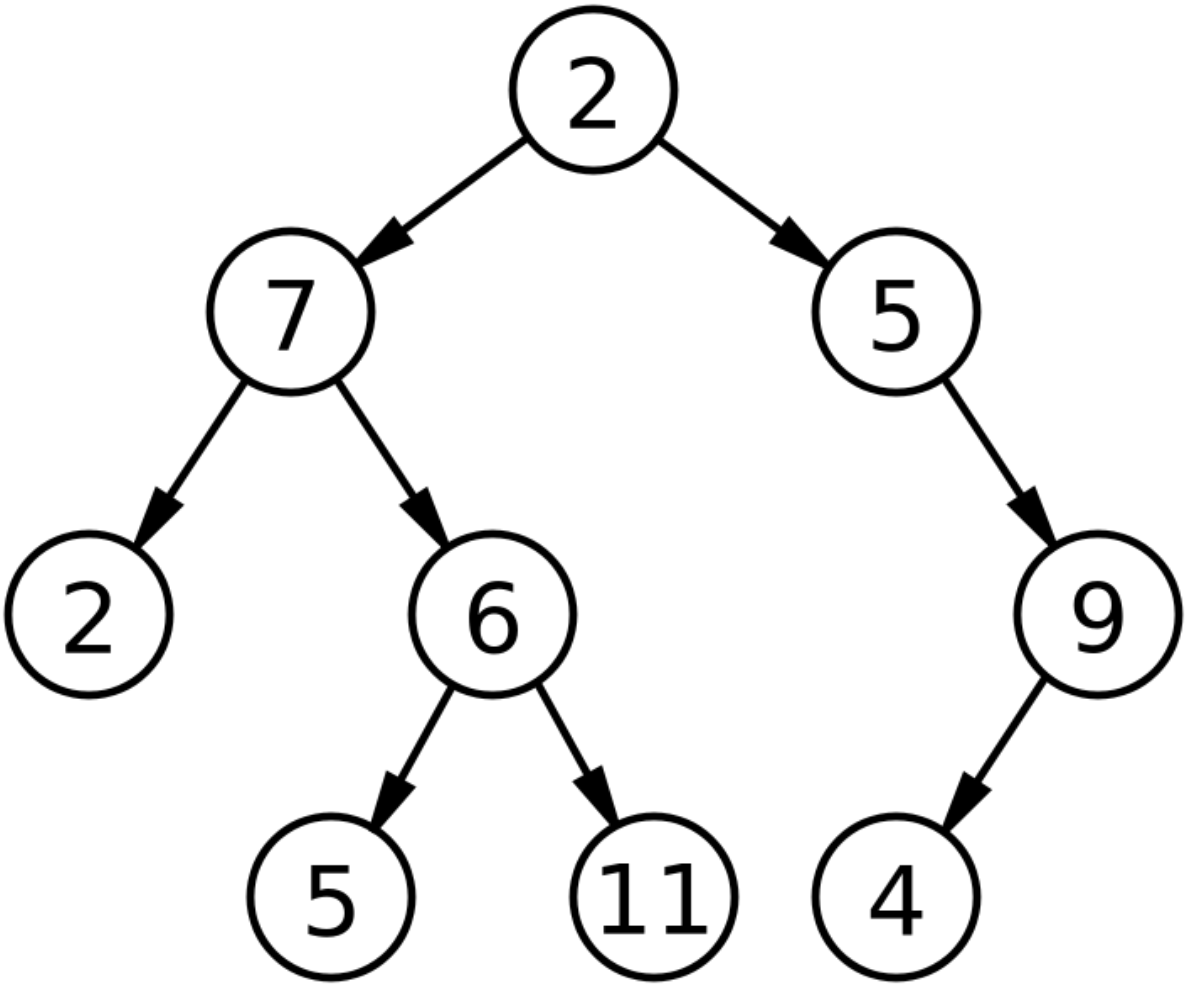
Seminararbeit: Exploring Data Structures in C

-Binärbäume / Binary Trees-



Gliederung

1. Vorwort

Mit diesem schriftlichen Teil möchte ich das Modul: Seminar: Exploring Data Structures in C vervollständigen und meine mündliche Ausarbeitung komplementieren. Das zentrale Thema in dieser schriftlichen Arbeit stellt dabei die Datenstruktur: Binärbaum / Binary Tree dar.

Neben dem grundlegenden Aufbau eines solchen Baumes, möchte ich noch tiefer in die Materie des Themas eintauchen und über die verschiedenen Anwendungsgebiete schreiben. Zusätzlich sollen auch die Punkte Suchkomplexität, aber auch Traversierung eines Binärbaums ihren Platz in dieser Seminararbeit finden. Schlussendlich möchte ich die Arbeit mit einem Programmierbeispiel abrunden.

2. Aufbau Binärbaum

Was ist ein Binärbaum?

Bevor ich den Aufbau eines Binärbaumes erörtere oder gar auf die Frage – *Was ist ein Binärbaum?* – eingehe, möchte ich noch einen kurzen Bogen spannen und zunächst erklären was überhaupt eine Baumdatenstruktur in der Informatik darstellt:

Bei einem Baum handelt es sich um ein Objekt, was in der Informatik zum Speichern und Strukturieren von Daten verwendet wird, dabei werden die Daten in einer speziellen Art und Weise gespeichert, die das spätere Abrufen wesentlich einfacher und effizienter gestalten soll. Die Anordnung bzw. Formation der gespeicherten Inhalte, erinnert dabei in ihrer abstrakten Form sehr stark an einen umgedrehten Baum oder an die Wurzel eines Baumes.

Wie auch in der Natur unterscheiden wir auch in der Informatik zwischen verschiedenen Bäumen, dabei unterteilen wir aber nicht in Eiche, Birke oder Fichte, sondern in beispielsweise AVL-Bäume oder Fibonacci-Trees. Bei diesen aufgezählten Varianten handelt es sich um Spezialformen von Binärsuchbäumen – welche wiederrum eine Spezialform der Binärbäume darstellen [*mehr zu den Spezialformen von Binärbäumen im 5. Abschnitt*]. Was uns zurück zur eigentlichen Frage – *Was ist ein Binärbaum?* – bringt.

Ein Binärbaum stellt die wohl wichtigste oder zumindest grundlegendste Form eines Baumes/ einer Baumdatenstruktur dar. Hierbei handelt es sich um eine Anordnung von Daten in Form von Knoten, die wie es der Name bereits assoziiert, binär angeordnet sind. Im Konkreten bedeutet dies, dass alle Knoten über **maximal** zwei Unterknoten verfügen dürfen. Häufig bezeichnet man die übergeordneten Knoten auch als Elternknoten und schließlich die daraus resultierenden Knoten als Kindsknoten oder Kinderknoten.

Wie ist ein Binärbaum aufgebaut?

