
AuD - Zusammenfassung

Moritz Gerhardt

1 Inhaltsverzeichnis

1	Inhaltsverzeichnis	1
2	Was ist ein Algorithmus?	2
3	Laufzeitanalyse	3
3.1	O Notation	3
3.1 (a)	Big-O Notation	3
3.1 (b)	Little-o Notation	4
3.1 (c)	Rechenregeln	4
3.2	Ω Notation	5
3.2 (a)	Ω Notation	5
3.2 (b)	ω Notation	6
3.2 (c)	Rechenregeln	6
3.3	Θ Notation	7
4	Sortieren	8
4.1	Sortierproblem	8
4.2	Insertion Sort	9
4.3	Merge Sort	10
4.4	Quicksort	11
4.5	Radix Sort	12
5	Grundlegende Datenstrukturen	13
5.1	Stacks	13
5.2	Queues	14
5.3	Linked List	15
5.4	Binary Search Tree	16
5.5	Red-Black Tree	19

2 Was ist ein Algorithmus?

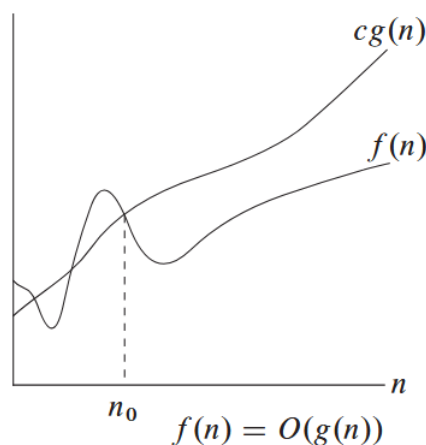
Ein Algorithmus beschreibt eine Handlungsvorschrift zur Umwandlung von Eingaben in eine Ausgabe. Dabei sollte ein Algorithmus im allgemeinen folgende Voraussetzungen erfüllen:

1. Bestimmt:
 - Determiniert: Bei gleicher Eingabe liefert der Algorithmus gleiche Ausgabe.
⇒ Ausgabe nur von Eingabe abhängig, keine äußeren Faktoren.
 - Determinismus: Bei gleicher Eingabe läuft der Algorithmus immer gleich durch die Eingabe.
⇒ Gleiche Schritte, Gleiche Zwischenstände.
2. Berechenbar:
 - Finit: Der Algorithmus ist als endlich definiert. (Theoretisch)
 - Terminierbar: Der Algorithmus stoppt in endlicher Zeit. (Praktisch)
 - Effektiv: Der Algorithmus ist auf Maschine ausführbar.
3. Anwendbar:
 - Allgemein: Der Algorithmus ist für alle Eingaben einer Klasse anwendbar, nicht nur für speziellen Fall.
 - Korrekt: Wenn der Algorithmus ohne Fehler terminiert, ist die Ausgabe korrekt.

3 Laufzeitanalyse

3.1 O Notation

Die O-Notation wird grundsätzlich für *Worst-Case* Laufzeiten verwendet. Sie gibt also eine obere Schranke an, die der Algorithmus im schlechtesten Fall erreicht. Dabei wird oft zwischen Big O-Notation und Little o-Notation unterschieden. Ein Graph zur Repräsentation der O-Notation ist hier zu sehen:



3.1 (a) Big-O Notation

Mathematische Definition:

$$O(g(n)) = \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)$$

Es existieren die positiven Konstanten c und n_0 , sodass für alle $n \geq n_0$ gilt, dass $f(n) \geq 0$ und $f(n) \leq c \cdot g(n)$. Das bedeutet, dass die Funktion $f(n)$ für $n \rightarrow \infty$ den gleichen Wachstumsfaktor hat wie die Funktion $g(n)$. Einfache Berechnung findet wie folgt statt (anhand vom Beispiel $f(n) = 5n^2 + 2n$):

1. Finde den Term mit dem höchsten Wachstumsfaktor ($5n^2$)
2. Konstanten werden weggelassen (n^2)
3. Demnach ist $f(n) = O(n^2)$

