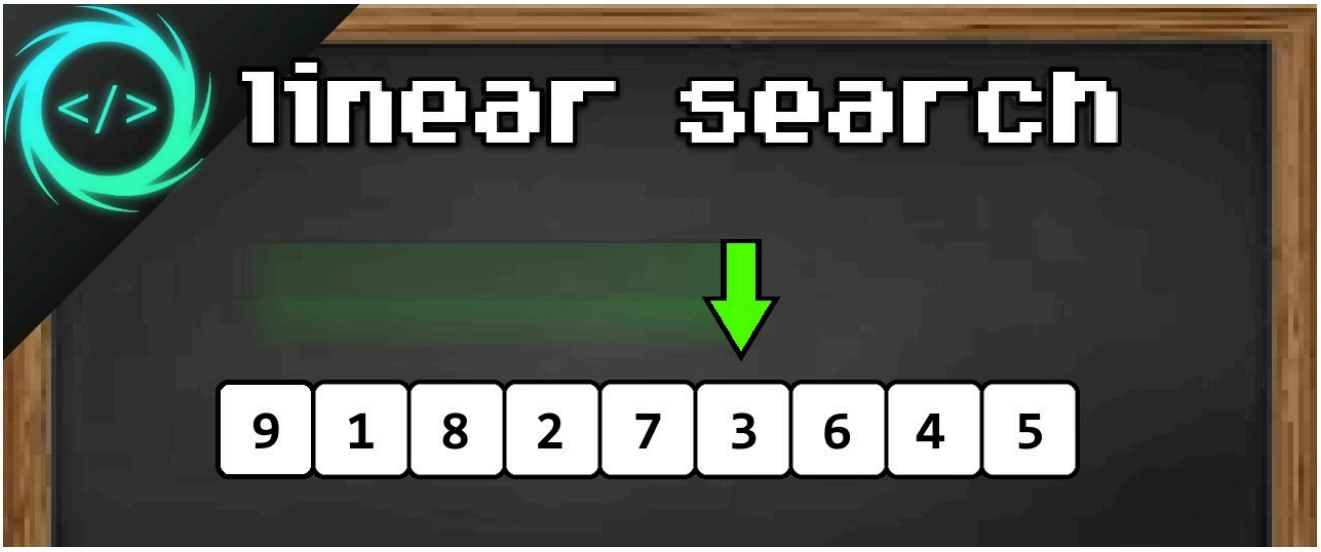
Such-Algorithmen

Lineare Suche →→



Vorgehen

Die Sammlung wird durchiteriert. Element für Element. Jedes Element wird dann untersucht ob es das gesuchte Element ist. Ziemlich grundlegender Suchalgorithmus ohne Schnickschnack.

Vorteil

Braucht keine Sortierte Sammlung. Schnell für kleine Datenmengen.

Nachteil

Laufzeit steigt Linear zur Datenmenge.