Die simple Form eines Binärbaums lässt den Trugschluss aufkommen, dass ein solcher Baum keine, bis wenige wichtige Elemente besitzen muss, diese Annahme möchte ich aber im folgenden Abschnitt widerlegen und die vielfältigen Elemente, bzw. die einzelnen Abschnitte eines Binary Trees genauer unter die Lupe nehmen:

Da ein Binärbaum immer vom obersten Element bis zu dem untersten Element, also einfach formuliert, von oben nach unten, gelesen wird, möchte ich auch ganz „oben“ anfangen den Binärbaum zu skalpieren. Bei diesem ersten und „obersten“ Knoten handelt es sich nämlich um eine erste Besonderheit. Hier spricht man vom sogenannten Wurzelknoten oder auch

kurz der Wurzel. Diese Wurzel stellt den zentralen Ursprung des Baums dar, alle anderen folgenden Knoten sind lediglich Kinder der Wurzel bzw. Kinder der Kinder, der Wurzel. Neben der zentralen Aufgabe als Ursprungspunkt, besitzt die Wurzel aber eine weitere wichtige Eigenschaft, nämlich fungiert die Wurzel auch als Ausgangspunkt für die Höhenberechnung des Binärbaums. Als erster Knoten erhält die Wurzel deswegen die Höhe 0 oder die Höhe 1. [*Beide Varianten sind bei der Höhenbestimmung richtig, allerdings wird häufiger auf die Höhe 1 zurückgegriffen, da man einem leeren Binärbaum dann die Höhe 0 geben kann, andernfalls müsste ein leerer Binärbaum sonst die Höhe -1 zugeschrieben bekommen, was mathematisch einen Widerspruch auslösen würde.*]

Neben der Wurzel verfügt ein Binärbaum zusätzlich über Kanten, diese sind die Verbindungsbrücken zwischen den einzelnen Elementen bzw. Knoten des Baumes. Jeder Knoten [*außer der Wurzelknoten*] besitzt eine eingehende Kante und maximal zwei weitere ausgehende Kanten, man kann also behaupten, dass ein Knoten über einen Eingangsgrad von eins und einem Ausgangsgrad von **mindestens** 0, bis **maximal** 2, verfügt.

Eine solche adjazentische Betrachtung der Knoten ist wichtig für die Definition des nächsten Elements: einem Blatt bzw. Halbblatt. Wir sprechen über ein Blatt, immer dann, wenn in einen Knoten eine Kante hineinfließt, allerdings keine weitere Kante hinausströmt [*Eingangsgrad von 1; Ausgangsgrad von 0*], sinnbildlich handelt es sich um ein Ende des Binärbaums. Währenddessen stellt ein Halbblatt einen Abschnitt der Datenstruktur, der über einen Eingangsgrad von 1 **und** einen Ausgangsgrad von 1 verfügt, dar. Bei einem Halbblatt kann man also niemals von einem Ende des Binärbaums sprechen.

Das letzte Strukturelement beschreibt die linke bzw. rechte Seite eines Baumes. Passend dazu trägt dieser Abschnitt etwa den Namen linker oder rechter Teilbaum. In der folgenden Abbildung sind alle Elemente eines Binärbaumes nochmal zusammengefasst:

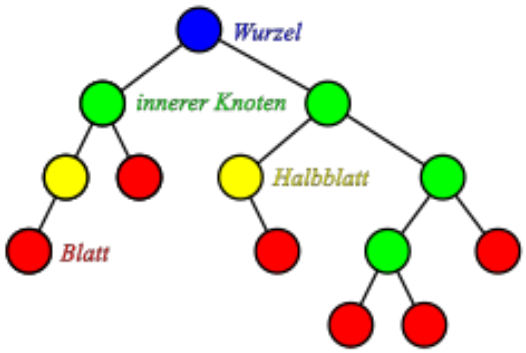


Abbildung 1: Die verschiedenen Elemente eines Binärbaums

Welche speziellen Formen von Binärbäumen gibt es?

Aufgrund der speziellen Struktur eines Binärbaums, also sprich durch die variierende Anzahl von null bis maximal zwei Kindsknoten, eines jeden Elternknotens, ergeben sich einzigartige Formationen und Formen von Binärbäumen, die ich im folgenden Abschnitt einzeln benennen und durchleuchten möchte:

Bei der ersten und auch wichtigsten Regelform eines Binärbaums, handelt es sich um einen **geordneten** Binärbaum. Diese birgt die Besonderheit, dass alle Bezugsknoten ein linkes Kind, mit einem geringeren gespeicherten Wert, enthalten und möglicherweise auch über ein rechtes Kind, mit einem höheren gespeicherten Wert, verfügen. In der unten abgebildeten Grafik habe ich versucht, das „geordnete“-Schema mit Relationszeichen zu verdeutlichen. Ein geordneter Binärbaum, ist dahingehend die wichtigste Spezialform eines Binärbaums, da sie elementar wichtig für die Definition eines Binärsuchbaums ist [*mehr dazu in Abschnitt 4: Anwendungsmöglichkeiten*].

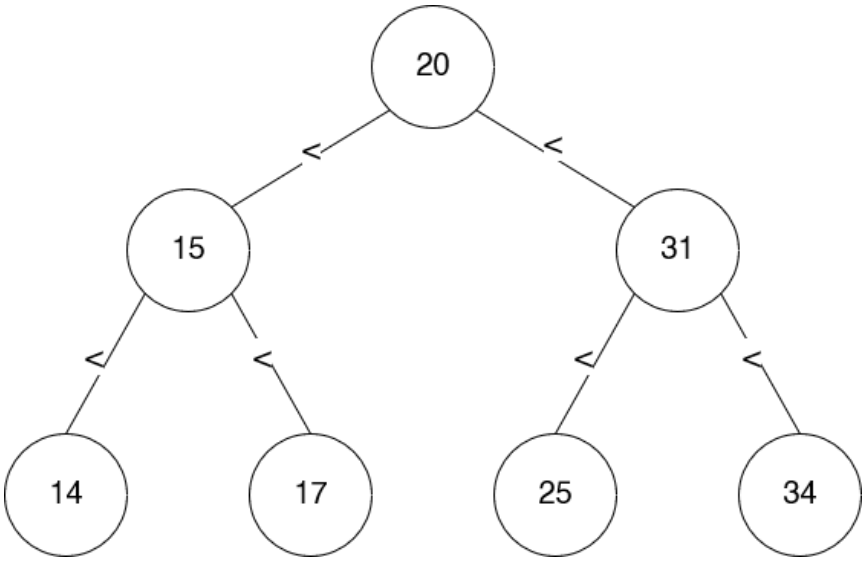


Abbildung 2: Geordneter Binärbaum

Neben den geordneten-Binärbaum, gibt es drei weitere Spezialformen des Binary Trees, diese haben zwar keinen direkten Bezug zu weiteren Anwendungsmöglichkeiten wie beispielsweise dem Binärsuchbaum, erfüllen aber dennoch eine wichtige Rolle in Bezug auf die Berechnung der zeitlichen Suchkomplexität von Binärsuchbäumen. Darunter zählen **volle** Binärbäume [*häufig auch als **saturiert** oder **strikt** bezeichnet*], aber auch **vollständige** und **entartete** Bäume.