Dies kann man dann im Rückschluss so anwenden: Um die Konstanten c und n_0 zu finden, wird die obige Gleichung benutzt:

1. Simplifiziere die Ungleichung $5n^2 + 2n \leq c \cdot n^2$ zu $5 + \frac{2}{n} \leq c$
2. Da $n \geq n_0$ kann man die Gleichung für $n \geq 1$ auflösen um die Konstanten c und n_0 zu finden.
 $\implies 5 + \frac{2}{1} = 7 \leq c \implies c \geq 7$
3. Dementsprechend kann man dann die Konstanten $c = 7$ und $n_0 = 1$ auswählen.

3.1 (b) Little-o Notation

Mathematische Notation:

$$O(g(n)) = \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0 : 0 \leq f(n) < c \cdot g(n)$$

Es existieren die positive Konstanten c und n_0 , sodass für alle $n \geq n_0$ gilt, dass $f(n) \geq 0$ und $f(n) < c \cdot g(n)$. Little-o Notation unterscheidet sich also von Big-O Notation nur oberen Schranke. Während bei Big-O der Wachstumsfaktor beider Funktion gleich sein kann ($f(n) = c \cdot g(n)$), gilt bei Little-o, dass der Wachstumsfaktor der Funktion $f(n)$ kleiner ist als der Wachstumsfaktor der Funktion $g(n)$.

Einfache Berechnung findet analog zu Big-O wie folgt statt (anhand vom Beispiel $f(n) = 5n^2 + 2n$):

1. Finde den Term mit dem höchsten Wachstumsfaktor ($5n^2$)
2. Konstanten werden weggelassen (n^2)
3. Demnach ist $f(n) = o(n^2)$

Hier muss allerdings noch geprüft werden, ob der Wachstumsfaktor der Funktion $f(n)$ kleiner ist als der Wachstumsfaktor der Funktion $g(n)$. Wenn ja, ist die Little-o Notation korrekt für $g(n)$.

Um zu zeigen, dass $f(n) = o(g(n))$:

1. Finde den Limes des simplifizierten Ausdrucks $\frac{f(n)}{g(n)}$, der die Wachstumsrate der Funktion $f(n)$ zur Wachstumsrate der Funktion $g(n)$ vergleicht.
 - $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{5n^2 + 2n}{n^2} = \lim_{n \rightarrow \infty} 5 + \frac{2}{n} = 5$
 $\Leftarrow \frac{2}{n}$ für $n \rightarrow \infty = 0$
2. Da der Limes $\neq 0$ ist, bedeutet das, dass der Wachstum von $f(n)$ nicht geringer ist als der von $g(n)$. Deshalb müssen wir ein Polynomgrad hochgehen, weswegen $f(n) = o(n^3)$ sein muss.
3. Um nun die Konstanten c und n_0 zu finden müssen wir einfach $\frac{f(n)}{g(n)}$ auflösen
 - $\frac{5n^2 + 2n}{n^3} < c$
 - $\frac{5}{n} + \frac{2}{n^2} < c$
 - $\frac{5}{n} < c$, da $\frac{2}{n^2}$ für $n \rightarrow \infty$ schneller abfällt als $\frac{5}{n}$
 - Für $c = 1$ muss dann $n_0 > 5$ sein und kann somit als $n_0 = 6$ gewählt werden.

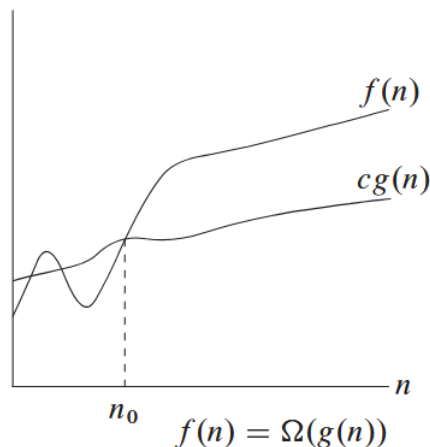
3.1 (c) Rechenregeln

Die Rechenregeln für die O Notation sind:

- Konstanten: $f(n) = a$ mit $a \in \mathbb{R}_{>0} \implies f(n) = O(1)$
Ist die Funktion konstant so ist die Komplexität $O(1)$
- Skalare Multiplikation: $f(n) = O(g) \longrightarrow a \cdot f(n) = O(g)$
Multiplikation der Funktion ändert Komplexität nicht.
- Addition: $f_1 = O(g_1), f_2 = O(g_2) \longrightarrow f_1 + f_2 = O(\max\{g_1, g_2\})$
Die Komplexität der Summe zweier Funktion ist der Maximalwert der Komplexität der beiden Funktionen.
- Multiplikation: $f_1 = O(g_1), f_2 = O(g_2) \longrightarrow f_1 \cdot f_2 = O(g_1 \cdot g_2)$
Die Komplexität des Produkts zweier Funktionen ist das Produkt der Komplexität der beiden Funktionen.

3.2 Ω Notation

Ähnlich zur O Notation, allerdings geht es hier um den *Best-Case* also minimale Anzahl der Schritten, die ein Algorithmus ausführt.



Wird auch wieder in Ω und ω aufgeteilt, die sich nur darin unterscheiden, wie strikt die Grenze ist.

3.2 (a) Ω Notation

Mathematische Definition:

$$\Omega(g(n)) = \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)$$

Es existieren die positiven Konstanten c und n_0 , sodass für alle $n \geq n_0$ gilt, dass $0 \leq c \cdot g(n) \leq f(n)$. Das bedeutet, dass der Wachstumsfaktor von $f(n) \geq c \cdot g(n)$ ist.

Die Berechnung von Ω ist leider nicht immer so simpel wie die Berechnung von O Notation. Nehme zum Beispiel einen Linearen Suchalgorithmus, der eine Liste so lange durchläuft, bis er die gesuchte Zahl gefunden hat. Die Komplexität ist $O(n)$, da, wenn das Element an letzter Stelle steht alle Eingaben durchlaufen werden müssen. Gleichmaßen kann es aber sein, dass das Element an erster Stelle steht, was dann die Komplexität $\Omega(1)$ besitzt. Dies muss allerdings durch Analyse des Algorithmus selbst erkannt werden und kann nicht aus der Funktionsrepräsentation ermittelt werden.

Gilt allerdings nicht sowas, wie vorzeitiger Abbruch bei Suche, so kann Ω ähnlich zu O verwendet werden (Anhand vom Beispiel $f(n) = 5n^2 + 2n$):

1. Finde den Term mit dem höchsten Wachstumsfaktor ($5n^2$)
2. Konstanten werden weggelassen (n^2)
3. Demnach ist $f(n) = \Omega(n^2)$
Da $5n^2 + 2n$ für $n \rightarrow \infty$ mindestens so schnell wächst wie n^2 .

Um Werte für c und n_0 zu finden, kann das Prinzip wie in O Notation verwendet werden, jedoch auf der Definition von Ω angepasst (Umgekehrtes Gleichheitszeichen).

3.2 (b) ω Notation

Mathematische Definition:

$$\omega(g(n)) = \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0 : 0 \leq c \cdot g(n) < f(n)$$

Es existieren die positiven Konstanten c und n_0 , sodass für alle $n \geq n_0$ gilt, dass $0 \leq c \cdot g(n) < f(n)$.

Das bedeutet, dass der Wachstumsfaktor von $f(n) > c \cdot g(n)$ ist.

