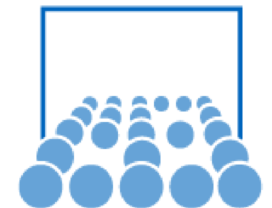
Kompaktkurs
Einführung in die Programmierung

10. Binärbäume

Stefan Zimmer

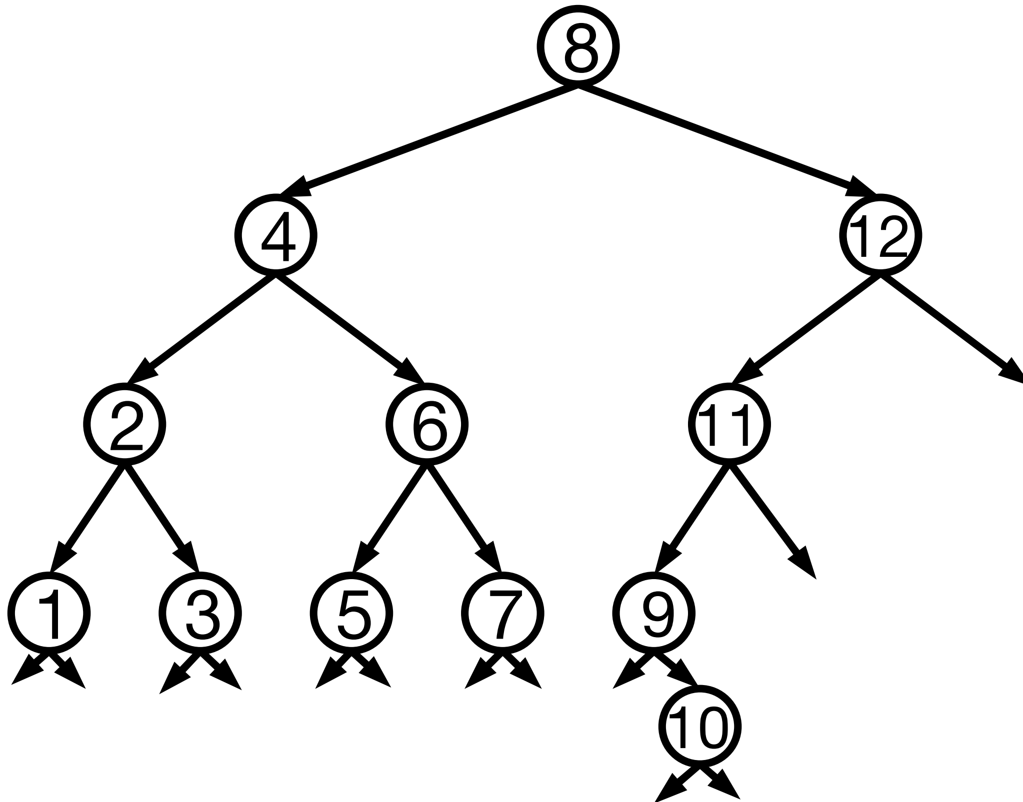
12.3.2010



Binärbäume

- Als Beispiel für eine interessantere dynamische Datenstruktur sehen wir uns jetzt *Binärbäume* an
- Ein Binärbaum wird rekursiv definiert:
  - Er ist leer oder
  - besteht aus
    - einem *Knoten* (die *Wurzel* des Binärbaums) mit einem Inhalt (der *Beschriftung*) und
    - zwei (möglicherweise leeren) Binärbäumen: dem *rechten Sohn* und dem *linken Sohn*
- Die Beschriftung (die im Baum gespeicherten Werte) ist beliebig – wir werden im Beispiel (wie schon im Listenbeispiel) einen `int`-Wert verwenden

Binärbäume: Beispiel



- Leere Binärbäume sind hier durch einen Pfeil in's Nichts angedeutet (später werde ich sie einfach weglassen)
- Die Beschriftungen brauchen (noch) keinen Regeln zu folgen

- Tiefe eines Knotens: Abstand von der Wurzel (des Gesamtbaums, hier '8') plus 1: '8' hat Tiefe 1, '10' hat Tiefe 5
- Tiefe des Baums: maximale Tiefe eines Knotens (hier: 5)

