TU Ilmenau, Fakultät für Informatik und Automatisierung
FG Komplexitätstheorie und Effiziente Algorithmen
Univ.-Prof. Dr. M. Dietzfelbinger, Philipp Schlag, Stefan Walzer, Alexander Jeuk
https://www.tu-ilmenau.de/iti/lehre/lehre-ss-2020/aud/

TECHNISCHE UNIVERSITÄT
ILMENAU

Algorithmen und Datenstrukturen SS 2020 – Praktikum 2

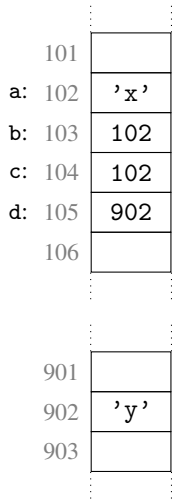
Zur Bearbeitung zwischen 15. Juni - 19. Juni (KW 25)

In diesem Praktikum beschäftigen wir uns mit binären Suchbäumen. Wiederholen Sie dazu sorgfältig Kapitel 4 der Vorlesung. Die Suchbaumanfragen sind dabei *zufällig* generiert. Insbesondere gilt Satz 4.2.1 der Vorlesung, der entstehende Baum wird also sehr wahrscheinlich nur logarithmische Tiefe haben.

Aufgabe 0 (Zeiger, `new` und `delete`, `struct`, Repräsentation eines binären Suchbaums) optional

Falls Sie sich noch etwas mit Zeigern und `structs` vertraut machen möchten, können Sie sich zunächst dieser Aufgabe widmen.

(a) Mittels der Anweisung `char a = 'x'` erzeugt man eine Variable `a` vom Typ `char` und dem Zeichen `'x'` als Inhalt. Über deren Namen `a` kann man auf dessen Inhalt zugreifen: `a == 'x'`. Im nebenstehenden Beispiel wird dieser an Adresse 102 abgespeichert. Diese Adresse liefert der `&`-Operator: `&a == 102`.
Einen Zeiger `b` auf eine Variable vom Typ `char` erzeugen wir mit Hilfe von `*` durch `char *b = &a`. Der Inhalt dieses Zeigers ist dann die Adresse von `a`: `b == 102`. Die Zeigervariable besitzt natürlich auch selber eine Adresse: `&b == 103`. Mit dem `*`-Operator kann man auf den Inhalt der Variablen/Speicherzelle, auf die der Zeiger zeigt, lesend und schreibend zugreifen: `*b == 'x'`.[1]
Natürlich kann man Zeiger auch kopieren: `char *c = b`. Damit zeigt der Zeiger `c` auf genau dieselbe Position wie Zeiger `b`: `c == 102` und `*c == 'x'`, aber `&c == 104`.



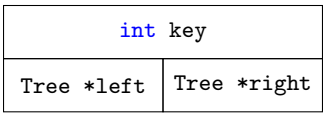
In C++ wird der Speicher in verschiedene Bereiche unterteilt. Wie oben definierte Variablen landen dabei im Stapelspeicher (Stack), ihr zugehöriger Speicherplatz wird automatisch wieder freigegeben, wenn ihr Gültigkeitsbereich verlassen wird. Größere Objekte oder solche, die über den aktuellen Gültigkeitsbereich hinaus existieren sollen, sollten hingegen explizit mittels `new` im Haldenspeicher (Heap) angelegt und abschließend mit `delete` wieder gelöscht werden.

Bei der Anweisung `char *d = new char{'y'}` wird mittels des `new`-Operators auf dem Heap für ein `char` reserviert und mit dem Inhalt `'y'` initialisiert. Rückgabe ist ein Zeiger auf diesen Speicherbereich (d. h. die Adresse 902), der dann als Zeiger `d` gespeichert wird. Die Zeigervariable `d` wird dabei wiederum im Stack abgelegt. Da im Heap benutzter Speicher nicht automatisch freigegeben wird (z. B., wenn kein Zeiger mehr auf diese Speicherposition zeigt), muss man sich selber darum kümmern. Das ist mit dem `delete`-Operator möglich, der auf einem Zeiger aufgerufen wird und den zugehörigen Speicherbereich wieder freigibt: `delete d`.[2]

[1] Insbesondere gilt auch `b == &*b`: `b` ist genau die Adresse (`&`) des Speicherbereichs (`*`), auf den `b` zeigt.