Merkmale

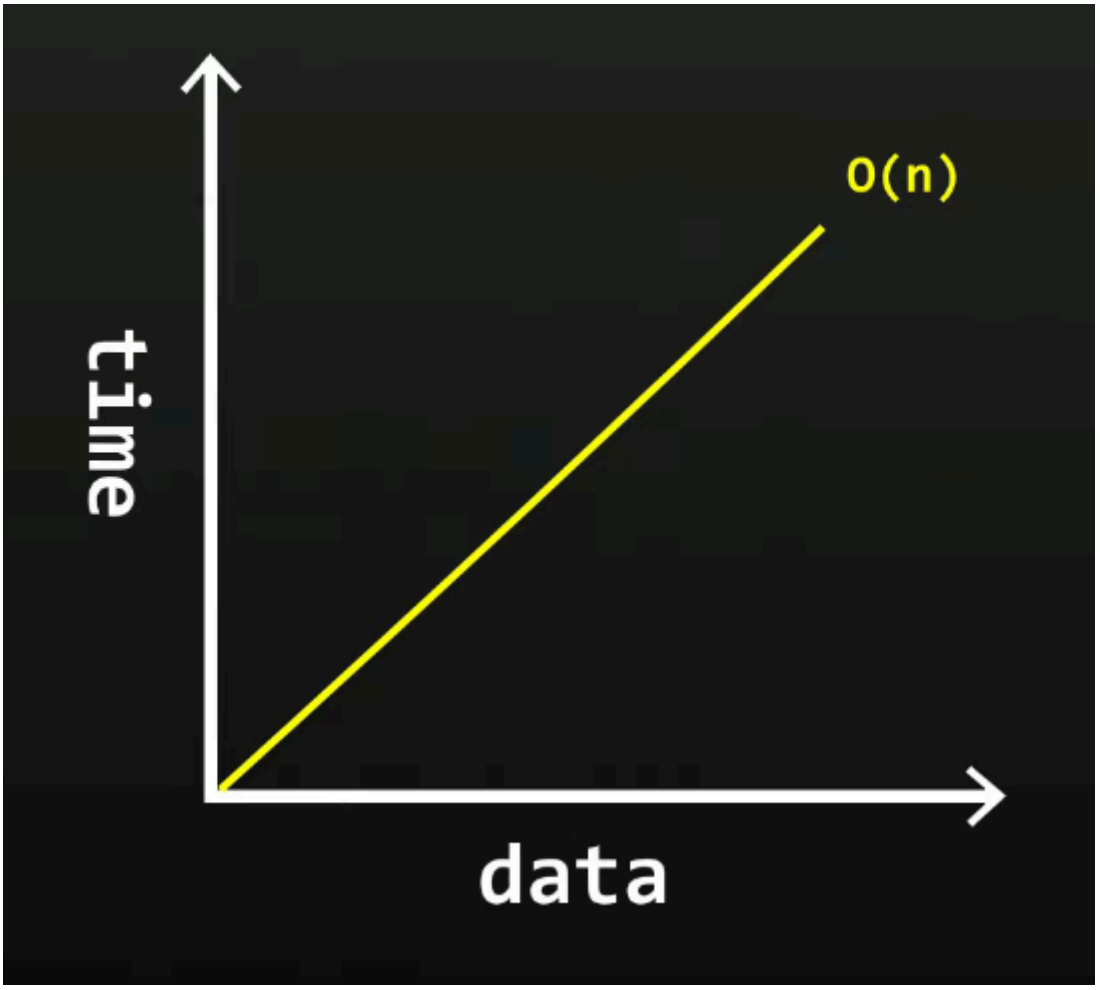
Eine einfache for-Schleife die über die Länge der gesamten Sammlung geht, array.length. Jedes Element auf dem aktuellen Index wird mit dem gesuchten Wert verglichen.

