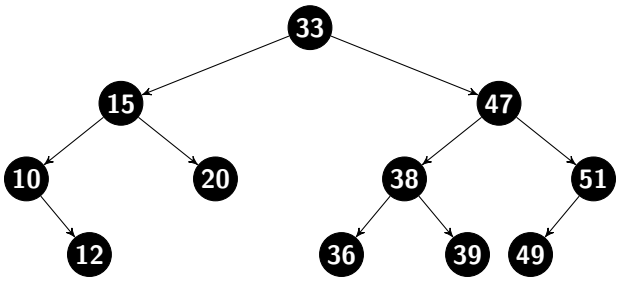

Abbildung 1: Ein Binärbaum, der die Suchbaumeigenschaft erfüllt.

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)`. Wie haben Sie Ihren Code auf Korrektheit getestet? Speichern Sie den Source-Code unter `baumstruct.c` in das Verzeichnis `serie08`.
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 8.4. Implementieren Sie eine binäre Suche für die Baumstruktur aus der Aufgabe 8.3. Orientieren Sie sich hierbei an Folie 85 der Vorlesung. Wie hoch ist der Aufwand in Abhängigkeit von der

Tiefe des Baumes? Wie haben Sie Ihren Code auf Korrektheit getestet? Speichern Sie den Source-Code unter `binsearch.c` in das Verzeichnis `serie08`.

Aufgabe 8.5. Die Datenstruktur aus Aufgabe 8.3 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? Wie haben Sie Ihren Code auf Korrektheit getestet? 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 8.6. 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. Wie haben Sie Ihren Code auf Korrektheit getestet? 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);
}
```

C++ Aufgaben

Aufgabe 8.7. Erstellen Sie eine Klasse `Kunde` für einen Kunden bei einer Bank. Diese Klasse soll den Namen des Kunden als `string`, den aktuellen Kontostand als `double` und eine PIN als `int` beinhalten. Implementieren Sie neben geeigneten `get` und `set` Methoden noch folgende Klassenmethoden

- `void printBalance()`
  gibt den aktuellen Kontostand am Bildschirm aus.
- `bool checkPIN()`
  liest einen PIN ein und überprüft, ob dieser korrekt ist.
- `void drawMoney()`
  überprüft zuerst den PIN, liest den abzuhebenden Betrag ein und gibt den neuen Kontostand am Bildschirm aus. Das Konto darf hierbei nicht überzogen werden. Geben Sie gegebenenfalls eine Warnung am Bildschirm aus.

Wie haben Sie Ihren Code auf Korrektheit getestet? Speichern Sie den Source-Code unter `kunde.{hpp,cpp}` in das Verzeichnis `serie08`.

Aufgabe 8.8. Schreiben Sie eine Klasse `Stoppuhr` welche zur Simulation einer Stoppuhr dienen soll. Die Stoppuhr bestehe dabei aus zwei Methoden. Wird die erste aufgerufen, so soll die Zeitmessung gestartet werden. Wird diese Methode nochmals gedrückt, wird die Zeitmessung gestoppt. Die zweite Methode dient dazu die Zeit wieder zurückzusetzen. Schreiben Sie dazu die Methoden `pushButtonStartStop` und `pushButtonReset`. Implementieren Sie weiters eine Methode, welche die verstrichene Zeit im Format `hh:mm:ss.xx` ausgibt (Beträgt die gemessene Zeit also zwei Minuten so soll `00:02:00.00` ausgegeben werden). Sie können diese Stoppuhr nun dazu verwenden Zeitmessungen durchzuführen. Wie haben Sie Ihren Code auf Korrektheit getestet? Speichern Sie den Source-Code unter `stoppuhr.{hpp,cpp}` in das Verzeichnis `serie08`.
Hinweis: Verwenden Sie den Datentyp `clock_t` und die Funktion `clock()` aus der Bibliothek `time.h`. Vermutlich ist es auch sinnvoll, eine Variable `isRunning` vom Typ `bool` einzuführen. Beim Aufrufen der ersten Methode wird diese Variable entweder von `false` auf `true` gesetzt oder umgekehrt.