6. Suchalgorithmen

6.1 Allgemeines zum Suchen

Suchen gehört zu den häufigsten Aufgaben, die Computer im Alltag übernehmen – ob beim Auffinden von Internetadressen, Telefonnummern, Artikeln oder Dateien. Suchalgorithmen spielen daher eine zentrale Rolle in der Informatik.

Grundsätzlich unterscheidet man beim Suchen nach dem Zustand der Daten, die durchsucht werden sollen. Es ergeben sich dabei folgende typische Situationen:

1. Die Daten sind **unsortiert**.
2. Die Daten sind zwar sortiert, das Suchkriterium entspricht aber nicht dem Sortierkriterium.
3. Die Daten sind zwar sortiert, das Suchkriterium entspricht dem Sortierkriterium, allerdings sind die Daten nicht sehr suchfreundlich angeordnet.
4. Die Daten sind suchfreundlich sortiert, zum Beispiel in einem binären Suchbaum oder einem B-Baum, und das Suchkriterium entspricht dem Sortierkriterium.

Zu 1

Sind die Daten unsortiert, kann man sie nur ***sequentiell*** oder ***linear*** durchsuchen. Dabei werden die Elemente *nacheinander* durchsucht – beginnend beim ersten Eintrag, bis entweder das gesuchte Element gefunden oder das Ende der Daten erreicht ist.

Zu 2

Eine Sammlung von Büchern ist nach dem **Kaufdatum** sortiert, es wird aber nach einem bestimmten **Autor** gesucht. Sortierkriterium (Kaufdatum) und Suchkriterium (Autor) stimmen nicht überein. Daher kann nur sequentiell gesucht werden.

Zu 3

In einer Bibliothek stehen in 20 Regalen 10.000 Bücher, sortiert nach Autoren. Das erste Regal enthält die Autoren Aa bis Bm, das zweite Regal Bn bis Ce, das dritte Cf bis Da und so weiter.

Zu 4

In einer Bibliothek stehen in 26 Regalen 10.000 Bücher, sortiert nach Autoren. Das erste Regal enthält alle Autoren mit dem Anfangsbuchstaben A, das zweite Regal B, das dritte C und so weiter.

6.1.1 Vier Beispiele aus dem Alltag

Beispiel 1 aus dem Alltag - Interpolationssuche

Sie stehen vor dem Bücherregal, in dem Sie all die Romane aufbewahren, die Sie seit Jahren gesammelt haben. Die Romane sind alphabetisch nach Autor bzw. Autorin geordnet. Sie wollen nun ein Buch von Karl Melcher lesen. Dann werden Sie sicherlich nicht oben links im Regal bei "A" anfangen zu suchen, aber auch nicht unten rechts bei "Z", sondern werden gezielt irgendwo in der Mitte des Regals mit der Suche anfangen.

Sie haben quasi in Gedanken ***interpoliert***, wo im Regal der beste Einstiegspunkt für Ihre Suche ist.

Beispiel 2 aus dem Alltag - Indexsuche

Sie befinden sich in der Uni-Bibliothek und suchen ein ganz bestimmtes Buch. Leider kennen Sie nicht den oder die Autoren, sondern nur den Titel des Buches: *"Anwendung diverser Suchverfahren im Alltag"*. Früher hatte man für solche Zwecke Karteikästen. In einem Karteikasten waren die Karteikarten alphabetisch nach Autoren sortiert, in dem anderen Karteikasten hatte man die Karten systematisch nach Fachgebieten sortiert.

Sie gehen also zum Karteikasten mit den Fachgebieten, suchen nach dem Reiter "Informatik" und blättern dann - sequentiell - die Karten durch, bis Sie das gesuchte Buch gefunden haben - oder bis Sie beim nächsten Reiter angekommen sind und frustriert feststellen müssen, dass das Buch nicht in der Bibliothek vorhanden ist.

Aber wir nehmen einmal an, dass Sie die Karteikarte mit dem Buch gefunden haben. Auf der Karteikarte ist dann der Standort (Regal, Fach, Nummer) vermerkt, an dem das Buch steht. Diese Information ist dann Ihr ***Index***. Sie stehen auf, gehen zum Regal und suchen dann - wieder sequentiell - dort nach dem Buch.

Beispiel 3 aus dem Alltag - Binäre Suche

Sie wollen den Papierkorb Ihres Rechners löschen, das Betriebssystem unterbricht den Löschvorgang aber, weil Sie für irgendeine Datei keine Rechte besitzen.

Sie wollen jetzt diese Datei identifizieren. Sehr zu Ihrem Ärger befinden sich aber 2.600 Dateien im Papierkorb.

Die beste Strategie ist jetzt eine binäre Suche. Sie markieren die ersten 1.300 Dateien und versuchen, diese zu löschen. Wenn es nicht funktioniert, wissen Sie, dass sich die Problemdatei in diesen 1.300 Dateien befindet.

Also markieren Sie jetzt die obere Hälfte dieser Dateien, also die ersten 650. Der Löschversuch gelingt jetzt. Damit wissen Sie, dass sich die problematische Datei in den zweiten 650 Dateien befindet.

Diese halbieren Sie wieder und versuchen, die oberen 325 Dateien zu löschen und so weiter.

Auf diese Weise kommen Sie nach recht wenigen Schritten zur problematischen Datei und können versuchen, diese auf andere Weise zu löschen.

Kleine Kritik an diesem Beispiel

Handelt es sich bei dem Beispiel 3 wirklich um eine binäre Suche?

Nein, bei einer binären Suche müssen die Daten sortiert vorliegen, was hier nicht der Fall ist. Zwar sind die Dateien irgendwie sortiert, beispielsweise nach Name oder Erstellungsdatum, aber zum Suchen wird diese Sortierung nicht genutzt (Suchkriterium entspricht nicht dem Sortierkriterium).