Für die Bestimmung von ω gilt das selbe wie für Ω , nur das zusätzlich noch folgendes beachtet werden muss:

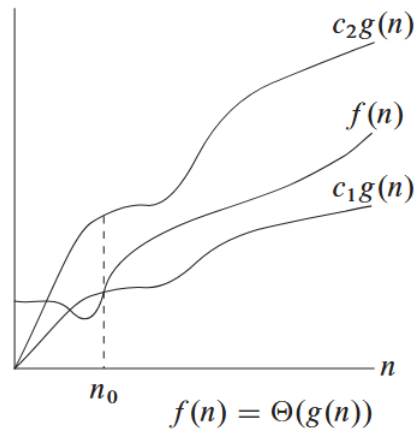
- Hat der Algorithmus einen konstanten Best-Case, so ist ω nicht anwendbar, da $\omega < 1$ sinnlos ist, da per Definition die Komplexität nicht kleiner als 1 sein kann und so der Best-Case schon durch Ω definiert ist.
- Falls nicht konstant, dann muss bei ω ähnlich zu Little-o herausgefunden werden, ob der Wachstumsfaktor von $f(n)$ strikt größer ist als der Wachstumsfaktor der Funktion $g(n)$.

– $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$

– Wenn $\lim = \infty$, so gilt $\omega(g(n))$

– Andernfalls muss der Polynomgrad von $g(n)$ verringert werden:
 $\rightarrow n^x = n^{x-1} \implies x = 1$

3.2 (c) Rechenregeln



4 Sortieren

4.1 Sortierproblem

Sortieralgorithmen sind die wohl am häufigsten verwendeten Algorithmen. Hierbei wird als Eingabe eine Folge von Objekten gegeben, die nach einer bestimmten Eigenschaft sortiert werden. Der Algorithmus soll die Eingabe in der richtigen Reihenfolge (nach einer bestimmten Eigenschaft) zur Ausgabe umwandeln. Es wird hierbei meist von einer total geordneten Menge ausgegangen. (Alle Elemente sind miteinander vergleichbar).

Eine Totale Ordnung wird wie folgt definiert:

Eine Relation \leq auf M ist eine totale Ordnung, wenn:

- Reflexiv: $\forall x \in M : x \leq x$
(x steht in Relation zu x)
- Transitiv: $\forall x, y, z \in M : x \leq y \wedge y \leq z \implies x \leq z$
(Wenn x in Relation zu y steht und y in Relation zu z steht, so folgt, dass x in Relation zu z steht)
- Antisymmetrisch: $\forall x, y \in M : x \leq y \wedge y \leq x \implies x = y$
(Wenn x in Relation zu y steht und y in Relation zu x steht, so folgt, dass $x = y$)
- Totalität: $\forall x, y \in M : x \leq y \vee y \leq x$
(Alle Elemente müssen in einer Relation zueinander stehen)

4.2 Insertion Sort

```
1 class InsertionSort {
2     void insertionSort(int[] arr) {
3         for (int i = 1; i < arr.length; i++) {
4             // 1 to n - 1
5             int key = arr[i];
6             int j = i - 1;
7             while (j >= 0 && arr[j] > key) {
8                 // Loops backwards through the array starting at i - 1
9                 // until it finds an element that is greater than the key or the beginning of the array
10                arr[j + 1] = arr[j];
11                // Shifts the element to the right
12                j--;
13            }
14            arr[j + 1] = key;
15            // Assigns the key to the correct position
16        }
17    }
18 }
```

Prinzip: Die Eingabe wird von links nach rechts durchlaufen. Dafür wird für jedes Element startend bei $i = 1$ der Array von i bis 0 nach links durchlaufen, bis 0 erreicht ist oder key-Wert größer gleich dem i -ten Element ist. Während des durchlaufens nach links werden die Elemente so nach Rechts geschoben, so dass eine Einfügespalte entsteht. Nach dem Ende dieses Durchgangs ist der Spalt bei der position, bei der der Wert eingefügt werden soll. Dies wird dann wiederholt, bis für alle i der Array durchlaufen ist.