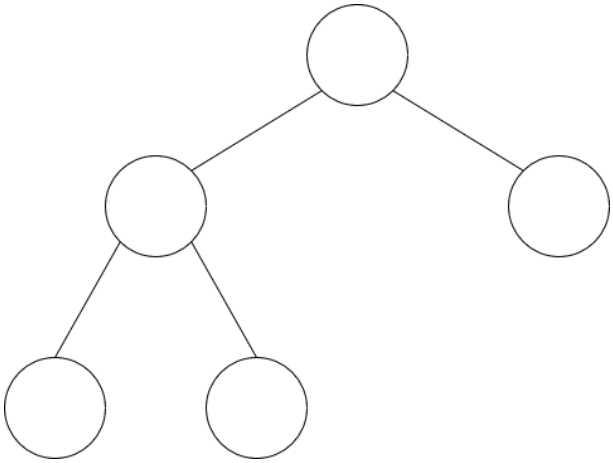


Abbildung 3: Voller Binärbaum: Alle Knoten sind hier entweder Blatt oder Eltern von zwei Kindern

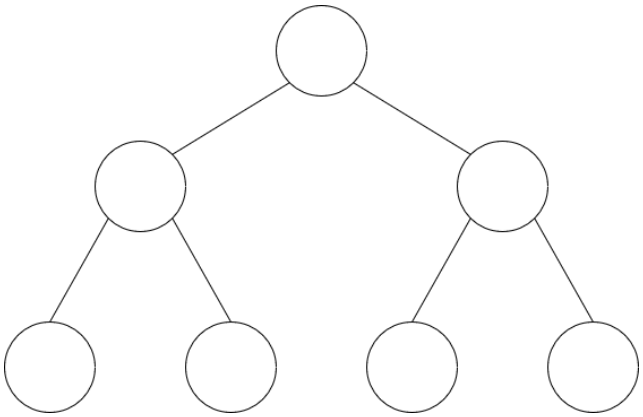


Abbildung 4: Vollständiger Binärbaum: Alle Blätter im Baum enden auf derselben Höhe

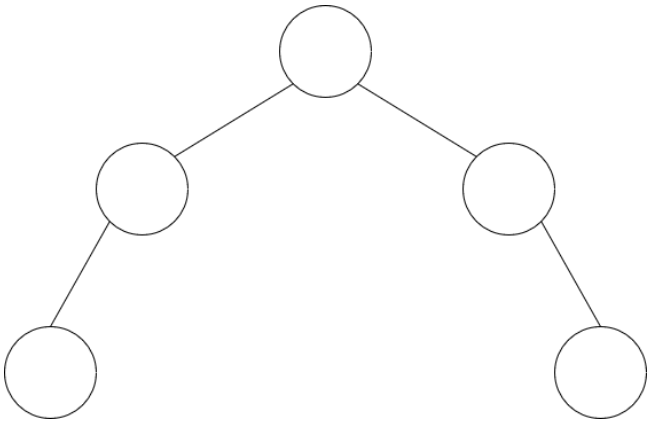


Abbildung 5: Entarteter Binärbaum: Alle inneren Knoten sind Halbbläter im Baum

3. Traversierung

Die Traversierung eines Binärbaums beschreibt das strategische Durchlaufen dieser Datenstruktur. Hierbei werden die einzelnen Knotenelemente in unterschiedlicher Reihenfolge durchgegangen und anschließend in einer linearen Form [z.B. *Liste*] gespeichert. Bei der Traversierung von Binärbäumen unterscheiden wir zwischen fünf verschiedenen Methoden den Baum zu durchsuchen:

Pre-Order-Verfahren

Bei diesem [*und auch den drei folgenden*] Traversierungsverfahren handelt es sich um ein Suchalgorithmus für deren Durchführung man häufig auf die Strukturelemente: Wurzel – linker Teilbaum – rechter Teilbaum oder auch kurz (W) – (L) – (R) zurückgreift. Diese Strukturelemente des Binärbaums helfen bei der Durchführung des jeweiligen Order-Verfahrens. Bei dem Pre-Order-Verfahren, was häufig auch als Tiefensuche bezeichnet wird, extrahiert man die einzelnen Knotenelemente nach dem Schema (W) – (L) – (R). Man beginnt also bei der Wurzel des Baumes und durchläuft anschließend alle Elemente des Pfades von links-oben bis rechts-unten. Sind alle Knoten dieses Pfades abgearbeitet, kann man schließlich beginnen den [*rechts*] anliegenden Pfad erneut von links-oben, nach rechts-unten zu durchleuchten. Dieses Vorgehen wird wiederholt, bis alle Pfade abgearbeitet sind.

Post-Order-Verfahren

Bei dem Post-Order-Verfahren oder auch Nebenreihenfolgenverfahren wird der Binärbaum, ähnlich wie bei dem Pre-Order-Verfahren, anhand der drei Strukturelemente analysiert. Hier lautet die Reihenfolge (L) – (R) – (W). Mit anderen Worten bedeutet dies, dass als erstes der linke und dann der rechte Teilbaum durchquert werden, anschließend wird noch die Wurzel betrachtet und aufgeschrieben.

In-Order-Verfahren

Dieses symmetrische Verfahren durchläuft den Binärbaum in der Reihenfolge: (L) – (W) – (R). Zunächst blickt man also auf den linken Teilbaum, anschließend auf die Wurzel und zu guter Letzt durchläuft man noch den rechten Teilbaum.

Reverse-In-Order-Verfahren

Auch bei diesem Algorithmus greifen wir auf das bewährte 3er-Schema mit Wurzel, linken und rechten Teilbaum zurück. Diese, antisymmetrische Reihenfolge, kann also über (R)-(W)-(L) auseinandergenommen werden. Man könnte also behaupten, dass wir das In-Order-Verfahren vice versa anwenden.

Level-Order-Verfahren

Der Breitensuchalgorithmus durchbricht erstmalig das dreiteilige (W) – (L) – (R) – Schema, was für die Analyse und Durchführung der anderen vier Algorithmen notwendig war. Bei diesem Verfahren wird der Binärbaum ähnlich wie ein Buch „gelesen“. Dabei durchläuft man die einzelnen Knoten von links-oben, nach rechts-unten. Im Gegensatz zum Pre-Order-Verfahren spielen die einzelnen Pfade allerdings keine Rolle, ein vertikales Springen, zu Knoten auf derselben Höhe [*die nicht auf einem Pfad liegen*], ist also möglich.

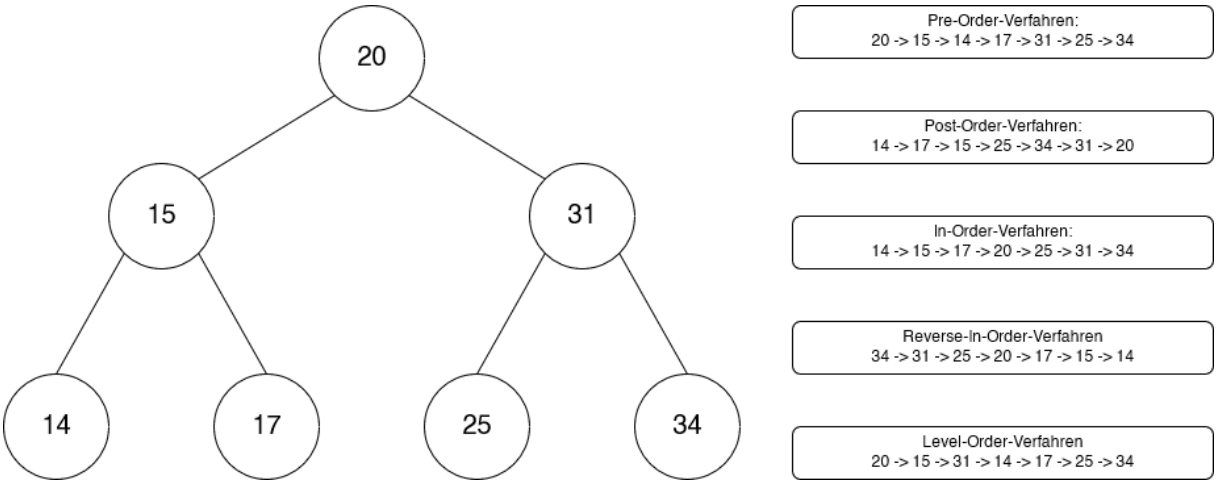


Abbildung 6: Traversierung eines Binärbaumes

Weswegen werden die Order-Verfahren verwendet?