Beispielcode

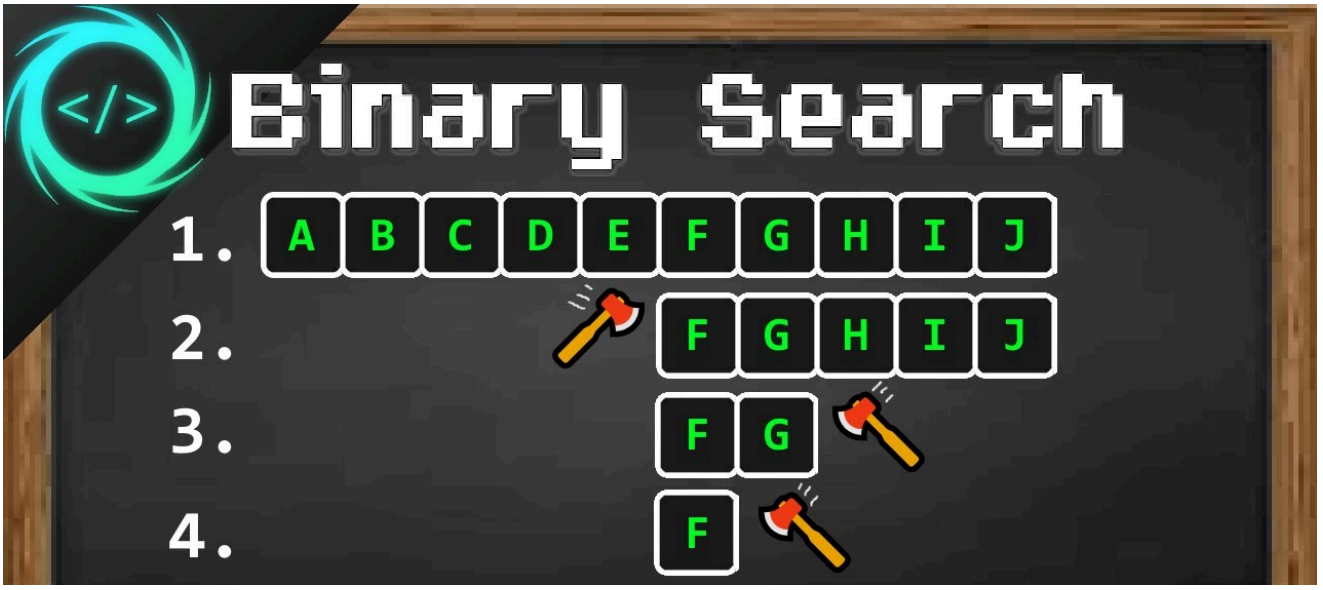
```
// ------------------------
// ---- LINEAR-SEARCH ----
// https://www.youtube.com/watch?v=246V51AWwZM

private static int linearSearch(int[] array, int searchValue) {
    for (int i = 0; i < array.length; i++) {
        if (array[i] == searchValue) {
            return i;
        }
```

```
    }
    return -1; // return negative Value if element is not found!
}
```

Runtime Complexity / Big 0



Binäre Suche



Vorgehen

Suchalgorithmus der bei jeder Iteration die Hälfte der Sammlung eliminiert. Auf einer sortierten Sammlung wird immer der Index in der Mitte gesetzt und mit dem gesuchten Wert verglichen. Dann wird entschieden ob die rechte Hälfte oder die linke Hälfte eliminiert wird. Solange bis nur noch das gesuchte Element übrig bleibt.

Vorteil

Sehr gute Laufzeit bei großen Datenmengen, da mit jeder Iteration 50% eliminiert wird.

Nachteil

Eine sortierte Sammlung ist zwingend erforderlich.

