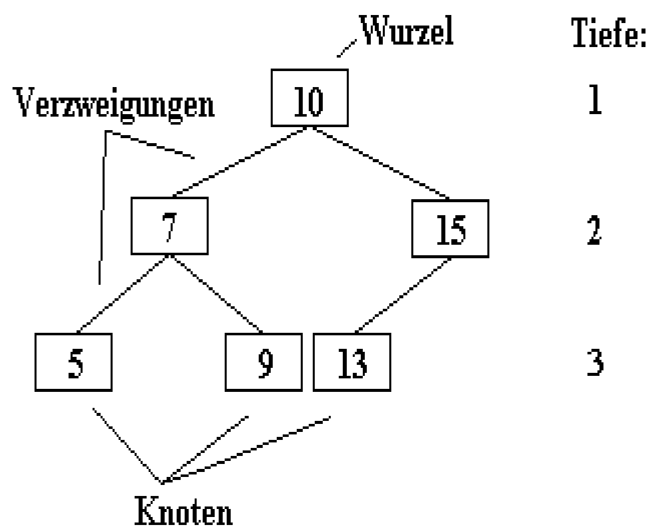
Binäre Bäume

1. Allgemeines

Binäre Bäume werden grundsätzlich verwendet, um Zahlen der Größe nach, oder Wörter dem Alphabet nach zu sortieren. Dem einfacheren Verständnis zu Liebe werde ich mich hier besonders auf die Zahlen beziehen, doch mit Buchstaben besteht in der Funktionsweise kein Unterschied.
Zur Definition einiger Begriffe:
Knoten bzw. Knotenpunkte sind die Stellen, wo die Zahlen eingetragen werden.
Wurzel wird der aller oberste Knoten genannt.
Verzweigungen sind die Verweise eines Knoten auf seinen linken oder rechten Nachfolgers.
Als Tiefe wird bei den Binären Bäumen die Anzahl der Knotenpunkte bis zum tiefsten Element genannt.
Nimmt man n als Tiefe an, so kann ein binärer Baum maximal $2^n - 1$ Knoten haben.



Ein Beispiel eines binären Baumes

2. Funktionsweise

2.1 Eintragen

Das Prinzip basiert darauf, daß eine Zahl immer auf zwei weitere verweist, eine größere rechts und eine kleinere links. Die erste Zahl, die eingelesen wurde, dient als Wurzel. Von ihr gehen dann die weiteren Verzweigungen aus. Bei der nächsten Zahl vergleicht man diese mit der Wurzel.

Ist die neue Zahl kleiner als die Wurzel, folgt man dem linken Zweig, ist sie größer, dem Rechten. Steht dort wieder eine Zahl, wird wieder verglichen, bis man an einem freien Platz angekommen ist. Hat man einen freien Platz erreicht, kann die Zahl eingetragen werden.

2.2 Ausgeben

Möchte man nun die Zahlen ausgeben, fängt man wieder bei der Wurzel an. Sollen die Zahlen aufwärts sortiert ausgegeben werden, geht man den linken Verzweigungen nach, solange dies möglich ist. Ist man am äußersten linken Punkt angekommen, ist dies nun die niedrigste Zahl. Danach geht man eine Verzweigung zurück und nimmt diese Zahl, dann wird deren rechter Zweig abgearbeitet, wobei dort wieder zuerst nach links gegangen werden muß. Nachdem auch der rechte Zweig abgeschlossen wurde, geht man eine weitere Verzweigung zurück und macht das selbe noch einmal.
Sollen die Zahlen abwärts sortiert ausgegeben werden, muß man nur die Instruktionen im oberen Absatz befolgen, wobei man aber links und rechts vertauscht. Das heißt man geht zuerst zum äußersten rechten Knoten, usw.
Sucht man eine bestimmte Zahl braucht man nur an jedem Knotenpunkt vergleichen, ob diese Zahl größer oder kleiner ist, und geht dem entsprechend der linken (Zahl ist kleiner als Knoten) oder der rechten (Zahl ist größer als Knoten) Verzweigung nach, bis die Zahl gefunden wurde, oder die Verzweigung auf NICHTS (bzw. NULL) verweist.

3. Programmierung

Binäre Bäume werden mit Hilfe von Pointern programmiert. Jeder Knotenpunkt beinhaltet die eigene Adresse, die darin gespeicherte Zahl, einen Zeiger für den linken Zweig und einen weiteren für den rechten Zweig. Da das Umgehen mit Pointern in C noch am einfachsten ist und diese Programmiersprache noch halbwegs aktuell ist, sind die unten angeführten Beispiele in dieser Sprache geschrieben.

