Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion
CAU Kiel, Informatik und Prakt. Mathematik
Prof. Dr. M. Hanus, Dr. F. Huch

24118 Kiel
Christian-Albrechts-Platz 4
Tel.: 0431 / 880-7277

Programmierpraktikum zur Informatik I
Testat am 14.12., 10-13 Uhr, CAP4, Raum 709

In den folgenden Praktikumsaufgaben wollen wir ein Adressbuch implementieren. Hierfür benötigen wir zunächst eine Datenbank, welche Funktionen zum Ändern und Zugreifen der Einträge im Adressbuch zur Verfügung stellt.
Als effiziente Implementierung der Datenbank verwenden wir die bereits in der Vorlesung eingeführten sortierten, binären Bäume, im folgenden Suchbäume (siehe effiziente Implementierung von Mengen).

Hinweis: Um zu verhindern, dass Sie Ihre Implementierung mit einer **falschen/unschönen Datenstruktur** vornehmen, sollten Sie alle entworfenen Datenstrukturen **frühzeitig** bei einem **Praktikumstermine** einem Betreuer zeigen.

Aufgabe 3
Implementieren Sie zunächst einen abstrakten Datentyp (ADT) `Binary-Tree` mit Einträgen in den Knoten. Definieren Sie folgende Konstruktoren:

- `emptyTree` erzeugt den leeren Baum,
- `node` erzeugt einen Baumknoten und
- `leaf` zum einfacheren erzeuge eines Blattes (`leaf` ist also nur eine Abkürzung für einen `node` mit leeren Unterbäumen).

Zum Umgang mit `Binary-Tree`s sollen außerdem noch folgende Operationen implementiert werden:

- `(treeToList tree)` erzeugt aus einem Baum eine Liste. Hierbei soll für jeden Teilbaum gelten, dass die Elemente des linken Teilbaumes vor dem Wert der Wurzel stehen und die Elemente des rechten Teilbaumes dahinter.
- `(mapTree function tree)` wendet eine Funktion `function` auf alle Elemente des Baumes `tree` an und liefert diesen Baum als Ergebnis.
- `(search predicate tree)` sucht in dem Baum `tree` nach den Elementen, für die die Funktion `predicate` zu `#t` auswertet. Das Ergebnis dieser Funktion sei die Liste dieser Elemente.

Aufgabe 4
Definieren Sie den ADT `Entry` (als Scheme-Struktur), welchen wir in der folgenden Aufgabe für die Einträge des Suchbaums verwenden werden. Ein `Entry` besteht aus einem Schlüssel und einem zugehörigen Wert. Im Kontext einer Adressdatenbank kann man sich unter dem Schlüssel z.B. den Namen und unter dem Wert die Adresse vorstellen.

Aufgabe 5
In dieser Aufgabe soll der ADT `Binary-Tree` mit Hilfe des ADT `Entry` zu einem ADT `Search-Tree` verfeinert werden. Definieren Sie den Konstruktor `(emptySearchTree less equal)` zur Konstruktion eines leeren Suchbaums. In der Datenstruktur des Suchbaums sollen auch die Vergleichsfunktionen abgelegt werden, welche später zum Arbeiten mit dem Suchbaum benötigt werden und die Schlüssel der `Entry`s miteinander vergleichen.
Implementieren sie folgende Operationen:

- `(replaceTreeInSearchTree t sT)` ersetzt den `Binary-Tree` in `sT` durch `t`. Diese Hilfsfunktion werden Sie bei den weiteren Definitionen häufiger verwenden können.

  folgenden Funkt

- `(insert e sT)` sortiert den Eintrag `e` in den Suchbaum `sT` ein. Ist der Schlüssel bereits belegt, wird der alte Eintrag überschrieben.

- `(lookup key sT)` sucht den unter dem Schlüssel `key` abgelegten Wert im Suchbaum `sT`. Kommt `key` nicht im `sT` vor, wird der Wert `null` zurückgeliefert.

- `(modify key fun sT)` sucht den unter dem Schlüssel `key` abgelegten Wert im Suchbaum `sT` und wendet die Funktion `fun` darauf an. Neben dem neuen Suchbaum, soll die Funktion auch den veränderten Eintrag zurückliefern (z.B. als Paar). Kommt `key` nicht im `sT` vor, wird der Wert `null` zurückgeliefert.

- `(delete key sT)` löscht den Schlüssel `key` und den zugehörigen Wert im Suchbaum `sT`. Achten Sie auf eine effiziente Implementierung, d.h. es soll möglichst viel Struktur von `sT` erhalten bleiben. Also keine Verwendung von `mergeTree` oder Umwandlung in Listen.

- `(mergeTree sT1 sT2)` fügt zwei Suchbäume (welche gleiche Einträge haben sollten) zu einem Suchbaum zusammen. Ist hier eine ähnlich effiziente Implementierung wie bei `delete` möglich?

- `(addListToTree l sT)` fügt die Einträge der Liste nach einander zum Suchbaum `sT` hinzu.

- `isSorted sT` welche Testet, ob ein gegebener Suchbaum sortiert ist. Diese Funktion benötigen Sie um die anderen Funktionen zu testen. Überlegen Sie sich für jede Funktion einen Test, in dem die Funktion mindestens 50 mal verwendet wird und die Sortiertheitseigenschaft für die Ergebnisssuchbäume geprüft wird. Hilfreich kann in diesem Zusammenhang die Verwendung der vordefinierten Scheme-Funktion `random` sein.

Erweitern Sie auch die Operationen des ADT `Binary-Tree` auf entsprechende sinnvolle (Begründung!) Operationen des ADT `Search-Tree`.

Hinweis: Die Vergleichsfunktionen werden bei der Generierung des leeren Suchbaums festgelegt. So können in einem Programm sortierte Bäume mit unterschiedlichen Schlüsseltypen (z.B. Zahlen und Strings) verwendet werden.