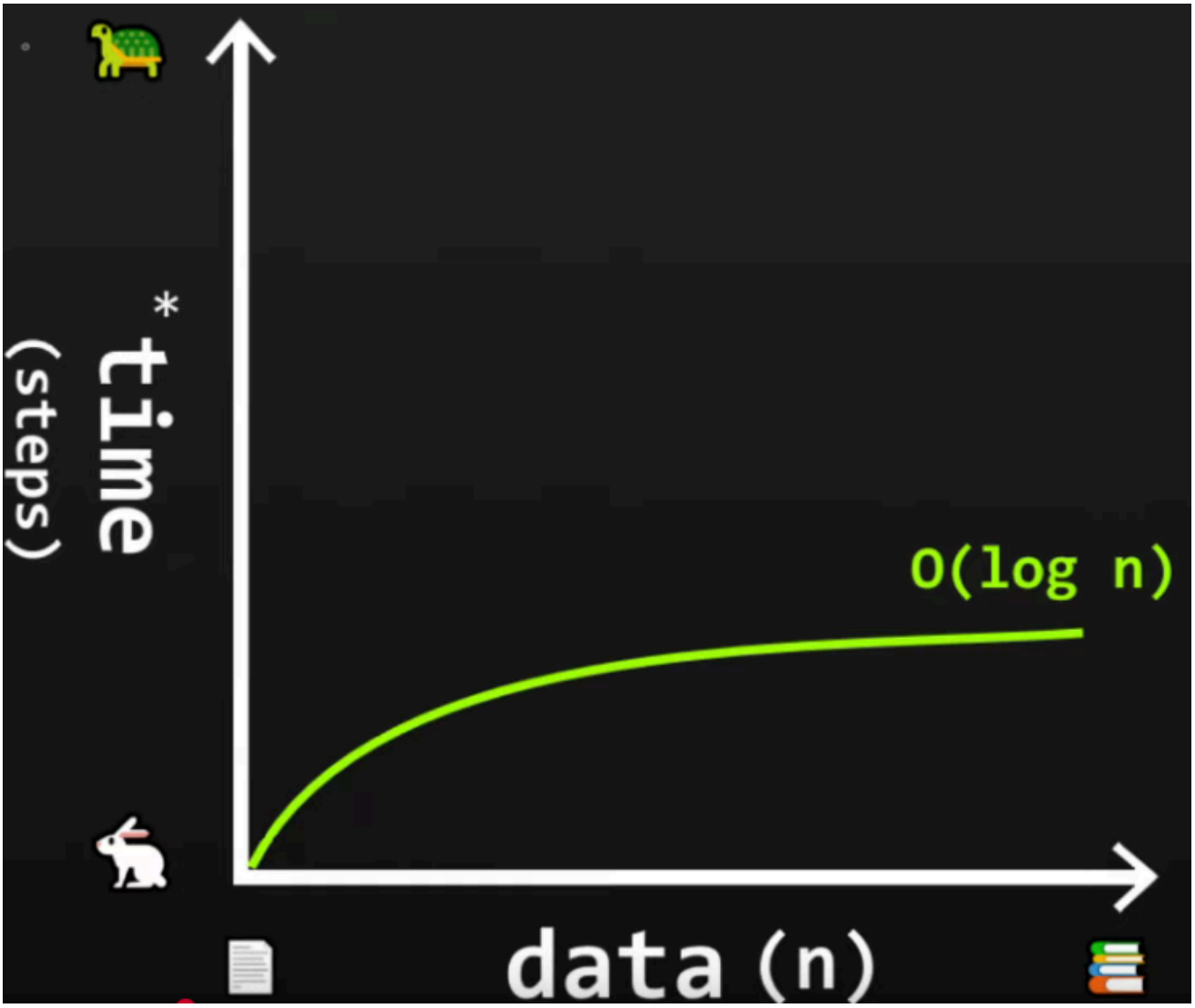
Merkmale

Die übergebene Sammlung muss sortiert sein. Es gibt drei Hilfsvariablen: ein "hoch", ein "tief" und eine "mitte". Die Formel für die Mitte muss in der while-Schleife auftauchen. `middle = low + (high - low) / 2`