Die verschiedenen Traversierungsmethoden bieten den übergeordneten Vorteil Binary Trees möglichst effizient zu durchsuchen und anschließend die einzelnen Elemente in andere Datenstrukturen einzusortieren [*z.B. in eine geordnete Liste*]. Im Umkehrschluss bieten die Order-Verfahren auch die Möglichkeit Elemente [*aus bspw. einer geordneten Liste*] rückwirkend in einen leeren Binärbaum zurückzugeben. So verwendet man u.a. das In-Order-Verfahren, um einen Binärbaum in aufsteigender Sortierung auszugeben [*sofern es sich vorher um einen binären Suchbaum gehandelt hat*].

4. Anwendungsbeispiele

Hash-Bäume, Huffman-Code-Bäume, Syntaxbäume sind nur einige Beispiele der schier unbegrenzten Liste an möglichen Anwendungsbeispielen von Binärbäumen. Diese lange Liste an verschiedenen Anwendungsbeispielen ergibt sich aus der Tatsache, dass es sich bei den Binärbäumen weniger um eine einzelne Datenstruktur handelt, sondern gar um eine Familie von Datenstrukturen, welche vielfältige und unterschiedliche Leistungsmerkmale aufweisen. Da es den Rahmen einer einzelnen Seminararbeit sprengen würde über all diese unterschiedlichen Baumstrukturen, mit ihren vielfältigen Merkmalen, zu schreiben, werde ich mich in den nachfolgenden Zeilen auf zwei [*triviale, aber auch die vermutlich geläufigsten*] Anwendungsbeispiele konzentrieren: Binärsuchbäume und die Sortierung des Morsecodes nach Binärbaumkriterien.

Organisation des Morsecodes

Das erste Beispiel, auf das ich eingehen möchte, bezieht sich auf den Morsecode oder präzise formuliert, auf den Zeichensatz des Morsecodes. Dieser ist nämlich nach den grundlegenden Prinzipien des Binärbaumes aufgebaut. Der Morsecode verfügt über einen Startpunkt, dieser ist vergleichbar mit der Wurzel des Binärbaumes. Zusätzlich sind 39 weitere Buchstaben, Zahlen und Sonderzeichen im Baum enthalten, die durch manövrieren zum jeweiligen linken oder rechten Kindsknoten dargestellt werden können. Eine Bewegung nach links lässt sich dabei mit einem Punkt oder einem „kurz“-Signal darstellen, während eine Bewegung nach rechts mit einem Strich oder „lang“-Signal modelliert werden kann. Durch Aneinanderreihung der verschiedenen Signale kann also ein vollständiger Buchstabe ausgegeben werden. Da der Morsecode hauptsächlich für die Übermittlung von kurzen Nachrichten verwendet wird, verzichtet man i.d.R. darauf ganze Wörter oder gar Sätze zu übermitteln, häufig wird deswegen auf eine Nachrichtenlänge von ein bis vier Buchstaben zurückgegriffen, die in einer speziellen Anordnung unterschiedliche Situationen beschreiben soll [*z.B. SOS deutet auf eine Notsituation hin*]. Der Binärbaum des Morsecodes ist so konzipiert, dass Buchstaben, die im Sprachgebrauch regelmäßiger auftreten als andere Buchstaben, weiter oben im Baum gelistet werden. Dadurch ist eine schnellere Modellierung der wichtigsten Zeichen möglich.

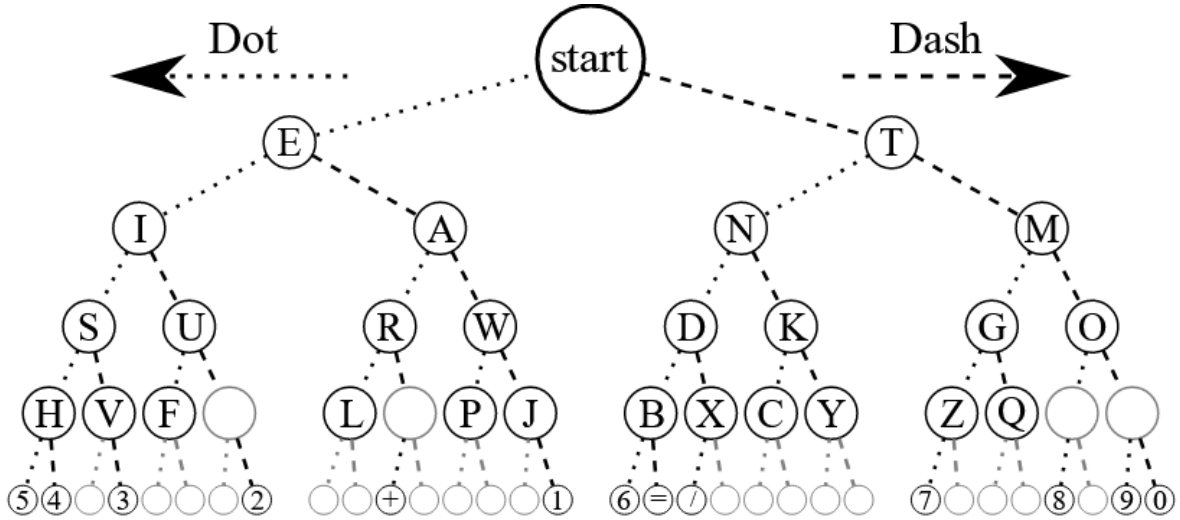


Abbildung 7: Organisation des Morsecodezeichensatzes nach Binärbaumkriterien

Binärsuchbaum

Mein zweites Anwendungsbeispiel dürfte zwar weniger geläufiger sein als der Morsecode, besitzt aber eine ähnlich wichtige Bedeutung, dabei handelt es sich um den Binärsuchbaum. Diese einzigartige Baumart stellt eine Spezialform eines Binärsuchbaumes dar. Alle Elemente innerhalb dieser Datenstruktur sind in einer besonderen Reihenfolge angeordnet, welche einem geordneten Binärbaum gleicht. Zur Wiederholung: alle Zahlen, die größer sind als die des Bezugsknoten, werden als rechter Kindsknoten gespeichert, während Elemente, die kleiner sind als die des Bezugsknoten, als linkes Kind eingeordnet werden. Diese schematische Anordnung durchdringt den gesamten Binärbaum.