[2] Danach zeigt `d` zwar immer noch auf die Position 902, beim Zugriff `*d` oder einem wiederholten `delete d` kommt es allerdings zu einem Speicherzugriffsfehler. Deshalb ist es sinnvoll, nach der Freigabe den Zeiger auf den Nullzeiger `nullptr` zu setzen: `d = nullptr`. Bei erneutem `delete d` passiert dann einfach nichts, und Dereferenzierungen `*d` sollte man sowieso nur dann verwenden, wenn man weiß, dass `d` nicht der Nullzeiger ist und auf einen gültigen Speicherbereich zeigt.

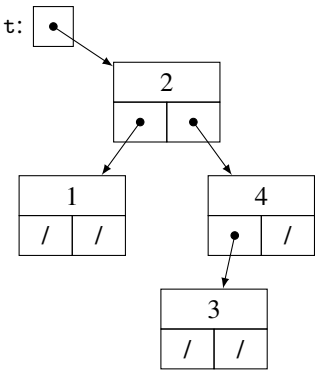
(b) Jeder Knoten eines binären Suchbaumes (ohne Datenelemente) besteht aus einem Schlüssel und zwei Zeigern zu einem linken bzw. rechten (evtl. leeren) Unterbaum. Im C++-Programm `bintree` zu dieser Aufgabe definieren wir uns dafür eine Struktur `Tree`, die ähnlich wie Datentypen und Klassen verwendet wird:




```
struct Tree { Tree *left; int key; Tree *right; };
```

Den nebenstehenden BSB können wir dann wie folgt in drei Schritten explizit konstruieren:

```
1  Tree *t = new Tree{nullptr,2,nullptr};
```



Zunächst erzeugen wir die Wurzel mit initialen Werten `left == nullptr`, `key == 2` und `right == nullptr` entsprechend der Reihenfolge in der Definition der Struktur. `t` ist dann ein Zeiger auf diese Wurzel und repräsentiert den BSB im Programm. `nullptr` repräsentiert den leeren BSB.

```
2  (*t).left = new Tree{nullptr,1,nullptr};
```

Auf die Komponente `left` der Wurzel `*t` können wir mittels des `.`-Operators zugreifen: `(*t).left`. Nach der obigen Operation zeigt dieser Zeiger also auf einen neu erzeugten Knoten mit Schlüssel 1.

```
3  t->right = new Tree{new Tree{nullptr,3,nullptr},4,nullptr};
```

Dank des `->`-Operators kann man statt `(*t).right` auch kurz `t->right` schreiben. Ebenso können wir ineinander verschachtelt neue Knoten erzeugen und die zurückgelieferten Zeiger direkt verwerten. Die obigen drei Anweisungen lassen sich deshalb wie folgt zusammenfassen:

```
Tree *t = new Tree{new Tree{nullptr,1,nullptr},2,new Tree{new Tree{nullptr,3,nullptr},4,nullptr}};
```

(c) In der Definition der Struktur `Tree` sehen Sie noch zwei spezielle Methoden. Der Konstruktor

```
Tree(Tree *l, int k, Tree *r) : left(l), key(k), right(r) {
  ++livingTrees;
}
```

wird immer dann aufgerufen, wenn eine neue Instanz von `Tree` erzeugt wird, wie es beispielsweise durch `new Tree{nullptr,5,nullptr}` geschieht. Durch `: left(l), key(k), right(r)` werden die drei übergebenen Werte zunächst direkt in die drei `Tree`-Komponenten übernommen. Anschließend wird die mit `0` initialisierte globale Variable `int livingTrees` um eins erhöht.

Umgekehrt wird der Destruktor

```
~Tree() {
  delete left; delete right;
  --livingTrees;
}
```

immer dann aufgerufen, wenn eine `Tree`-Instanz gelöscht wird, z. B. mittels des `delete`-Operators. Hier sorgen wir also dafür, dass zusätzlich zu dem aktuellen Knoten auch automatisch die beiden Unterbäume rekursiv gelöscht werden. **Achtung!** Wenn man nur einen einzelnen Knoten, nicht aber dessen Unterbäume löschen möchte, muss man sich deshalb etwas geschickter anstellen. Eine Möglichkeit dazu finden Sie im Beispielprogramm zu dieser Aufgabe.

Tipp: Da zusätzlich der Zähler `livingTrees` um eins verringert wird, gibt dieser also zu jedem Zeitpunkt im Programmablauf die aktuelle Anzahl der im Speicher vorhandenen `Tree`-Instanzen an. Damit stellt diese Variable ein einfaches Hilfsmittel dar, um beim Debuggen Speicherlecks zu entdecken.