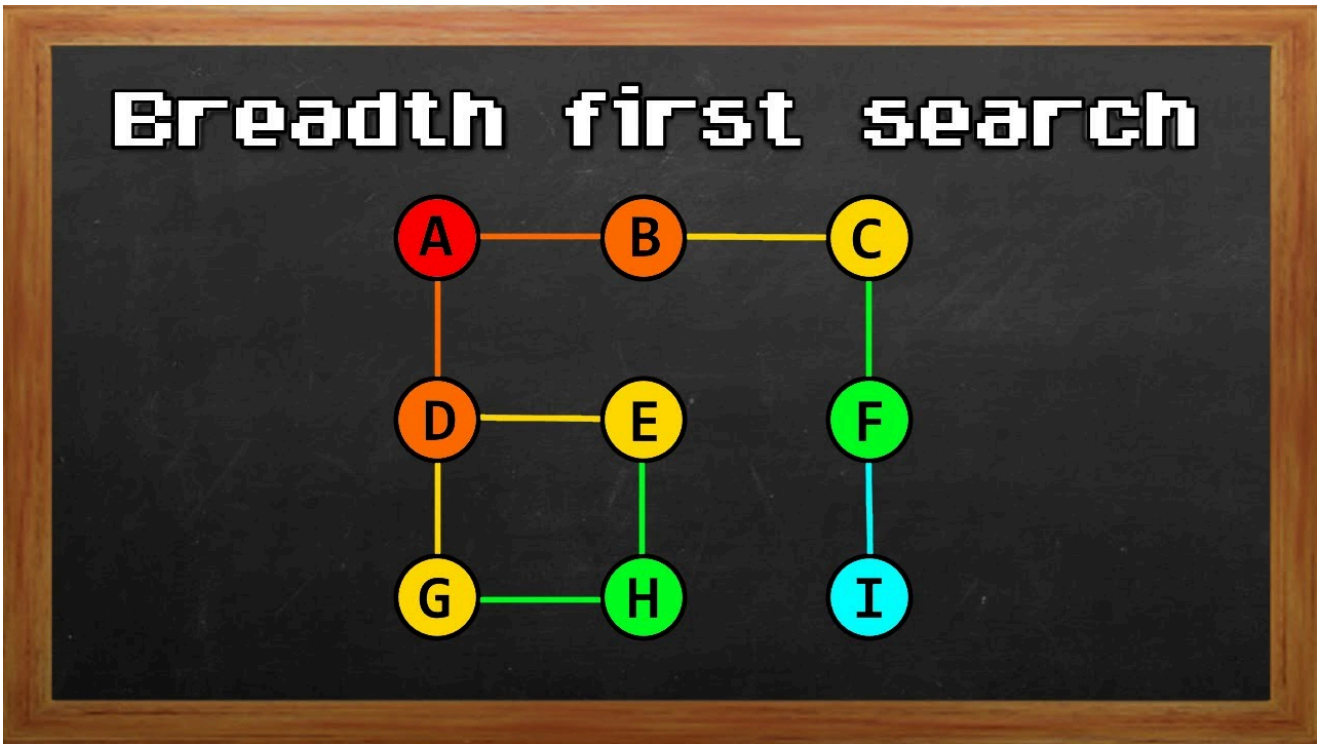
Beispielcode

```
// ----------------------
// ---- BINARY-SEARCH ----
// https://www.youtube.com/watch?v=xrMppTpoqdw

private static int binarySearch(int[] sortedArray, int searchValue){
    int low = 0;
    int high = sortedArray.length - 1;

    while (low <= high) {
        int middle = low + (high - low) / 2;
        int middleValue = sortedArray[middle];

        if (middleValue < searchValue){
            low = middle + 1;
        } else if (middleValue > searchValue) {
            high = middle - 1;
        } else {
            return middle;
        }
    }
    return -1; // negative Value if element is not found!
}
```

Runtime Complexity / Big 0



Breiten Suche / Breadth First Search ← 🔍 →



Vorgehen

Wird verwendet um einen Graphen oder einen Datenbaum zu durchsuchen. Keine Liste! Mit dieser Suche wird jede Ebene einer Sammlung komplett durchsucht. Im Gegenteil von einem Zweig pro Suche, siehe Tiefensuche.

Vorteil

Sehr zuverlässig egal ob bei gerichteten oder ungerichteten Graphen.

Nachteil

Hat einen Hohen Speicherverbrauch, da jeder Knoten in jeder Ebene vermerkt werden muss.