Um den Erstellungs-/ Sortierungsprozess eines binären Suchbaums zu veranschaulichen, habe ich das folgende Beispiel vorbereitet. Hierfür habe ich die gegebenen Zahlen von links nach rechts entnommen und mit den gegebenen Bezugselement verglichen. Für die Wurzel bzw. das erste Bezugselement habe ich mich für die Zahl 50 entschieden, da diese dem Mittelwert aller gegebenen Zahlen entspricht. Die Wahl eines Wurzelelements, welches in der Nähe des Mittelwertes aller gegebenen Zahlen liegt, ist zu empfehlen [*Dies lässt sich mit einer verbesserten Sucheffizienz begründen, nähere Ausführungen dazu im fünften Abschnitt*].

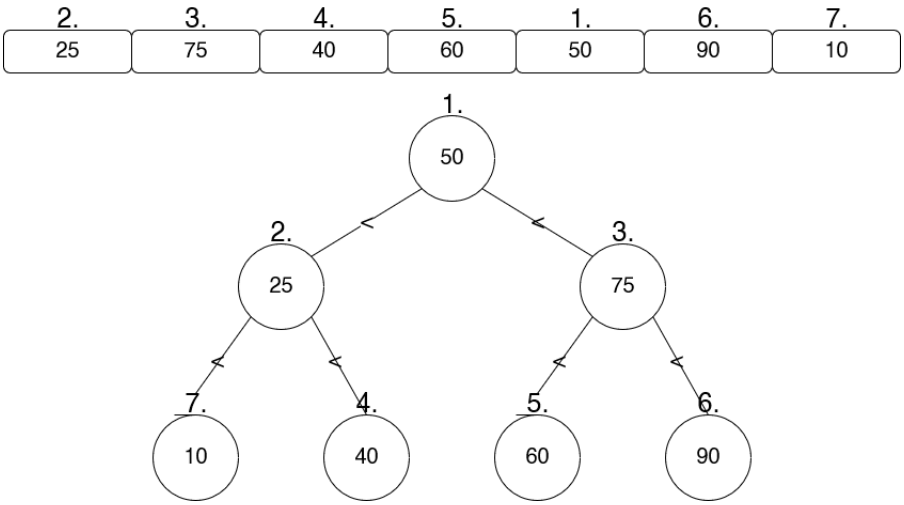


Abbildung 8: Erstellen eines Binärsuchbaumes

Warum wird ein Binärsuchbaum verwendet?

Ähnlich wie ein Binärbaum, wird auch ein Binärsuchbaum, hauptsächlich zum Speichern und Abrufen von Daten verwendet. Die einzigartige und geordnete Struktur eines Binärsuchbaumes bietet allerdings die Möglichkeit die gespeicherten Daten wesentlich schneller bzw. effizienter abzurufen als bspw. bei einem Binärbaum [*Ein passender Vergleich ist im fünften Abschnitt zu finden*].

5. Vergleich mit anderen Datenstrukturen

In diesen Abschnitt möchte ich Anhand eines ausgewählten Beispiels einen Vergleich zwischen verschiedenen Datenstrukturen ziehen. Für dieses Experiment habe ich die drei Datenstrukturen: Liste, Binärbaum und Binärsuchbaum vorbereitet. Ziel des Vergleiches ist es die Zahl 113 möglichst schnell, also mit so wenig Vergleichsoperationen wie möglich, zu finden. Alle Strukturen sind mit denselben Zahlen gefüttert.

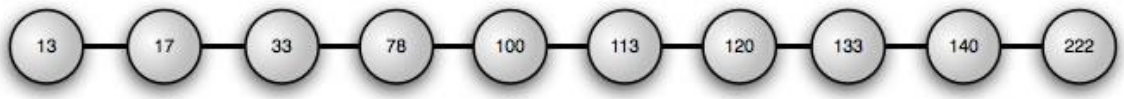


Abbildung 9: Liste mit zufälligen Zahlen

Bei dem ersten Beispiel durchlaufen wir eine Liste zufälliger Zahlen von links nach rechts, mit Hilfe der linearen Suche. Um das ausgemachte Ziel, die Zahl 113, zu erreichen sind also sechs Vergleichsoperationen notwendig.

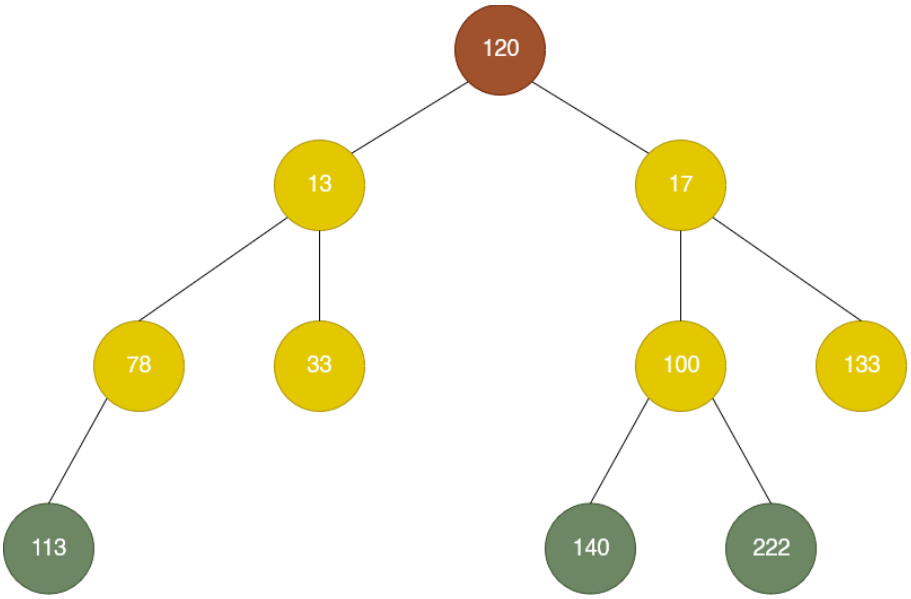


Abbildung 10: Binärbaum

Um dieselbe Zahl innerhalb dieses willkürlich zusammengesetzten Binärbaum zu finden, werden wiederum fünf Schritte benötigt, vorausgesetzt wir durchlaufen ihn mittels Pre-Order-Verfahren. Eine andere Traversierungsmethode wie bspw. das Reverse-In-Order-Verfahren nimmt sogar deutlich mehr Versuche in Anspruch.

