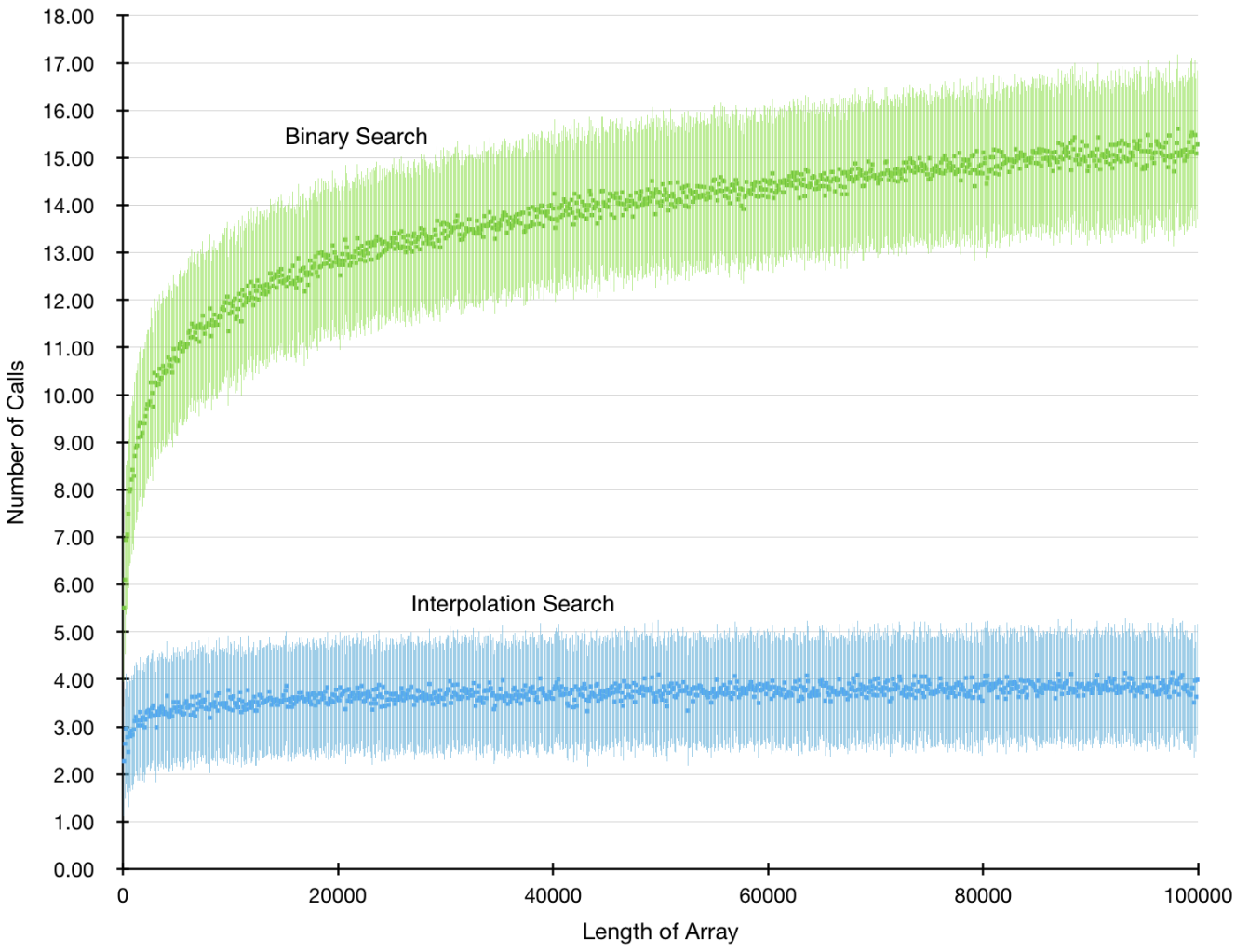
ALP III - Übung 5 - Gruppe 1.8

Tobias Lohse, Marvin Kleinert, Anton Drewing
Tutor: Marcel Erhardt, Mi 10-12

Aufgabe 1

(a)

Im folgenden Diagramm sind die über 100 Iterationen gemittelten Werte für die Anzahl der rekursiven Aufrufe für Array Längen von 100 bis 100,000 mit ihrer Standardabweichung eingetragen. Man kann gut erkennen, dass die Interpolations Suche in O(log log n) liegt und die Binärsuche in O(log n)



(b)

Wenn wir im Array `[1, 1, 1, 1, 1, 1, 1, 1, 2, 99]` nach `2` suchen, so ergibt sich für `l` zwischen `0` und `7` folgendes:

$$m = (key-arr[l])/(arr[r]-arr[l])*(r-l) \leq (2-1)/(99-1)*(9-7) \approx 0.01$$

Somit ergibt sich also immer `m=l` und wir verkleineren in jedem Druchlauf das Array nur um eins. Damit ergeben sich `(n-1) = 9` rekursive Aufrufe, bis wir die `2` gefunden haben. Die Laufzeit liegt also in Θ(n), da die obere und untere Schranke gleich der Funktion `(n-1)` in O(n) ist.

Für solche Datenmengen mit ungleicher Dichte in verschiedenen Wertebereichen, in denen man nach einem Element am Rand des Bereichs mit großer Dichte sucht, ist Insertion Sort daher weniger geeignet.

Aufgabe 2

```
finde(k) -> Node:
  p = head
  while p.unten != null:
    while k >= p.rechts.key:
      p = p.rechts
    p = p.unter
  return p


einfüge(k,d) -> void:
  fügein = [false] * height
  i = 0
  while münzwurf:
    fügein[i] = true
    i++
  p = head
  i = 0
  while p.unter != null:
    while k >= p.rechts.key:
        p = p.rechts
    if fügein[i]:
      p.next = p.next.next
      p.next = new Node(k,d)
      i++
    p = p.unter
```

```
lösche(k) -> void:
  p = head
  i = 0
  while p.unter != null:
    while k >= p.rechts.key:
        p = p.rechts
    if key == p.next.key:
      p.next = p.next.next
    p = p.unter
```

Aufgabe 3

(a)

Um die Balancierung des Baumes zu gewährleisten, wenn es viele Elemente mit gleichem Schlüssel gibt. Ansonten würden wir die Balancierung verlieren und die Operationen könnten nicht mehr in O(log n) ausgeführt werden.

(b)

```
findeAlle(k, node=root, out=[]) -> Node[]:
    if node == nil:
        return out
    else:
        if k == node.key:
            out += node
            if k == node.right.key:
                return findAlle(k, node.right, out)
            if k == node.left.key:
                return findAlle(k, node.left, out)
        elif k < node.key:
            return findAlle(k, node.left, out)
        elif k > node.key:
            return findAlle(k, node.right, out)
```

Die Laufzeit für diesen Algorithmus liegt in O(h+s), da wir zunächst mit einer Suche auf einem normalen Binärbaum in O(h) das erste Element mit Schlüssel k finden. Wir wissen dann, dass alle weiteren Elemente mit Schlüssel k nur aus einer Linie direkter Nachkommen dieses ersten Elements stammen können. Also müssen wir maximal 2s Vergleiche durchführen, um sicher zu stellen, dass wir keine weiteren Elemente mehr finden können.