Merkmale

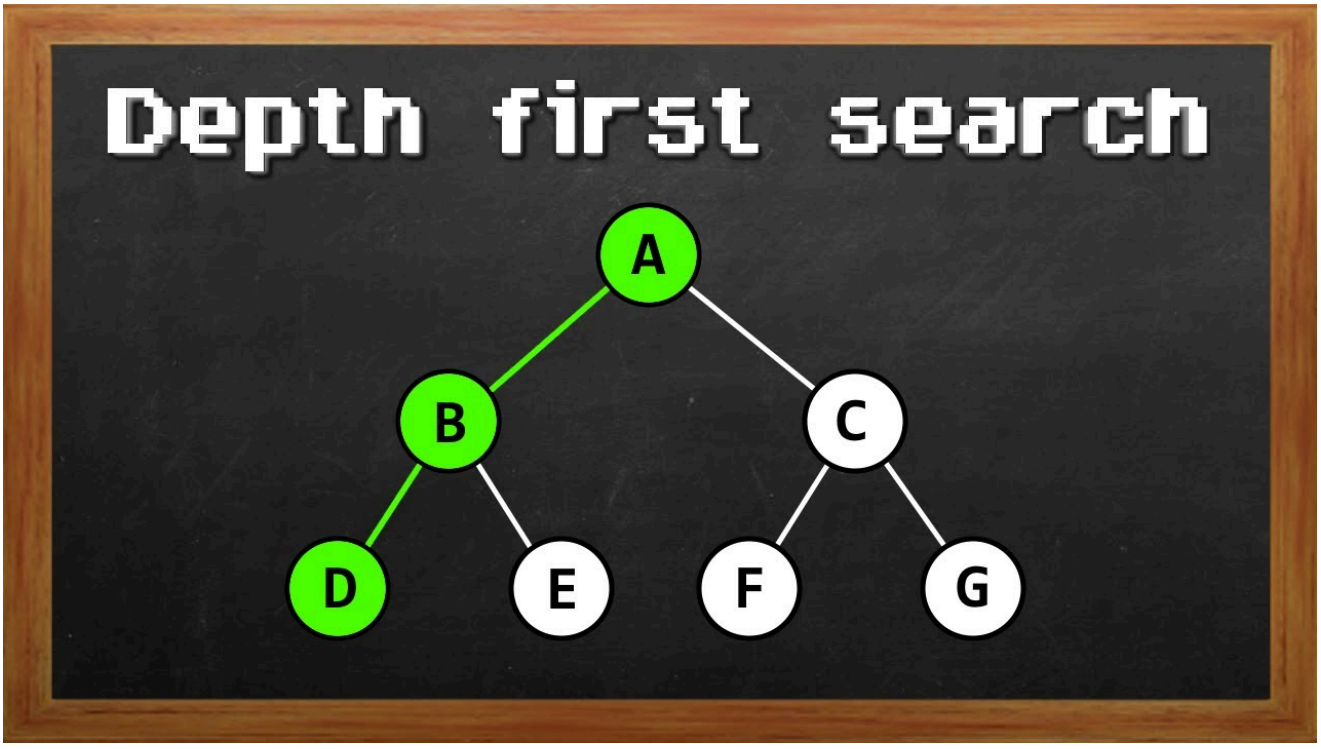
Verwendet immer eine Warteschlange (im Beispiel "Queue"). Muss besuchte Knoten markieren, im Beispiel mit einen Boolean visited = true.

Beispielcode

```
// ------------------------------
// ---- Breadth-First-Search ----
// https://www.youtube.com/watch?v=7Cox-J7onXw

BreadtgFirstSearch(Graph, Startknoten):
    Initialisiere eine leere Warteschlange (Queue)
    Füge den Startknoten zur Queue hinzu
    Markiere den Startknoten als besucht

    Solange die Queue nicht leer ist:
```

```
        Entferne den ersten Knoten aus der Queue (aktuelle Ebene)
        Besuche alle benachbarten Knoten dieses Knotens:
            Wenn ein Nachbarknoten nicht besucht wurde:
                Markiere ihn als besucht
                Füge ihn zur Queue hinzu
```

Tiefen Suche / Depth First Search ⬇🔎⬇



Vorgehen

Wird verwendet um einen Graphen oder einen Datenbaum zu durchsuchen. Keine Liste! Mit dieser Suche wird jeder Zweig bis zur Sackgasse komplett durchsucht bevor zum nächsten Zweig gewechselt wird. Im Gegenteil vom kompletten durchsuchen pro Ebene, siehe Breitensuche.

Vorteil

Benötigt weniger Speicher wie die Breitensuche.

Nachteil

Nicht für breite Graphen geeignet. Keine Garantie für kürzeste Pfade.

Merkmale

Benutzt einen Stack oder einen Rekursiven Aufruf auf sich selbst zur Verwaltung von besuchten Knoten. Markiert besuchte Knoten.

Beispielcode

```
// ------------------------------
// ---- Depth-First-Search ----
// https://www.youtube.com/watch?v=7Cox-J7onXw

DepthFirstSearch(Graph, Startknoten):
    Initialisiere einen leeren Stapel (Stack)
    Füge den Startknoten zum Stack hinzu

    Solange der Stack nicht leer ist:
        Entferne den obersten Knoten
        Wenn der Knoten unbesucht ist:
            Markiere ihn als besucht
            Füge alle unbesuchten Nachbarn des Knotens zum Stack hinzu
```