3.1 Einfügen

Die Prozedur für das Einfügen ist der, in Kapitel 3.3 über das Löschen beschriebenen teilweise ähnlich. In beiden Prozeduren lasse ich eine Hilfsvariable nachlaufen.

```
void eingeben (struct knoten *akt, int x);
{
  struct knoten *hilfe;
  if (akt != NULL)
          {
           if (akt => zahl != x)
                   {
                    if (akt => zahl > x)
```

```
                {
                  hilf = akt;
                  eingeben (akt => links, x);
                }
                else
                if (akt => zahl < x)
                        {
                          hilf = akt;
                          eingeben (akt => rechts, x);
                        }
        }
    }
    else
    {
      akt = malloc (struct knoten);
      akt => rechts = NULL;
      akt => links = NULL;
      akt => zahl = x;
      if (hilf => zahl < x)
            hilf => rechts = akt;
            else
            hilf => links = akt;
    }
}
```

3.2 Ausgeben

Man kann sich die Zahlen natürlich auf mehrere Arten ausgeben lassen. Doch ich möchte nur zwei, die wahrscheinlich am häufigsten verwendet werden, zeigen.
Möchte man die Zahlen geordnet ausgeben, kann man folgende Prozedur benutzen, wobei akt bei Aufruf der Prozedur auf die Wurzel zeigen sollte:

```
void ausgabe (struct knoten *akt);
{
  if (akt != NULL)
    {
      ausgabe (akt => links);
      printf ("%d \n", akt => zahl);
      ausgabe (akt => rechts);
    }
}
```

Hat man sich, wie in Kapitel 3.4 beschrieben, die Tiefe eines binären Baumes bereits ausgerechnet, kann man die folgende Funktion verwenden, um den Baum in strukturierter Form darzustellen:

```
void ausgabe (struct knoten *akt, int tiefe);
{
  int i;
  if (akt != NULL)
          {
            ausgabe (akt => rechts, ++ tiefe);
            for (i = 1; i < tiefe; i++) printf ("   ");
            printf ("%d \n", akt => zahl);
            ausgabe (akt => links, tiefe);
          }
}
```

Die ausgegebenen Daten sehen nach Aufruf der zweiten Prozedur aus, wie ein gekippter binärer Baum (d.h. wie um 90° gedreht).

3.3 Löschen

Löschen ist der schwierigste Bereich bei den binären Bäumen. Aus diesem Grund muß eine Hilfsvariable verwendet werden, welche dem anderen Pointer immer um einen Knoten hinten nach ist.

```
void löschen (struct knoten *akt, int x);
{
  struct knoten *hilfe;
  if (akt != NULL)
          {
            if (akt => zahl != x)
                    {
                      if (akt => zahl > x)
                              {
                                hilf = akt;
                                löschen (akt => links, x);
                              }
                              else
                              if (akt => zahl < x)
                                      {
                                        hilf = akt;
                                        löschen (akt => rechts, x);
                                      }
                    }
```

```
                else
                {
                  if (hilf => rechts == akt)







                          {
                            if (akt => rechts != NULL)
                                  {
                                    hilf => rechts = akt => rechts;
                                    hilf = hilf => rechts;
                                    for (hilf = hilf; hilf => links != NULL; hilf = hilf => links);
                                    hilf => links = akt => links;
                                  }
                                  else hilf => rechts = akt => links;
                            free (akt);
                          }
                          else if (hilf =>links == akt)
                                  {
                                    if ( akt=>links != NULL)
                                  {
                                    hilf => links = akt => links;
                                    hilf = hilf => links;
                                    for (hilf = hilf; hilf => rechts != NULL; hilf = hilf => rechts);
                                    hilf => rechts = akt => rechts;
                                  }
                                  else hilf => links = akt => rechts;
                                  free (akt);
                                  }
                          else free (akt);
                }
        }
}
```

3.4 Tiefe berechnen

Die folgende Funktion liefert als Retourwert die Tiefe eines binären Baumes. An die Funktion muß beim Aufruf bloß die Wurzel als Parameter übergeben werden.

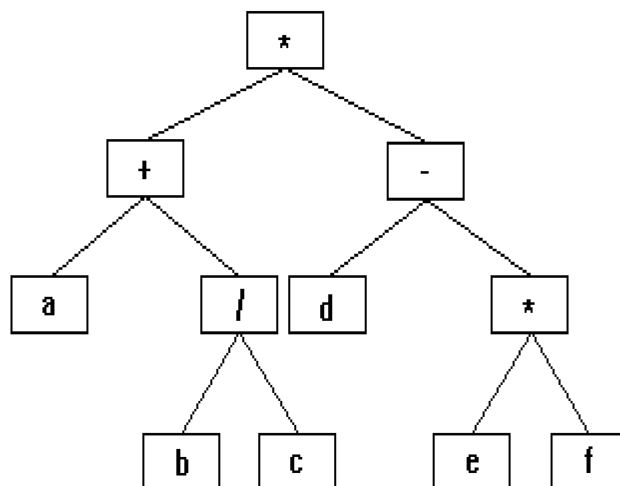
```
int tiefe (ptr *akt);
{
  int tl, tr;
  if (akt = = NULL) return (0)
        else
```

```
        {
         tl=tiefe (akt => links);
         tr=tiefe (akt => rechts);
         if (tl > tr ) return (tl + 1)
                else return (tr + 1);
        }
}
```

4. Sonstige Verwendungsmöglichkeiten für binäre Bäume

Ein binärer Baum kann auch dazu verwendet werden, um dyadische Operatoren darzustellen. Dies könnte im folgenden so aussehen:

(a + b / c) * (d - e * f)



Es gibt 3 Bearbeitungsarten:

1. PRÄFIX: root before node
   z.B.: * + a / b c - d * e f

2. INFIX: subtree root subtree
   z.B.: a + b / c * d - e * f

3.POSTFIX: subtrees before root
   z.B.: a b c / + d e f * - *