Aufgabe 1 (Binäre Suchbäume, rekursiver Ansatz)

(a) Schreiben Sie ein Programm `bintree`, das die Suchbaumoperationen `insert`, `lookup` und `delete` mittels `extractMin` rekursiv implementiert. Schlüssel sind natürliche Zahlen im `int`-Wertebereich, und der Einfachheit halber lassen wir die Datenelemente des Baumes weg, d. h., eine `lookup`-Operation soll nur zurückgeben, ob ein Element im Baum enthalten ist (`1`) oder nicht (`0`).

Die Eingabedatei enthält einen Befehl pro Zeile. Das erste Zeichen jeder Zeile ist `i`, `l` oder `d` und steht für `insert`, `lookup` oder `delete`. Danach kommt der Schlüssel, der eingefügt, nachgeschlagen oder entfernt werden soll. Führen Sie die Befehle nacheinander aus, startend mit dem leeren Baum. Geben Sie nach jeder `insert` und `delete` Operation den aktuellen Baum aus. Geben Sie nach jeder `lookup` Operation dessen Ergebnis, also `0` bzw. `1` aus.

Listing 1: `sample.in`

```
1  i 3
2  i 7
3  i 7
4  i 5
5  i 9
6  i 2
7  i 6
8  l 5
9  l 2
10 l 8
11 l 1
12 l 7
13 d 1
14 d 5
15 d 7
16 d 3
17 l 7
```

Listing 2: `sample.ans`

```
1  (-,3,-)
2  (-,3,(-,7,-))
3  (-,3,(-,7,-))
4  (-,3,((-,5,-),7,-))
5  (-,3,((-,5,-),7,(-,9,-)))
6  ((-,2,-),3,((-,5,-),7,(-,9,-)))
7  ((-,2,-),3,((-,5,(-,6,-)),7,(-,9,-)))
8  1
9  1
10 0
11 0
12 1
13 ((-,2,-),3,((-,5,(-,6,-)),7,(-,9,-)))
14 ((-,2,-),3,((-,6,-),7,(-,9,-)))
15 ((-,2,-),3,((-,6,-),9,-))
16 ((-,2,-),6,(-,9,-))
17 0
```

Testen Sie Ihre Implementierung mit `sample.in` (und bei Bedarf mit dem zusätzlichen Ein-/Ausgabepaar `special.in/ans`) und reichen Sie sie für das Problem **p2-a1-bsb** im DOMjudge-System ein.

Hinweise:

(1) Sie dürfen das C++-Schlüsselwort `delete` nicht für eigene Bezeichner verwenden. Nennen Sie sie stattdessen beispielsweise `erase`.

(2) Den Materialien liegt eine Datei `bintree.cpp` bei, die Sie als Ausgangspunkt für Ihre Implementierung nutzen können. In dieser wird u. a. `struct Tree` wie in Aufgabe 0 beschrieben definiert und der Stream-Operator `<<` überladen, so dass `cout` mit Zeigern auf Bäume wie gewünscht umgehen kann. Außerdem werden geeignete Signaturen für die zu implementierenden Operationen vorgeschlagen.[3]

(3) Wenn Sie nicht weiterkommen, benutzen Sie die Vorlesungsfolien als Vorbild. Die Fallunterscheidungen dort sollten sich auch in Ihrem Programm wiederfinden.

(b) **Optional:** Achten Sie darauf, sämtlichen Speicher, den Sie mit `new` anfordern, auch wieder mit `delete` freizugeben! Sie können das DOMjudge-Problem **p2-a1-bsb** (oder die lokalen Eingabedateien) zur Überprüfung

[3] `Tree *insert(Tree *T, const int key)` bedeutet beispielsweise, dass die `insert`-Methode einen Zeiger `T` auf die Wurzel eines BSBs und einen in diesen BSB einzufügenden Schlüssel `key` als Eingabe erhält. Der Rückgabewert ist dann ein Zeiger auf die (möglicherweise neue) Wurzel des BSBs, in den `key` eingefügt wurde.

verwenden, indem Sie den in Aufgabe 0 beschriebenen Ansatz mit der Variablen `livingTrees` umsetzen und zusätzlich die beiden folgenden Ergänzungen vornehmen:

- Binden Sie am Anfang Ihres Programmes mittels `#include <cassert>` die Header-Datei `cassert` ein.
- Löschen Sie am Ende Ihres Programmes, kurz bevor die `main`-Methode verlassen wird, den evtl. noch vorhandenen Rest-BSB mit `delete t`. Führen Sie dann den Befehl `assert(livingTrees == 0)` aus. Wenn `livingTrees != 0` ist, kommt es zu einem Programmabbruch, im DOMjudge-System erhalten Sie dann RUN-ERROR als Rückmeldung.