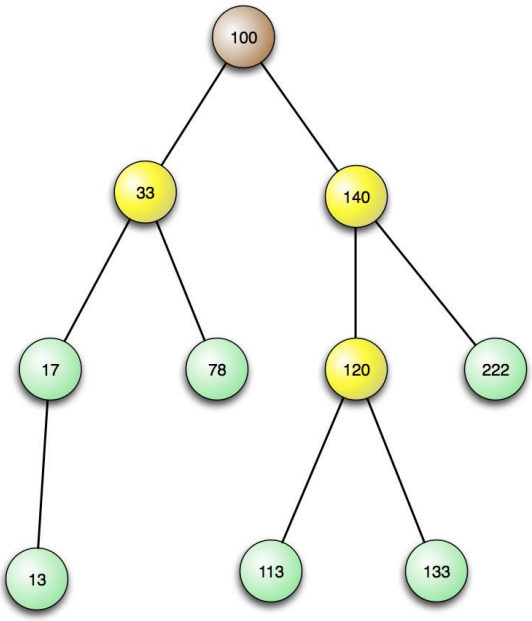


Abbildung 11: Binärsuchbaum

Nehmen wir jetzt dieselbe Baumstruktur wie zuvor und ordnen die Zahlen nach Binärsuchbaumkriterien, dann können wir die gewünschte Zahl nach lediglich vier Vergleichen ausfindig machen. Bei einem weiteren Experiment, [*es soll z.B. die Zahl 100 gefunden werden*], wird schnell klar, dass auch hier der Binärsuchbaum als Gewinner hervor geht. Zum Erreichen dieser Zahl wird lediglich ein Versuch benötigt.

Natürlich könnte man jetzt behaupten, dass ich mich bei den Beispielen auf Zahlen beschränkt habe, die den Binärsuchbaum als explizit schneller als die anderen beiden Datenstrukturen darstellen lassen. Möchte man nämlich in den oben aufgelisteten Datenstrukturen die Zahl 13 finden, so wäre sogar die Liste, die Datenstruktur, welche am wenigsten Vergleiche benötigt, um das Zielobjekt ausfindig zu machen. Dieser Einwand ist also durchaus berechtigt, diesen möchte ich aber mit dem Argument entgegnen, dass mit steigender Anzahl an Elementen innerhalb einer Datenstruktur, die Anzahl an Fällen, indem ein Binärbaum oder gar eine Liste das Ziel schneller ausfindig machen können als ein Binärsuchbaum, deutlich abnimmt. Man kann also behaupten, dass Binärsuchbäume deutlich effizienter beim Abrufen von Daten sind als die zum Vergleich herangezogenen Datenstrukturen.

Dennoch ergeben sich auch innerhalb von Binärsuchbäumen teils erhebliche Differenzen in der Sucheffizienz. Dies lässt sich mit unterschiedlichen Strukturen beim Auftreten von Bäumen erklären. Ein vollständig ausgeglichener Binärsuchbaum ist optimal geeignet für das binäre Suchverhalten und somit nicht ausgeglichenen Artgenossen einen großen Schritt in Sachen Sucheffizienz voraus. Dies wird auch im nächsten Beispiel

deutlich. Hierfür habe ich erneut zwei Binärsuchbäume vorbereitet, welche sich lediglich in ihrer Struktur, nicht aber bei den aufkommenden Zahlen unterscheiden.

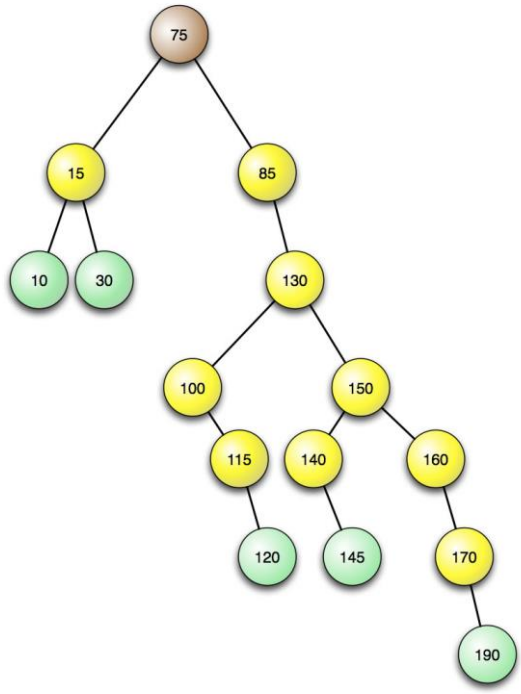


Abbildung 12: Unausgeglichener Binärsuchbaum

Obwohl dieser Binärbaum alle Kriterien eines Binärsuchbaums erfüllt und in geordneter Reihenfolge auftritt, wird schnell klar, dass sich dennoch eine starke Diskrepanz in der Sucheffizienz ergibt. Während die Zahl zehn nach bereits drei Operationen gefunden werden kann, benötigt man sieben Vergleiche, um den Knoten mit dem Inhalt 190 zu erreichen.

Rechnet man alle Optionen zusammen, also 1*1 + 2*2 + 3*3 + 2*4 + 3*5 + 3*6 + 1*7 = 62 und teilt die Summen aller möglichen Vergleiche durch die Anzahl der Elemente, dann erhält man ein Ergebnis von ~4,13 durchschnittlichen Vergleichen, um eine bestimmte Zahl zu finden.

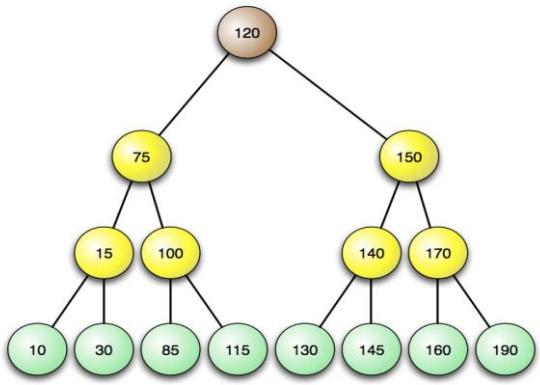


Abbildung 13: Vollständig ausgeglichener Binärbaum

Wandelt man den oben verwendeten Binärsuchbaum zu einen vollständig ausgeglichenen Binärsuchbaum um, kann man schnell erkennen, dass jetzt alle Zahlen im Baum mit maximal vier Vergleichsoperationen gefunden werden können. Bei der Berechnung der durchschnittlichen Vergleiche erhält man sogar ein Ergebnis von ~3,27 Vergleichen, die nötig sind, um eine Zahl im Baum zu finden [*1*1 + 2*2 + 4*3 + 8*4 = 49/15 = 3,27*]. Es ist also sinnvoll, seine Daten, innerhalb eines geordneten und vollständig ausgeglichenen Binärbaum, zu speichern.

6. Suchkomplexität

Nachdem ich bereits im letzten Abschnitt einige lose Berechnungen zur Suchkomplexität durchgeführt habe, möchte ich nun eine allgemeine Formel zur Sucheffizienz innerhalb von Binärbäumen ableiten. Da die Elemente in einem Binärbaum meist zufällig angeordnet werden und auch häufig unterschiedliche Strukturen auftreten, lässt sich keine allgemeine Formel für einen Binärbaum formulieren, stattdessen möchte ich mich auf eine Spezialform des Binärbaums konzentrieren: vollständig ausgeglichene Binärsuchbäume.