Vielmehr handelt es sich um das ***Prinzip des binären Eingrenzens***: Wir teilen die Menge und testen, in welcher Hälfte die Problemdatei steckt.

Beispiel 4 aus dem Alltag - Sequentielle Suche

Sie haben von Ihrem Vater eine große Platten-Sammlung geerbt. Ihr Vater war ein sehr ordentlicher Mensch und hatte seine Platten-Sammlung gut sortiert. Allerdings nicht alphabetisch nach Interpreten, sondern nach dem Kaufdatum.

Sie wollen nun nachschauen, ob auch eine Platte von Ihrer Lieblingsband "The Luftschiff" dabei ist.

Wo fangen Sie an mit der Suche? Sie wissen, dass die ersten Platten dieser Gruppe schon 1984 erschienen sind, also zu der Zeit, als Ihr Vater mit dem Plattensammeln anfing. Aber vielleicht hat er diese Platte auch in den 90er Jahren gekauft oder um 2010 oder erst kurz vor seinem Tod im Jahre 2024.

Es ist also völlig egal, ob Sie ganz vorne oder ganz hinten in der Sammlung mit Ihrer Suche anfangen - aber anfangen müssen Sie ja schließlich.

Obwohl die Plattensammlung hoch geordnet ist, müssen Sie hier eine sequentielle Suche durchführen, da das Ordnungssystem nicht Ihrem Suchkriterium entspricht. Für Ihre Suche ist die Plattensammlung völlig unsortiert.

Hätte Ihr Vater mehrere Register oder Karteikästen angelegt, wäre die Suche kein Problem mehr: Sie suchen im Interpreten-Register nach "The Luftschiff" und finden dort das Kaufdatum. Wenn die Platte im Januar 2010 gekauft wurde, müssen Sie nicht ganz vorn mit der Suche anfangen, aber auch nicht ganz hinten, sondern vielleicht im hinteren Viertel der Sammlung. Dort können Sie dann binär oder sequentiell nach der Platte suchen.

6.2 Sequentielle Suche

Bei der sequentiellen (linearen) Suche wird eine Liste Element für Element durchlaufen, bis das gesuchte Element gefunden ist oder das Ende der Liste erreicht wird.

Dieses Verfahren setzt weder eine Sortierung noch eine spezielle Datenstruktur voraus und ist daher universell einsetzbar. Der Algorithmus ist sehr einfach zu implementieren und eignet sich insbesondere für kleine Datenmengen.

Auch große Datenmengen müssen sequentiell durchsucht werden, wenn sie nach einem anderen Kriterium sortiert sind, als es dem Suchkriterium entspricht.

Beispiel: Datensammlung mit 8.000.000 Datensätzen von Personen, die nach Namen sortiert sind. Wenn man jetzt alle Menschen finden will, die am 1. Januar 1991 geboren sind, müsste man den gesamten Datenbestand sequentiell durchsuchen.

6.2.1 Zeitverhalten der sequentiellen Suche

Sei **n** die Anzahl der Elemente in der Datensammlung und **S** die Suchzeit.

Günstigster Fall (Best Case): S = 1

Das gesuchte Element befindet sich an erster Stelle.

Ungünstigster Fall (Worst Case): S = n

Das gesuchte Element steht am Ende der Liste oder kommt gar nicht vor.

Durchschnittlicher Fall (Average case): S = (n+1)/2

In einer Liste mit sechs Elementen wird ein Element nach einem Vergleich gefunden, ein Element nach zwei Vergleichen und so weiter: S = (1+2+3+4+5+6) / 6 = 21/6 = 3,5.

Zeitkomplexität:

Die sequentielle Suche besitzt eine Zeitkomplexität von **O(n)**. Die Laufzeit wächst also linear mit der Anzahl der Elemente.

6.2.2 Implementierung einer sequentiellen Suche

Wir wollen nun eine sequentielle Suche in Java implementieren. Dazu verwenden wir einen int-Array `zahlen` aus 100 Zufallszahlen.

```
public int getIndex(int[] zahlen, int suchzahl)
{
   for (int i = 0; i < zahlen.length; i++)
      if (zahlen[i] == suchzahl)
         return i;
   return -1;
}
```

Diese Methode sucht in dem Array den **Index** der Suchzahl. Falls die Suchzahl nicht in dem Array vorhanden ist, wird **-1** zurückgegeben.