Realisierung in C

- Die Realisierung in C sieht ganz ähnlich aus wie bei den Listen, nur dass es statt eines Nachfolgers jetzt zwei Söhne gibt:

```
struct knoten {
    // Beschriftung
    int x;
    // Linker und rechter Sohn
    struct knoten *l, *r;
};
```

- Ein Baum wird nun repräsentiert durch
  - einen Zeiger auf seine Wurzel oder
  - den Nullzeiger
- Wer mag, kann sich einen Typnamen definieren:

```
typedef struct knoten *baum;
```

Baumwachstum

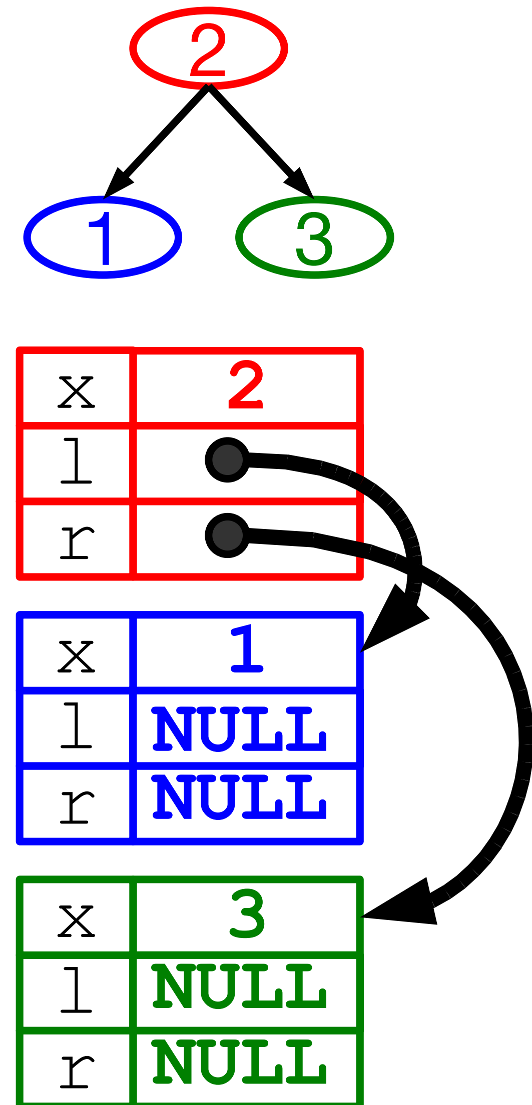
- Das Anlegen eines neuen Baums hat als Parameter die Wurzelbeschriftung sowie die beiden Söhne:

```
// Neuer Binärbaum mit Beschriftung xneu,
// und Söhnen links und rechts
struct knoten *neu(int xneu,
                   struct knoten *links,
                   struct knoten *rechts) {
   struct knoten *b = new struct knoten;
   b->x = xneu;
   b->l = links;
   b->r = rechts;
   return b;
}
```

Ein Beispielbaum

- Geschachtelte Aufrufe von neu sind nun schon bei kleinen Bäumen etwas unübersichtlich:

```
struct knoten *b;
b = neu(2,
        neu(1, NULL, NULL),
        neu(3, NULL, NULL));
```



- Vielleicht doch lieber mit Hilfsvariablen:

```
struct knoten *b1, *b2, *b;
b1 = neu(1, NULL, NULL);
b2 = neu(3, NULL, NULL);
b = neu(2, b1, b2);
```

Baum ausdrucken

- Auch das Ausdrucken des Baums entsteht schnell aus dem Listendrucken (eine passende Funktion zum Einrücken einer Druckzeile sei hier vorausgesetzt):

```
void drucke(struct knoten *b, int tiefe) {
    if (b) {
        einruecken(tiefe);
        std::cout << "Beschriftung: "
                  << b->x << '\n';
        einruecken(tiefe);
        std::cout << "Linker Sohn:\n";
        drucke(b->l, tiefe+1);
        einruecken(tiefe);
        std::cout << "Rechter Sohn:\n";
        drucke(b->r, tiefe+1);
    }
}
```