Bei der Suchkomplexität handelt es sich einfach ausgedrückt, um das zeitliche Suchverhalten, um ein gewünschtes Element innerhalb der Datenstruktur zu finden. Trivialerweise könnte man behaupten, die Suchkomplexität eines vollständig ausgeglichenen Binärsuchbaums, beschreibt die Anzahl wie häufig sich die Elemente (N) durch 2 teilen lassen, bis man die Zahl 1 erhält [*da ein vollständig ausgeglichener Binärsuchbaum immer aus einer ungeraden Anzahl an Elementen besteht, empfiehlt es sich zunächst N+1 zu rechnen, bevor man mit dem Dividieren beginnt*]. Mathematisch lässt sich dies wie folgt ausdrücken:

$1 = N/2^x$

Durch Umstellen und Logarithmieren, ergibt sich die Formel:

$2^x = N$

$\text{Log}_2 (2^x) = \text{Log}_2 (N)$

$x * \text{Log}_2 (2) = \text{Log}_2 (N)$

$\text{Log}_2 (N) = x$

Es werden also maximal $\text{Log}_2 (N)$ Vergleichsoperationen benötigt, um ein Element innerhalb des vollständig ausgeglichenen Binärsuchbaumes zu finden [*Erinnerung: es empfiehlt sich zunächst die Anzahl der Knoten +1 zu nehmen, damit sich bei der Rechnung kein Rest ergibt*].

Beispiel: vollständig ausgeglichener binärer Suchbaum mit 15 Knoten [*15 + 1 = 16 = N*]

$\text{Log}_2 (16) = 4$ ✓ [*In Abschnitt 4 wurde dies bereits graphisch bewiesen*]

Kekule-Nummer

Bei der Kekule-Nummer handelt es sich um ein Nummerierungsverfahren für die Identiätsermittlung bei Stammbäumen oder Ahnentafeln. Diese Nummerierungsmethode lässt sich, wenn auch zweckentfremdet, auf einen Binärsuchbaum übertragen. Dadurch können bestimmte Abschnitte im Baum nummeriert und berechnet werden.

Zunächst erhält die Wurzel die Zahl eins zugeschrieben, man bezeichnet den Wurzelknoten dann auch als Proband. Anschließend werden alle nachfolgenden Knoten mit der Zahl des Vorgängers+1, in Level-Order Verfahren, angeordnet. Ist der vollständig ausgeglichene Binärsuchbaum durchnummeriert, können nun die folgenden Berechnungen durchgeführt werden:

Die Höhe/ maximale Anzahl an Durchlaufversuchen: Log_2 (N+1) [*Im ursprünglichen Sinne, die Anzahl an Generationen in der Ahnentafel*]

Die Anzahl an Knoten im Baum: $2^h - 1$ [*Im ursprünglichen Sinne, die Anzahl an Verwandten in der Ahnentafel; h betitelt die Höhe*]

Bei diesen Berechnungen greifen wir also auf die bereits hergeleiteten Suchkomplexitätsformeln zurück. Deswegen wird die Keukele-Nummerierung auch häufig als übergeordnete Methode zur schnellen und effizienten Adressierung von Binärbaumknoten in einem Array, innerhalb der Informatik, verwendet.

7. Codebeispiel

Bei meinem Programmierbeispiel habe ich mich darauf konzentriert die erste Version des Programmes, welche ich während der mündlichen Ausarbeitung präsentiert hatte, weiterzuentwickeln. Während die alte Version lediglich einen vorgefertigten Binärbaum mit einer festen Anzahl an Elementen ausgeben konnte, besitzt die neue Version die Möglichkeit einen Binärbaum mit einer variierenden Anzahl an Elementen darzustellen. Dafür wird zunächst nach der gewünschten Knotenanzahl gefragt, anschließend werden die Knoten einzeln erstellt und bereits mögliche Kindsknoten präsentiert – auch hier handelt es sich um ein neues Feature. Bevor nun der vollständige Baum angezeigt werden kann, gibt die Konsole noch die erzeugte Anzahl an Knoten aus. Abschließend wird die finale Version des Binary Tree in der Konsole gelistet, um hier den Überblick in der Struktur zu behalten, sind Blätter mit ()-Klammern gekennzeichnet. Der Baum wird nach der Level-Order-Traversierungsmethode durchnummeriert.

```
PS C:\Users\Fritz\Desktop\TUBAF\S4\Seminar (+)\Codebeispiel> ./Binärbaum
- - - Erstellungsprozess Binaerbaum - - -
Aus wie vielen Knoten soll der Binaerbaum bestehen? 7

Knoten mit Element: 1 wurde erzeugt

Aktuelles Element: 1
->moegliches linkes Kind: 2
->moegliches rechtes Kind: 3

Knoten mit Element: 2 wurde erzeugt

Aktuelles Element: 2
->moegliches linkes Kind: 4
->moegliches rechtes Kind: 5

Knoten mit Element: 4 wurde erzeugt

Aktuelles Element: 4
->moegliches linkes Kind: 8
->moegliches rechtes Kind: 9

Knoten mit Element: 5 wurde erzeugt

Aktuelles Element: 5
->moegliches linkes Kind: 10
->moegliches rechtes Kind: 11

Knoten mit Element: 3 wurde erzeugt

Aktuelles Element: 3
->moegliches linkes Kind: 6
->moegliches rechtes Kind: 7

Knoten mit Element: 6 wurde erzeugt

Aktuelles Element: 6
->moegliches linkes Kind: 12
->moegliches rechtes Kind: 13

Knoten mit Element: 7 wurde erzeugt

Aktuelles Element: 7
->moegliches linkes Kind: 14
->moegliches rechtes Kind: 15

Erstellte Knoten: 7
- - - Erstellungsprozess beendet - - -

- - - Fertiger Binaerbaum - - -
Anmerkung: Blaetter werden mit ()-Klammern gekennzeichnet
Anmerkung: Durchnummerierung erfolgt in Level-Order-Verfahren
((4)<-[2]->(5))<-[1]->((6)<-[3]->(7))
```

Abbildung 14: Konsolenausgabe des Anwendungsbeispiels

Der zugehörige Quellcode zur Ausgabe:

```c
#include <stdio.h>
#include <stdlib.h>

struct Knoten {
    int element;
    struct Knoten* links;
    struct Knoten* rechts;
};

int zaehler(struct Knoten*);
struct Knoten* erstelleKnoten(int);
struct Knoten* erstelleBinaerbaum(int);
void baumrest(int, int, struct Knoten*);

//Alle nach dem Wurzelelement folgenden linken und rechten Knoten werden gezaehlt
int zaehler(struct Knoten* t) {
    if (t == NULL) {
    return 0;
    }
    return 1 + zaehler(t->links) + zaehler(t->rechts);
}

//Erstellen eines Knoten
struct Knoten* erstelleKnoten(int element) {
    struct Knoten* t = malloc(sizeof(struct Knoten));
    t->element = element;
    t->links = t->rechts = NULL;
    printf("Knoten mit Element: %d wurde erzeugt\n", element);
    return t;
}

//Zusammensetzen des Binaerbaums
struct Knoten* erstelleBinaerbaum(int zaehlerKnoten) {
    struct Knoten* baum = malloc(sizeof(struct Knoten));
    baum = erstelleKnoten(1);
    baumrest(zaehlerKnoten, 1, baum);
    return baum;
}