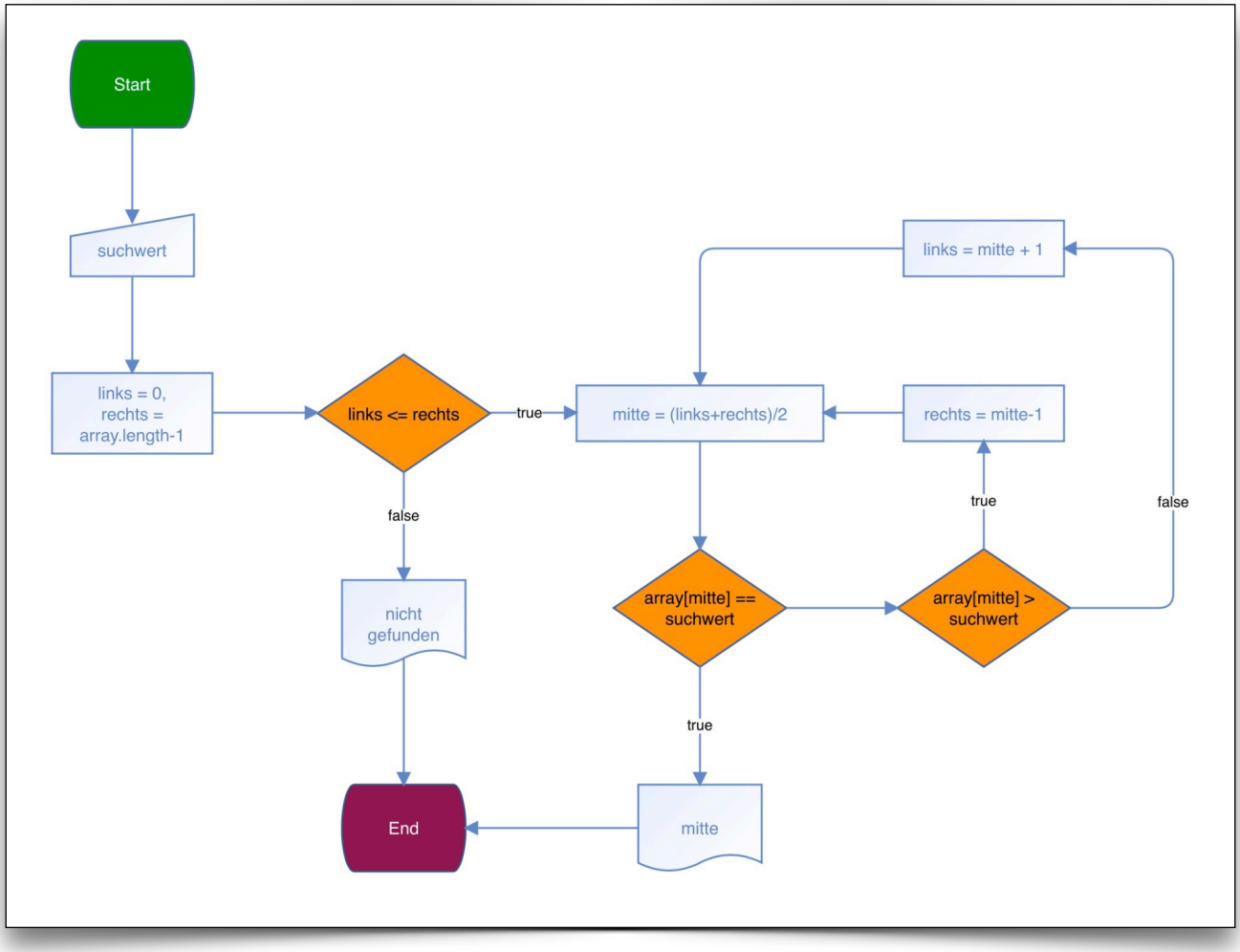
Eine alternative Implementierung mit einer while-Schleife sieht so aus:

```
public int getIndex(int[] zahlen, int suchzahl)
{
   int i = 0;
   while (i < zahlen.length)
   {
      if (zahlen[i] == suchzahl)
         return i;
      i++;
   }
   return -1;
}
```

6.3 Die binäre Suche

6.3.1 Der Algorithmus

Die binäre Suche erfolgt nach dem Prinzip **"Teile und herrsche"**. Das heißt, man teilt die zu durchsuchenden Daten in ein **mittleres Element** und **zwei Hälften links und rechts** davon. Sollte das mittlere Element mit dem Suchbegriff oder der Suchzahl übereinstimmen, ist die Suche erfolgreich beendet. Andernfalls wird entweder in der linken oder in der rechten Hälfte nach dem gleichen Verfahren weitergesucht.



Hier sehen wir ein Flussdiagramm, das den Algorithmus der binären Suche darstellt.

Achten Sie darauf, dass ein Flussdiagramm nicht die Syntax einer Programmiersprache einhalten muss, sondern relativ frei gestaltet werden kann.

6.3.2 Veranschaulichung der binären Suche

Gegeben ist ein int-Array aus 20 Zahlen, die Suchzahl ist 31.

Mit `mitte=(links+rechts)/2` berechnen wir den Index des mittleren Elements:

```
mitte = (0 + 19) / 2 = 9.5
```

das entspricht dem Index 9 und der Zahl 19:

1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Die Suchzahl 31 ist größer als 19 (*1. Vergleich*), das heißt, wir müssen rechts weiter suchen.

Die Zahl 21 an Index 10 ist jetzt die linke Grenze des neuen Suchintervalls, die rechte Grenze bleibt bestehen.

```
links = mitte + 1
```

```
mitte = (10 + 19) / 2 = 14.5
```

Die Zahl an Index 14, also die 29, ist jetzt die neue mittlere Zahl.

1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39

Die Suchzahl 31 ist größer als 29 (*2. Vergleich*) ⟹ Rechts weiter suchen.

```
mitte = (15 + 19) / 2 = 17.0
```

Das entspricht dem Index 17 und der Zahl 35:

1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39

Die Suchzahl 31 ist nicht größer als 35 (*3. Vergleich*) ⟹ Links weiter suchen.

```
mitte = (15 + 16) / 2 = 15.5
```

Das entspricht dem Index 15 und der Zahl 31 (*4. Vergleich*):

1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39

Damit ist die Suchzahl nach nur vier Vergleichen gefunden.

6.3.3 Quantitative Analyse der binären Suche

Die rot markierten Anweisungen protokollieren die **Zahl der Vergleiche**, die notwendig ist, um die Suchzahl zu finden bzw. um festzustellen, dass die Zahl nicht vorhanden ist.

```
public int vergleicheBinaer(int[] a, int suchzahl)
{
    int links = 0;
    int rechts = a.length - 1;
    int v = 0;

    while (links <= rechts)
    {
        int mitte = (links + rechts) / 2;

        v++;
        if (a[mitte] == suchzahl)
        {
            return v;
        }

        v++;
        if (suchzahl < a[mitte])
        {
            rechts = mitte - 1;
        }
        else
        {
            links = mitte + 1;
        }
    }

    return v;
}



public int vergleicheLinear(int[] a, int suchzahl)
{
    int v = 0;

    for (int i = 0; i < a.length; i++)
    {
        v++;
        if (a[i] == suchzahl)
        {
            return v;
        }
    }

    return v;
}
```

Die Methode `ermittleSuchzeit()` beginnt folgendermaßen:

```
public void ermittleSuchzeit(int arraygroesse)
{
    int[] zahlen = new int[arraygroesse];

    int vergleicheB = 0;
    int vergleicheL = 0;

    for (int i=0; i < zahlen.length; i++)
        zahlen[i] = i*2;
```

Es wird hier ein lokales Array in der angegebenen Größe erstellt. Das Array wird dann mit geraden Zahlen gefüllt, bei einem Array mit 100 Elementen als mit 0, 2, 4, ..., 196, 198.