Suchbäume

- Binärbäume kann man verwenden, um Ordnung zu hal-
ten, d.h., Dinge so abzulegen, dass wir sie leicht wieder-
finden
- Das geht z.B. mittels *Suchbäumen*. Ein Suchbaum ist ein Binärbaum, bei dem für jeden Knoten gilt:
  - Alle Knotenbeschriftungen des linken Sohns (des Baums, nicht nur des Knotens!) sind kleiner oder gleich der Wurzelbeschriftung
  - Alle Knotenbeschriftungen des rechten Sohns sind größer oder gleich der Wurzelbeschriftung
- Insbesondere ist der leere Baum ein Suchbaum
- Ein größeres Beispiel für einen Suchbaum findet man auf Folie 3

Finden im Suchbaum

- Für die Suche nach einem Wert ist praktisch, dass wir mit einem Vergleich mindestens einen der beiden Söhne mit allen seinen Nachkommen ausschließen können
- Diese Funktion gibt einen Zeiger auf einen Baumknoten mit Beschriftung `y` zurück (`NULL`, wenn's keinen gibt):

```
struct knoten *finde(struct knoten *b,
                     int y) {
  if (b) {
    if      (y<b->x) return finde(b->l, y);
    else if (y>b->x) return finde(b->r, y);
    else return b;    // gefunden
  } else return NULL; // leerer Baum
}
```

- Vorausgesetzt ist, dass `b` auch wirklich ein Suchbaum ist

Test, ob Suchbaum

- Wer seinen Bäumen nicht traut, kann auch nachsehen, ob es sich um einen Suchbaum mit Einträgen im Bereich `min..max` handelt:

```
bool istSuchbaum(struct knoten *b,
                 int min, int max) {
  if (b) {
//   std::cout << "Inhalt: " << b->x
//             << " min: " << min
//             << " max: " << max << '\n';
     if (b->x < min || b->x > max)
        return false;
     else
       return istSuchbaum(b->l, min, b->x)
           && istSuchbaum(b->r, b->x, max);
  } else return true; // leerer Baum
}
```

In Suchbaum einfügen

- Wenn wir ein neues Element in unsern Suchbaum einfügen wollen, müssen wir die richtige Stelle finden:

```
struct knoten *einfuegen(struct knoten *b,
                         int xneu) {
  if (b) {
    if (xneu < b->x)
      b->l = einfuegen(b->l, xneu);
    else
      b->r = einfuegen(b->r, xneu);
    return b;
   } else {
     return neu(xneu, NULL, NULL);
   }
}
```

- Ergebnis ist der vergrößerte Suchbaum

Suchbaum aufbauen

- Die Funktion kann man z.B. so nutzen:

```
struct knoten *b = NULL;
b = einfuegen(b, 3);
b = einfuegen(b, 1);
b = einfuegen(b, 2);
b = einfuegen(b, 4);
b = einfuegen(b, 8);
drucke(b, 1);
```

- Beim Aufbau des Suchbaums entstehen je nach Reihenfolge des Einfügens ganz unterschiedliche Bäume

In Suchbaum einfügen (2)

- Vielleicht habe wir lieber eine Variante, bei der der Baum ein Variablenparameter ist:

```
void einfuegen2(struct knoten **bp,
     int xneu) {
  struct knoten *b;
  b = *bp;
  if (b) {
    if (xneu < b->x)
      einfuegen2(&(b->l), xneu);
    else
      einfuegen2(&(b->r), xneu);
   } else {
      *bp = neu(xneu, NULL, NULL);
   }
}
```

Suchbaum aufbauen (2)

- Die neue Funktion kann man z.B. so nutzen:

```
struct knoten *b = NULL;
einfuegen2(&b, 3);
einfuegen2(&b, 1);
einfuegen2(&b, 2);
einfuegen2(&b, 4);
einfuegen2(&b, 8);
drucke(b, 1);
```

- Für das Verständnis dieser Funktion ist wesentlich, dass wir hier eine Zeigervariable als Variablenparameter haben – das erklärt die Deklaration `struct knoten **bp`

Aus dem Suchbaum löschen (1)