```

Abbildung 15: Teil 1 des Quellcodes

```c
//Baumrest wird erstellt (Baumrest = alle nachfolgende Elemente nach der Wurzel)
void baumrest(int nachf, int element, struct Knoten* baum) {
    int const kind_links  = 2*element;
    int const kind_rechts = 2*element+1;
    printf("\n");
    printf("Aktuelles Element: %d\n", element);
    printf("->moegliches linkes Kind: %d\n",  kind_links);
    printf("->moegliches rechtes Kind: %d\n\n", kind_rechts);
    if (element > nachf) {
        return;
    }
    else {
        baum->element = element;
        //Überprüfen ob linkes Kind mit gewuenschter Anzahl an Knoten erstellt werden kann
        //Falls linkes Kind nicht erstellen werden kann, kann das rechte Kind auch nicht erstellt werden
        if (kind_links <= nachf) {
            baum->links  = erstelleKnoten(kind_links);
            baumrest(nachf, kind_links, baum->links);
        if (kind_rechts <= nachf) {
            baum->rechts = erstelleKnoten(kind_rechts);
            baumrest(nachf, kind_rechts, baum->rechts);
        }
        }
    return;
    }
}

//Ausgabefunktion Binaerbaum
void binaerbaum(struct Knoten* baum) {
    if (baum->links != NULL) {
        printf("(");
        binaerbaum(baum->links);
        printf(")<-");
    }
    //Blaetter werden mit ()-Klammern markiert, innere Knoten und Wurzel mit []-Klammern
    if (baum->links != NULL && baum->rechts != NULL) {
        printf("[%d]",baum->element);
    }
    else {
        printf("%d",baum->element);
    }
    if (baum->rechts != NULL) {
        printf("->(");
        binaerbaum(baum->rechts);
        printf(")");
    }
}

```

Abbildung 16: Teil 2 des Quellcodes

```c
int main() {
    int zaehlerKnoten = 0;
    while (zaehlerKnoten<1) {
        printf("- - - Erstellungsprozess Binaerbaum - - - \n");
        printf("Aus wie vielen Knoten soll der Binaerbaum bestehen? ");
        scanf("%d", &zaehlerKnoten);
        printf("\n");
    }
    struct Knoten* Wurzel = erstelleBinaerbaum(zaehlerKnoten);
    printf("Erstellte Knoten: %d\n", zaehler(Wurzel));
    printf("- - - Erstellungsprozess beendet - - -\n\n");
    printf("- - - Fertiger Binaerbaum - - -\n");
    printf("Anmerkung: Blaetter werden mit ()-Klammern gekennzeichnet \n");
    printf("Anmerkung: Durchnummerierung erfolgt in Level-Order-Verfahren \n");
    binaerbaum(Wurzel);
    printf("\n");
}
```

Abbildung 17: Teil 3 des Quellcodes

8. Quellenverzeichnis

Literaturverzeichnis

Abschnitt 2: Aufbau Binärbaum:

-Unbekannter Autor (28.09.2021): Binärbaum; https://de.wikipedia.org/wiki/Bin%C3%A4rbaum

-Dr. Daniel Appel: Binärbäume; https://drdanielappel.de/informatik/mein-kleines-java-tutorium/java-im-abi-nrw/datenstrukturen/binaerbaeume/

-Unbekannter Autor: Binäre Bäume; http://www.inf.fu-berlin.de/lehre/WS11/infa/bintree.pdf

-Unbekannter Autor (22.09.2021): Baum (Datenstruktur); https://de.wikipedia.org/wiki/Baum_(Datenstruktur)

-YouTube Nutzer: „Bleeptrack“ (05.04.2013): Binärbäume + Traversierung; https://www.youtube.com/watch?v=xfhfRAccL2w

Abschnitt 3: Traversierung:

-YouTube Nutzer: „Bleeptrack“ (05.04.2013): Binärbäume + Traversierung; https://www.youtube.com/watch?v=xfhfRAccL2w

Abschnitt 4: Anwendungsbeispiele:

-Qastack-Nutzer: „BlueRaja“ und „iLiasT“: Was sind die Anwendungen von Binärbäumen?; https://qastack.com.de/programming/2130416/what-are-the-applications-of-binary-trees

-Ulrich Helmich (2021): Binäre Suchbäume; http://u-helmich.de/inf/kursQ1/folge17/folge17-2.html

-Unbekannter Autor: Binärer Suchbaum; https://studyflix.de/informatik/binaerer-suchbaum-1364

Abschnitt 5: Vergleich mit anderen Datenstrukturen:

-Ulrich Helmich (2021): Binäre Suchbäume; http://u-helmich.de/inf/kursQ1/folge17/folge17-2.html

Abschnitt 6: Suchkomplexität:

-IT-Swarm.Com-Nutzer: „Bunny Rabbit“ (18.11.2011): so berechnen Sie die binäre Suchkomplexität; https://www.it-swarm.com.de/de/algorithm/so-berechnen-sie-die-binaere-suchkomplexitaet/941377655/

-Unbekannter Autor (07.07.2019): Kekule-Nummer; https://de.wikipedia.org/wiki/Kekule-Nummer

-Unbekannter Autor (27.09.2021): Big O notation; https://en.wikipedia.org/wiki/Big_O_notation

Bilderverzeichnis

-Deckblatt: Unbekannter Autor: Binärer Suchbaum Binärer Baum Binärer Suchalgorithmus – Baum PNG; https://de.cleanpng.com/png-34epoz/

-Abbildung 1: Unbekannter Autor (28.09.2021): „Binärbaum mit Knotentypen“; aus Artikel: Binärbaum; https://de.wikipedia.org/wiki/Bin%C3%A4rbaum

-Abbildung 2: selbständig erstellt mit https://www.diagrammeditor.de/

-Abbildung 3: selbständig erstellt mit https://www.diagrammeditor.de/

-Abbildung 4: selbständig erstellt mit https://www.diagrammeditor.de/

-Abbildung 5: selbständig erstellt mit https://www.diagrammeditor.de/

-Abbildung 6: selbständig erstellt mit https://www.diagrammeditor.de/

-Abbildung 7: Qastack-Nutzer: „iLiasT“: Was sind die Anwendungen von Binärbäumen?; https://qastack.com.de/programming/2130416/what-are-the-applications-of-binary-trees

-Abbildung 8: selbständig erstellt mit https://www.diagrammeditor.de/

-Abbildung 9: Ulrich Helmich (2021): „Abbildung 1: Eine sortierte Liste von Zahle“; aus Artikel: Binäre Suchbäume; http://u-helmich.de/inf/kursQ1/folge17/folge17-2.html

-Abbildung 10: selbständig erstellt mit https://www.diagrammeditor.de/

-Abbildung 11: Ulrich Helmich (2021): „Ein binärer Suchbaum“; aus Artikel: Binäre Suchbäume; http://u-helmich.de/inf/kursQ1/folge17/folge17-2.html

-Abbildung 12: Ulrich Helmich (2021): „Abbildung 2: Ein sehr rechtslastiger Binärbaum“; aus Artikel: Binäre Suchbäume; http://u-helmich.de/inf/kursQ1/folge17/folge17-2.html

-Abbildung 13: Ulrich Helmich (2021): „Abbildung 3: Ein vollständig ausgeglichener Binärbaum“; aus Artikel: Binäre Suchbäume; http://u-helmich.de/inf/kursQ1/folge17/folge17-2.html

-Abbildung 14: selbständig erstellt

-Abbildung 15: selbständig erstellt

-Abbildung 16: selbständig erstellt

-Abbildung 17: selbständig erstellt