Mit einer for-Schleife werden dann die Zahlen von 1 bis 64, 1 bis 128, 1 bis 256 etc. in diesem Array gesucht. Die Hälfte der Suchzahlen besteht aus ungeraden Zahlen, die nicht im Array enthalten sind. Dadurch wird die Suche noch realistischer.

Ausgabe des Testprogramms:

```
Lineare Suche für 64 Zahlen = 2608 Vergleiche
Binaere Suche für 64 Zahlen = 353 Vergleiche
Verhältnis Linear/Binär =         7,39

Lineare Suche für 128 Zahlen = 10336 Vergleiche
Binaere Suche für 128 Zahlen = 833 Vergleiche
Verhältnis Linear/Binär =        12,41

Lineare Suche für 256 Zahlen = 41152 Vergleiche
Binaere Suche für 256 Zahlen = 1921 Vergleiche
Verhältnis Linear/Binär =        21,42

Lineare Suche für 512 Zahlen = 164224 Vergleiche
Binaere Suche für 512 Zahlen = 4353 Vergleiche
Verhältnis Linear/Binär =        37,73

Lineare Suche für 1024 Zahlen = 656128 Vergleiche
Binaere Suche für 1024 Zahlen = 9729 Vergleiche
Verhältnis Linear/Binär =        67,44

Lineare Suche für 2048 Zahlen = 2622976 Vergleiche
Binaere Suche für 2048 Zahlen = 21505 Vergleiche
Verhältnis Linear/Binär =       121,97

Lineare Suche für 4096 Zahlen = 10488832 Vergleiche
Binaere Suche für 4096 Zahlen = 47105 Vergleiche
Verhältnis Linear/Binär =       222,67
```

Übung 6.1

Erstellen Sie ein solches Testprogramm!

Übung 6.2

Ermitteln Sie, ob die rekursive binäre Suche weniger Vergleiche benötigt als die nicht-rekursive Variante.

Übung 6.3

Eine KI hat den Quelltext der nicht-rekursiven Variante analysiert und meinte, dass folgender Quelltext noch besser ist:

```
public int vergleicheBinaerOptimiert(int[] a, int suchzahl)
{
    int links = 0;
    int rechts = a.length - 1;
    int v = 0;

    while (links <= rechts) {
        int mitte = (links + rechts) / 2;
        v++;
        if      (suchzahl < a[mitte]) rechts = mitte - 1;
        else if (suchzahl > a[mitte]) links  = mitte + 1;
        else return v; // gefunden
    }

    return v;
}
```

Vergleichen Sie diesen Quelltext mit dem vorherigen nicht-optimierten.

Analysieren Sie, ob die binäre Suche dadurch noch realistischer dargestellt wird.

Übung 6.4

Analysieren Sie die folgende Tabelle:

N	64	128	256	512	1024	2048	4096
Vergleiche V bei binärer Suche	353	883	1921	4353	9729	21505	47105
$\log_2(N)$	6	7	8	9	10	11	12
Quotient V / N	5,52	6,90	7,50	8,50	9,50	10,50	11,50

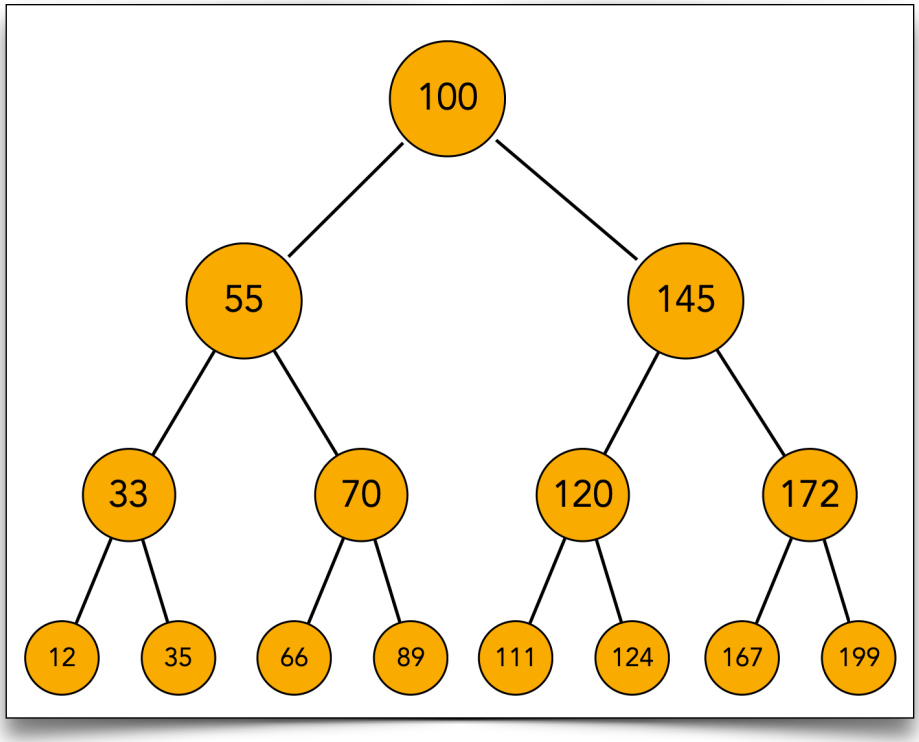
Begründen Sie mit den Daten aus der Tabelle, wieso das Zeitverhalten der binären Suche durch **O(log(N))** charakterisiert werden kann.