- Das Löschen eines Knotens im Suchbaums ist schon schwieriger – man muss darauf achten, dass die Suchbaumeigenschaft erhalten bleibt, auch wenn wir einen Knoten „mitten aus dem Baum“ entfernen
- Das mit allen Sonderfällen richtig ausprogrammiert gibt schon ein verhältnismäßig unübersichtliches Programm; deshalb sei hier nur das Prinzip erklärt
- Unproblematisch sind die Fälle, in denen wenigstens einer der Söhne leer ist – in diesem Fall kann der andere Sohn den (gelöschten) Vater ersetzen
- Wenn keiner der Söhne leer ist, sucht man (vom zu löschenden Knoten ausgehend) den mit der nächstgrößeren Beschriftung (ein Schritt nach rechts, dann so viele Schritte nach links wie möglich)

Aus dem Suchbaum löschen (2)

- Dann löscht man den so gefundenen Knoten und deponiert seine Beschriftung in dem Knoten, den man ursprünglich löschen sollte
- (Man darf natürlich auch symmetrisch dazu die nächstkleinere Beschriftung suchen)
- Als Parameter bekommt die Funktion am besten einen Baum als Variablenparameter („Lösche die Wurzel dieses Baums“), also wieder `struct knoten **bp`
- Wenn man es ausprogrammiert: es ist ohne weiteres erlaubt, einen Zeiger auf eine Strukturkomponente zu bilden (das sind Lvalues!), z.B. `&((*bp)->r)` – wie schon beim Einfügen mit Variablenparameter

Suchbäume: Effizienz

- Die typischen Operationen auf Suchbäumen (Einfügen, Suchen, Löschen) brauchen im schlimmste Fall größenordnungsmäßig so viele Operationen wie die Tiefe des Baums (Maximaler Abstand eines Knotens von der Wurzel plus 1)

- Ein Baum der Tiefe t hat minimal t, maximal 2^t-1 Knoten, ein Baum mit n Knoten also minimal Tiefe $\log_2(n+1)$ und maximal Tiefe n

- Für die Effizienz unseres Verfahrens ist es daher entscheidend, dass unsere Bäume schön „üppig“ bleiben (kleine Tiefe bei vielen Knoten) – es gibt Verfahren, um das durch passendes Umordnen des Baums sicherzustellen (z.B. AVL-Bäume)

Bäume

- Eine zu Binärbäumen verwandte Datenstruktur sind Bäume. Die haben eine verschränkt-rekursive Definition:
  - Ein *Wald* ist eine Menge von Bäumen
  - Ein *Baum* besteht aus einer Wurzel (mit Beschriftung) und einem Wald von Unterbäumen, den Söhnen
- Sieht zuerst nach einer Verallgemeinerung der Binär-bäume aus (beliebig viele Söhne)
- Bei Binärbäumen können wir aber linken und rechten Sohn unterscheiden (insbesondere, wenn genau einer der beiden leer ist),was in einer Menge nicht geht
- Noch was: es gibt einen leeren Binärbaum, aber keinen leeren Baum

Datenstruktur für Bäume

- Problem bei der Implementierung: Mengen sind heikler zu implementieren als Listen, daher unterstellen wir unsern Wäldern ab jetzt eine Anordnung ihrer Bäume („Sohn Nummer 1“ bis „Sohn Nummer *n*“)
- Ein Strukturtyp für Bäume könnte dann enthalten
  - die Beschriftung
  - die Söhne, d.h., einen Zeiger auf das erste Element der Sohnliste
  - und noch einen Zeiger auf den nächsten Bruder (unser Baum ist ja Element einer verketteten Liste)

Bäume und Binärbäume

- Als C-Datenstruktur aufgeschrieben sehen wir, dass die gleiche Datenstruktur rauskommt wie für unsere Binärbäume!
- Wir können also unsere Binärbäume auch verwenden, um Bäume zu speichern
- Die Verwandtschaftsbeziehungen in dem Binärbaum sind dabei aber völlig andere als die im dargestellten Baum
- Mit Ausnahme des leeren Binärbaums gehört übrigens zu jedem Binärbaum auch ein auf diese Weise dargestellter Baum