Aufgabe 2 (Laufzeitmessungen)

(a) Um die Ausgabe für große Eingabedateien klein zu halten, gibt es einen vierten Befehlstyp `p` (`print`). Der Befehl `p 0` bedeutet, dass im Folgenden auf eine Ausgabe des aktuellen Baums nach jedem `insert` und `delete` verzichtet werden soll. Die Antworten auf `lookup`-Befehle sollen nach wie vor ausgegeben werden. Jeder andere `p`-Befehl, z. B. `p 1`, soll die vollständige Ausgabe wieder aktivieren.

Testen Sie Ihre Implementierung mit `sample.in` und reichen Sie sie für das Problem **p2-a2-bsb-p** im DOMjudge-System ein.

(b) Nun soll Ihr Programm `bintree` auf einer großen Eingabe `large.in` ausgeführt und die Laufzeit gemessen werden. Wir unterscheiden dabei zwischen „CPU time“ (von dem Programm beanspruchte Prozessorzeit) und „wall-clock time“ (insgesamt vergangene Zeit während der Ausführung des Programmes). Neben `p 0` enthält die Datei insgesamt 1.6 Millionen Befehle.

Unter **WSL/Linux/macOS** können wir beide Werte leicht über den `time`-Befehl im Terminal ermitteln. Führen Sie dazu den folgenden Befehl im Unterordner `Aufgabe2` im Terminal aus:

```
time ./bintree < large.in > /dev/null
```

Das Programm `bintree` liest die Datei `large.in` mittels `cin` als Eingabe, durch `> /dev/null` wird die über `cout` erfolgte Ausgabe verworfen. Währenddessen werden die uns interessierenden Zeiten gemessen und zum Schluss ausgegeben, z. B.

```
real    0m2.019s
user    0m1.547s
sys     0m0.453s
```

Dabei gibt `real` die „wall-clock time“ an und `user` die „CPU time“.

Unter **Windows** oder als **Alternative unter WSL/Linux/macOS** können Sie die Messung der Zeiten auch **direkt im C++-Programm** vornehmen. In den Materialien finden Sie dazu eine Header-Datei `Timer.h`.[4] Binden Sie diese mittels

```
#include "Timer.h"
```

in Ihr Programm ein. Ganz am Anfang der `main`-Methode beginnen Sie das Messen der Zeiten mit einem `Timer t` dann mittels

```
Timer t; t.start();
```

Am Ende der `main`-Methode beenden Sie dann die Messung und geben die Zeiten aus:

```
t.stop(); clog << t.getTime() << endl;
```

[4] Die konkrete Implementierung unterscheidet sich zwischen Windows und WSL/Linux/macOS, die Verwendung bleibt aber gleich.

`clog` ist (genauso wie `cout` und `cerr`) ein Ausgabestream. Wenn wir die Standardausgabe (von `cout`) beim Ausführen des Programmes umleiten (z. B. bei der DIFF-Konfiguration in eine Datei `sample.out` mittels `> sample.out`) oder komplett verwerfen (`> /dev/null` unter WSL/Linux/macOS bzw. `> nul` unter Windows), wird die Ausgabe von `clog` (und die von `cerr`) trotzdem in der ausführenden Konsole angezeigt.

Kompilieren Sie nun Ihr erweitertes Programm.

- Unter WSL/Linux/macOS führen Sie es dann im Terminal mittels

  ```
  ./bintree < large.in > /dev/null
  ```

  aus.
- Unter Windows muss es in der Eingabeaufforderung

  ```
  .\bintree < large.in > nul
  ```

  heißen. Wenn Sie die PowerShell verwenden, nutzen Sie stattdessen

  ```
  cmd /c ".\bintree < large.in > nul"
  ```

Die Ausgabe sieht dann beispielsweise wie folgt aus:

```
cpu:        1.96875s
wall-clock: 1.9795s
```

Aufgabe 3 (Iterativer Ansatz) optional

Ersetzen Sie Ihre rekursiven `insert`, `lookup` und `delete` Funktionen jeweils durch eine iterative Implementierung. Stellen Sie sicher, dass Ihr Programm nach wie vor korrekt funktioniert, indem Sie es auf Ihrem eigenen System testen und für das DOMjudge-Problem **p2-a2-bsb-p** einreichen, und führen Sie erneut eine Zeitmessung durch.

Bemerkung: Es kann sein, dass der Unterschied zwischen iterativer und rekursiver Implementierung recht gering ausfällt. Das kann auch daran liegen, dass es dem Compiler selbstständig gelingt, den rekursiven Aufruf „weg zu optimieren“.