6.3.4 Binäre Suchbäume

Das in diesem Kapitel betrachtete binäre Suchverfahren eignet sich sehr gut für **bereits sortierte** Arrays oder andere Listen. Der zentrale Vorteil des Verfahrens liegt in seiner Effizienz: Durch das wiederholte Halbieren des Arrays kann ein gesuchtes Element in logarithmischer Zeit gefunden werden. Diese Effizienz ist jedoch an eine wesentliche Voraussetzung gebunden: **Die Daten müssen bereits sortiert vorliegen und dürfen sich während der Suche nicht verändern.**

Was aber, wenn die sortierten Daten ständig durch Einfügen neuer Daten und Löschen nicht mehr benötigter Daten durcheinander gebracht werden? **Große Datenbanken können nicht nach jeder Bearbeitung neu sortiert werden**, das wäre viel zu aufwendig, und der Vorteil, den die binäre Suche mit sich bringt, wäre wieder zunichte gemacht. Speichert man die Daten dagegen in einem binären Suchbaum, hat man diese Probleme nicht.

Binäre Suchbäume übertragen das Grundprinzip der binären Suche - also das schrittweise Vergleichen und Eingrenzen des Suchbereichs - auf eine dynamische Datenstruktur. Die Ordnung der gespeicherten Werte liegt dabei nicht wie bei einem sortierten Array einfach als feste Reihenfolge vor, sondern ergibt sich aus der Anordnung der Knoten im Baum. Dadurch können Suchen, Einfügen und Löschen von Daten effizient durchgeführt werden, ohne dass die gesamte Struktur jedes Mal neu sortiert werden muss.



Dieses Bild zeigt einen binären Suchbaum. In einem solchen Baum sind alle Elemente im linken Teilbaum eines Knotens kleiner oder gleich dem Element des Knotens selbst, während alle Elemente im rechten Teilbaum größer sind. Die Daten sind daher bereits suchfreundlich organisiert.

Beispiel 1:

Wir suchen die Zahl 124 und müssen dazu folgende Vergleiche durchführen:

1. Vergleich mit 100 → im rechten Nachfolger der 100 weiter suchen
2. Vergleich mit 145 → im linken Nachfolger der 145 weiter suchen
3. Vergleich mit 120 → im rechten Nachfolger der 120 weiter suchen
4. Vergleich mit 124 → Suchzahl gefunden nach vier Vergleichen.

Beispiel 2:

Wir suchen die Zahl 125 und müssen dazu folgende Vergleiche durchführen:

1. Vergleich mit 100 → im rechten Nachfolger der 100 weiter suchen
2. Vergleich mit 145 → im linken Nachfolger der 145 weiter suchen
3. Vergleich mit 120 → im rechten Nachfolger der 120 weiter suchen
4. Vergleich mit 124 → im rechten Nachfolger der 124 weiter suchen

Es existiert kein rechter Nachfolger der 124 → Suchzahl nicht im Baum enthalten!

Die hohe Geschwindigkeit der Suche in einem binären Suchbaum liegt darin begründet, dass mit jeder Entscheidung die Anzahl der noch zu durchsuchenden Elemente ungefähr **halbiert** wird. Dadurch wird jede Zahl im oben dargestellten Baum oder die Feststellung ihrer Abwesenheit nach maximal vier Vergleichen getroffen.

Das **Zeitverhalten** der Suche in einem binären Suchbaum ist **O(log(N))**.

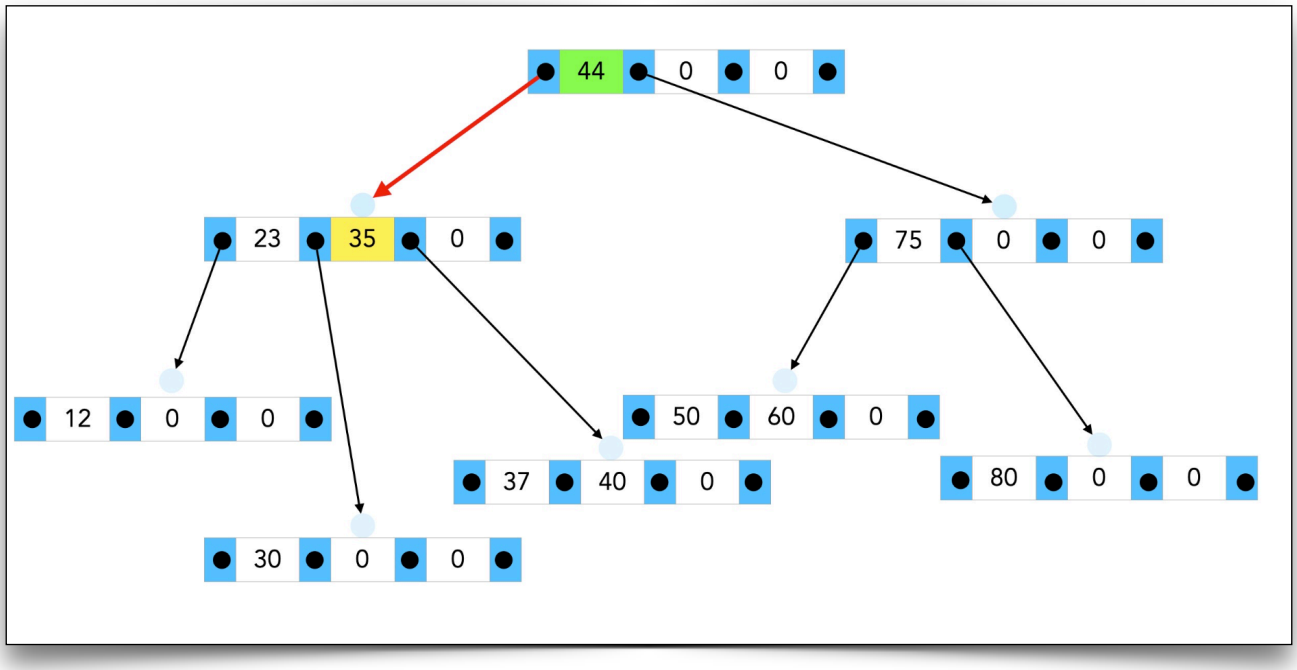
Betrachten wir unseren Suchbaum in der Abbildung mit 15 Elementen: Der Zweierlogarithmus von 16 ist 4. Ein binärer Suchbaum mit 31 Elementen hätte eine zusätzliche Ebene, wodurch jede Zahl nach maximal fünf Vergleichen gefunden würde. Bei einem Suchbaum mit 63 Elementen wären es maximal sechs Vergleiche.

Die gesamte Bevölkerung Deutschlands könnte man mit maximal 28 Vergleichen in einem binären Suchbaum finden, wenn es denn einen solchen gäbe.

6.3.5 B-Bäume

Für sehr große Datenmengen, insbesondere wenn diese nicht vollständig im Hauptspeicher gehalten werden können, stoßen einfache binäre Suchbäume allerdings an ihre Grenzen. Hier kommen die von Rudolf BAYER und Edward MCCREIGHT Anfang der 70er Jahre entwickelten B-Bäume ins Spiel. Diese B-Bäume erweitern das Konzept des Suchbaums, sodass auch in externen Speichern wie Festplatten oder SSDs effizient gesucht werden kann. B-Bäume bilden sozusagen das Fundament moderner Datenbank- und Dateisysteme.

Ein ***B-Baum*** ist ein ausgeglichener Suchbaum, bei dem jeder Knoten maximal **N Werte und N+1 Nachfolger-Knoten** enthalten darf, wodurch die Baumhöhe gering bleibt und er besonders effizient für den Einsatz in Datenbanken und Dateisystemen geeignet ist.



Dieses Bild zeigt einen einfachen B-Baum mit N = 3.

Wenn Sie mehr über B-Bäume wissen möchten, gehen Sie auf die Seite https://u-helmich.de/inf/kursQ1/folge18/folge18-7.html.

Zur Klausur

Sie sollten schon wissen, was man unter einem binären Suchbaum und einem B-Baum versteht. Einzelheiten zu deren Implementierung werden allerdings erst im Kurs "Algorithmen und Datenstrukturen" behandelt und sind daher nicht für unsere Klausur relevant. Aber welche Vorteile solche Bäume gegenüber sortierten linearen Listen haben (schnelles Einfügen und Löschen), sollten Sie schon wissen und auch kurz begründen können.

6.4 Weitere einfache Suchverfahren

6.4.1 Die Sprungsuche

Man springt im sortierten Array in festen Schritten vorwärts, bis der Block gefunden ist, in dem die Suchzahl liegen muss. Danach durchsucht man nur noch diesen Block linear.

1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39
0				4				8				12				16			

Vergleich 1 <= 31 = true → 4 Elemente weiter

Vergleich 9 <= 31 = true → 4 Elemente weiter

Vergleich 17 <= 31 = true → 4 Elemente weiter

Vergleich 25 <= 31 = true → 4 Elemente weiter

Vergleich 33 <= 31 = false → zurück zum vorherigen Block (Index 12+1)

Ab hier: Lineare Suche

Vergleich 27 <= 31 = false → 1 Element weiter

Vergleich 29 <= 31 = false → 1 Element weiter

Vergleich 31 <= 31 = false → 1 Element weiter

Zeitverhalten O(N), aber deutlich schneller als sequentielle Suche.

Optimale Sprungweite

Die optimale Sprungweite **m** berechnet sich nach **m = sqrt(n)**, wobei **n** die Zahl der zu durchsuchenden Elemente ist.

Bei einem Array der Größe 1000 wäre die optimale Sprungweite also **sqrt(1000) = 32** (gerundet).

Zwei-Ebenen-Sprungsuche

Bei einem Array der Größe 1.000.000 wäre die optimale Sprungweite **m = 1000**. Statt jetzt den gefundenen Zielabschnitt linear oder binär zu durchsuchen, bietet sich hierfür eine weitere Sprungsuche an. Man spricht dann von einer ***Zwei-Ebenen-Sprungsuche***.

Anwendung:

Die Sprungsuche eignet sich am besten dann, wenn die Zahlen bzw. zu suchenden Elemente weitgehend **gleichmäßig verteilt** sind.

6.4.2 Die Indexsuche

Man führt ein zusätzliches Index-Array, um schnell den passenden Wertebereich einzugrenzen. Anschließend sucht man nur noch im zugehörigen Teilbereich, meistens linear oder binär oder ebenfalls mit einer Indexsuche. Im letzteren Fall spricht man von einer ***mehrstufigen Indexsuche***.

Beispiel 1

Das Haupt-Array:

1	4	5	9	12	13	15	16	18	19	20	25	32	34	38	44	46	49	52	55	58	63	64	67	74	77
0				4						10		12			15			18			21			24	

Das Index-Array:

0	4	10	12	15	18	21	24
0	1	2	3	4	5	6	7
Start	erste Zahl >= 10	erste Zahl >= 20	erste Zahl >= 30	erste Zahl >= 40	erste Zahl >= 50	erste Zahl >= 60	erste Zahl >= 70

Beschreibung

Oben sehen wir ein Array aus 26 sortierten int-Zahlen im Bereich zwischen 1 und 77.

Darunter ist ein ***Index-Array*** gezeigt, dieses besteht aus 8 int-Zahlen. Die Werte dieses Index-Arrays verweisen auf bestimmte Indizes in dem Haupt-Array.

Im Element mit dem Index 1 steht beispielsweise die Zahl 4. Das ist die Position der ersten Zahl im ***Haupt-Array***, die größer oder gleich 10 ist.

An Position 2 im Index-Array steht die Zahl 10. An Index 10 im Haupt-Array befindet sich die erste Zahl >= 20. Entsprechend sind die Startwerte der nächsten Zehner-Intervalle an den Positionen 3 bis 7 des Index-Arrays gespeichert: 12, 14, 18, 21 und 24.

Suchbeispiel

Wir wollen nun die Zahl 44 suchen.

- Wir dividieren 44 durch 10 (ganzzahlig) und erhalten als Ergebnis 4.
- Dann schauen wir im Index-Array an Position 4 nach und sehen dort den Index 14.
- Wir springen in das Haupt-Array an Position 14 und sehen dort die 38. Das ist unser Startpunkt für die eigentliche Suche.
- Die nächste Zahl stimmt bereits mit der Suchzahl überein.

Beispiel 2

In dem zweiten Beispiel wollen wir die Indexsuche auf ein Array anwenden, das aus 10.000 Schlüsselwerten besteht, die im Bereich zwischen 1 und 20.000 liegen und auf die eigentlichen Daten verweisen.

In einem Hilfs-Array werden jetzt bestimmte Indizes des Haupt-Arrays gespeichert:

- hilf[0] = Index der ersten Zahl >= 2.000 ist 1.567
- hilf[1] = Index der ersten Zahl >= 4.000 ist 3.455
- hilf[2] = Index der ersten Zahl >= 6.000 ist 4.678
- hilf[3] = Index der ersten Zahl >= 8.000 ist 4.899
- hilf[4] = Index der ersten Zahl >= 10.000 ist 5.012
- hilf[5] = Index der ersten Zahl >= 12.000 ist 6.301
- hilf[6] = Index der ersten Zahl >= 14.000 ist 6.900
- hilf[7] = Index der ersten Zahl >= 16.000 ist 7.520
- hilf[8] = Index der ersten Zahl >= 18.000 ist 8.910

Angenommen, wir suchen jetzt den Schlüsselwert 12.345. Dann würden wir zunächst im Hilfs-Array nachschauen, bei welchem Index die Zahlen >= 12.000 beginnen. Das ist dann die Position 6.301.

Die eigentliche Suche im Hauptarray beginnt dann bei genau diesem Index. Der Abschnitt umfasst immer ca. 600 Zahlen, also könnte man hier vielleicht statt einer linearen Suche noch eine einfache binäre Suche einsetzen.

Zeitverhalten der Indexsuche

Das Zeitverhalten der Indexsuche ist O(N), wie bei der sequentiellen Suche.

Trotzdem ist die Indexsuche deutlich schneller als die sequentielle Suche. Solche Zeitvorteile werden in der O-Notation allerdings nicht berücksichtigt. Entscheidend ist nur der Unterschied: **logarithmisch < linear < quadratisch < exponentiell**.

6.4.3 Die Interpolationssuche

Wenn die Zahlen bzw. Elemente in dem Hauptarray einigermaßen gleichmäßig verteilt sind, benötigt man keinen Index-Array. Stattdessen kann man versuchen, die Einstiegsposition für die Suche zu *berechnen*, also zu interpolieren. Daher wird dieses Suchverfahren auch als ***Interpolationssuche*** bezeichnet.

1	3	5	8	11	13	15	17	18	23	25	27	28	29	30	**31**	33	38	39	41
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	**15**	16	17	18	19

Grundlegende Berechnungen

Spannweite

Zunächst einmal müssen wir die ***Spannweite*** der Werte in dem Array ermitteln. Dazu subtrahieren wir den kleinsten Wert vom größten Wert des Arrays. Bei einem sortierten Array ist das kein Problem, der kleinste Wert steht an Index `0`, der größte Wert an Index `length-1`.

Für unser Beispiel-Array beträgt die Spannweite also: 41 - 1 = 40.

Zahl der Intervalle

Die Zahl der Intervalle ist um 1 kleiner als die Zahl der Arrayelemente. Bei unserem Array haben wir es also mit 19 Intervallen zu tun.

Wertanstieg pro Index

Als Nächstes berechnen wir, wie stark die Werte des int-Arrays von Intervall zu Intervall ansteigen. Dazu dividieren wir die Spannweite durch die Zahl der Intervalle:

40 / 19 = 2,1

Pro Index steigt der Wert der Zahlen um ca. 2,1.

Interpolierter Index

Die Formel für die Berechnung des wahrscheinlichsten Einstiegspunkts lautet:

(Suchzahl - kleinster Wert) / Wertanstieg pro Index

Suchbeispiele

Suche nach der 31

Berechnung der möglichen Position der Suchzahl:

(31 - 1) / 2,1 = 14

Der kleinste Wert des Arrays ist 1, daher wird hier und in den beiden folgenden Beispielen die entsprechende Zahl eingesetzt.

Ab Index 14 beginnt dann die lineare Suche. Nach einem Vergleich wird die 31 gefunden.

Suche nach der 13:

Berechnung der möglichen Position der Suchzahl:

(13 - 1) / 2,1 = 5

Suchzahl wird sofort gefunden.

Suche nach der 25:

Berechnung der möglichen Position der Suchzahl:

(25 - 1) / 2,1 = 11

An Index 11 befindet sich die 27 - also wird rückwärts weiter gesucht.

Nach einem weiteren Vergleich wird die 25 gefunden.

Das wohl bekannteste Praxisbeispiel für eine Interpolationssuche ist die Suche in einem Telefonbuch. Wenn Sie in einem dicken Telefonbuch der Stadt Hamburg nach einem Namen wie "Wehemeier" suchen, fangen Sie bestimmt nicht in der Mitte des Buches an, sondern schlagen es gleich weiter hinten auf. Sie schätzen also den Einstieg in die Suche ab und ermitteln eine günstige Anfangsposition.

Auch wenn Sie in einem dicken Hochschul-Lehrbuch im Register nach einem Fachbegriff wie "Binäre Suche" suchen, werden Sie nicht bei "A" oder "Z" anfangen, sondern abschätzen, wo ungefähr die Stichwörter mit "Bi" stehen.

Das Register des berühmten Hollemann/Wiberg, Lehrbuch der Anorganischen Chemie, besteht beispielsweise aus 95 eng bedruckten Seiten. Wenn Sie dort nach "Bleioxid" suchen, werden sich ganz sicher eine Art Interpolationssuche anwenden.

6.5 Vergleich der nicht-linearen Suchverfahren

Wir haben nun

die binäre Suche

die Sprungsuche

die Indexsuche und

die Interpolationssuche

als nicht-lineare Suchverfahren kennengelernt. Alle vier Verfahren funktionieren nur, wenn die zu durchsuchende Sammlung von Elementen aufsteigend oder absteigend sortiert ist, wobei natürlich das Sortierkriterium mit dem Suchkriterium übereinstimmen muss. Die Suche nach einem Interpreten in einer nach Kaufdatum sortierten CD-Sammlung kann natürlich nur linear (sequentiell) erfolgen.

In diesem Abschnitt wollen wir nun das Zeitverhalten der vier nicht-linearen Suchalgorithmen experimentell erforschen

6.5.1 Die Klasse VergleicheSuchverfahren

Auf den oben verlinkten Webseiten finden Sie kurze Quelltexte zu den einzelnen Suchverfahren, die aber noch nicht lauffähig sind. Sie müssen jeweils noch in eine Java-Klasse eingebettet werden. Diese Aufgabe sollte für Sie aber kein Problem darstellen.

Ich habe dann die ChatGPT gebeten, aus den vier von mir implementierten Java-Methoden (auf den Webseiten vorhanden) eine große Klasse **VergleicheSuchverfahren** zu erstellen, mit der die vier Suchverfahren verglichen werden können.

Meine Vorgaben dabei waren:

- Es sollen aufsteigend sortierte int-Arrays mit 64, 128, 256, 512, 1024 und 2048 Elementen untersucht werden.
- Der Wertebereich der Arrayelemente soll zwischen 1 und 5000 liegen.
- Bei den vier Algorithmen soll nur die Zahl der erforderlichen Vergleiche mitgezählt werden. Eine Methode wie sprungsuche() liefert also nicht den Index der gefundenen Zahl (oder -1) zurück, sondern die Anzahl der erforderlichen Vergleiche.

Den kompletten und sehr langen Quelltext dieser von ChatGPT erstellten Klasse können Sie hier als Java-Datei herunterladen:

Java-Datei VergleicheSuchverfahren.java

Die Klasse wurde unter BlueJ und IntelliJ getestet, sie funktioniert dort einwandfrei.

6.5.2 Ergebnisse der Vergleiche

Wir wollen uns nun die Ergebnisse dieser Untersuchung anschauen und betrachten dazu die Konsolenausgabe des Programms:

```
Vergleich von vier Suchverfahren

Suchanfragen: alle Zahlen von 1 bis 5000

       n           Sprungsuche            Indexsuche         Interpolation           Binaersuche
-------------------------------------------------------------------------------------------------
      64      91032 /  18,21      83743 /  16,75      29934 /   5,99      59982 /  12,00
     128     114770 /  22,95     115771 /  23,15      29870 /   5,97      69712 /  13,94
     256     151441 /  30,29     179692 /  35,94      29742 /   5,95      79290 /  15,86
     512     199152 /  39,83     307549 /  61,51      29486 /   5,90      88504 /  17,70
    1024     271109 /  54,22     562964 / 112,59      28972 /   5,79      96960 /  19,39
    2048     367891 /  73,58    1073763 / 214,75      27950 /   5,59     103886 /  20,78
```

Links sieht man jeweils die Gesamtzahl der Vergleiche für alle 5000 Suchzahlen, rechts daneben jeweils den Mittelwert für das Suchen nach einer Zahl.

Es stellt sich heraus, dass - wie bereits vermutet - die Interpolationssuche das schnellste Verfahren ist, gefolgt von der Binärsuche. Die Sprungsuche ist auf Platz 3, und am langsamsten ist die Indexsuche.

Auffällig ist insbesondere das Verhalten der ***Interpolationssuche*** bei wachsender Arraygröße n. Während bei den anderen Verfahren die Zahl der Vergleiche mit zunehmendem n mehr oder weniger deutlich ansteigt, bleibt sie bei der Interpolationssuche nahezu konstant. Das ist mathematisch gut erklärbar: Bei gleichmäßig verteilten Werten kann die Position der Suchzahl sehr genau abgeschätzt werden, sodass häufig schon der erste berechnete Zugriff direkt zum Ziel führt oder nur noch sehr wenige Nachvergleiche nötig sind.

Die ***Binärsuche*** zeigt ebenfalls ein recht günstiges Verhalten. Ihre Zahl der Vergleiche wächst nur logarithmisch mit n, also ungefähr proportional zu log2(n). Selbst bei einer Verdopplung der Arraygröße steigt der Aufwand daher nur um etwa einen zusätzlichen Vergleich.

Die ***Sprungsuche*** ist deutlich langsamer als die Binärsuche, aber immer noch schneller als die Indexsuche.

Bei der ***Indexsuche*** kann ein Hilfsarray den Einstieg in die Suche verbessern, doch entsteht ein zusätzlicher Aufwand durch das Suchen im Index und durch die anschließende Suche im betreffenden Intervall.