4.3 Merge Sort

```
1 class MergeSort {
2     void mergeSort(int[] arr, int left, int right) {
3         if (left < right) {
4             // left < right, otherwise the region has no elements
5             int mid = (left + right) / 2;
6             // Split the region into two halves and do the recursive calls
7             mergeSort(arr, left, mid);
8             mergeSort(arr, mid + 1, right);
9             // Merge the two (now sorted) halves
10            merge(arr, left, mid, right);
11        }
12    }
13
14    private void merge(int[] arr, int left, int mid, int right) {
15        int[] temp = new int[right - left + 1];
16        // Create a temporary array to store the merged elements
17
18        int p = left;
19        int q = mid + 1;
20        for (int i = 0; i < right - left + 1; i++) {
21            // Loops for each element in the region
22            if (q > right || (p <= mid && arr[p] <= arr[q])) {
23                // If p > mid the left half is finished, therefore the element needs to be in right half
24                // Otherwise p needs to be <= mid and the element at p needs to be <= the element at q
25                temp[i] = arr[p];
26                p++;
27                // Adds the element at p to the temporary array and increases p
28            }
29            else {
30                temp[i] = arr[q];
31                q++;
32                // Adds the element at q to the temporary array and increases q
33            }
34        }
35        // Copy the merged elements from the temporary array back to the original array
36        for (int i = 0; i < right - left + 1; i++) {
37            arr[left + i] = temp[i];
38            // left + 0 is the start of the region
39        }
40    }
41 }
```

4.4 Quicksort

```
1 class Quicksort {
2     void quickSort(int[] arr, int left, int right) {
3         if (left < right) {
4             // Region contains more than one element
5             int part = partition(arr, left, right);
6             quickSort(arr, left, part);
7             quickSort(arr, part + 1, right);
8         }
9     }
10
11     private int partition(int[] arr, int left, int right) {
12         int pivot = arr[left];
13
14         int p = left - 1;
15         int q = right + 1;
16         while (p < q) {
17             while (arr[p] < pivot) {
18                 p++;
19             }
20             while (arr[q] > pivot) {
21                 q--;
22             }
23             // Increase / decrease p and q until they are equal to pivot
24             if (p < q) {
25                 int temp = arr[p];
26                 arr[p] = arr[q];
27                 arr[q] = temp;
28                 // Swap arr[p] and arr[q]
29             }
30         }
31         return q;
32     }
33 }
```

4.5 Radix Sort

```
1 import java.util.ArrayList;
2
3 class RadixSort {
4     int D = 10; // possible unique digits
5     int d; // Max amount of digits
6     ArrayList<Integer>[] buckets = new ArrayList[D];
7
8     void radixSort(int[] arr) {
9         d = amountDigits(arr);
10        for (int i = 0; i < d; i++) {
11            // for each digit in the array, 0 least significant
12            for (int j = 0; j < arr.length; j++) {
13                putBucket(arr, i, j);
14            }
15            // Sorts the numbers into their buckets
16            int a = 0;
17            for (int k = 0; k < D; k++) {
18                for (int b = 0; b < buckets[k].size(); b++) {
19                    arr[a] = buckets[k].get(b);
20                    a++;
21                }
22                buckets[k].clear();
23            }
24            // Reads out the buckets in order
25        }
26    }
27
28    private void putBucket(int[] arr, int i, int j) {
29        int z = arr[j] / (int) Math.pow(D, i) % D;
30        // Gets the ith digit of the number
31        int b = buckets[z].size();
32        // size is next free index
33        buckets[z].add(b, arr[j]);
34        // puts the number in the bucket z at position b
35        // Depending on implementation might need to increase size manually
36    }
37
38    private int amountDigits(int[] arr) {
39        int max = arr[0];
40        for (int i = 1; i < arr.length; i++) {
41            if (arr[i] > max) {
42                max = arr[i];
43            }
44        }
45        // Get the biggest number
46        return (int) Math.log10(max) + 1;
47        // Get the amount of digits of the number
48    }
49 }
```

5 Grundlegende Datenstrukturen

5.1 Stacks

Stacks operieren unter dem "First in - Last out" (FILO) Prinzip. Ähnlich zu einem Kartendeck, wo die unterste (Erste Karte) die ist, die als letztes gezogen wird.

Stacks werden normalerweise mit den folgenden Funktionen erstellt:

- **newn**: Erstellt einen neuen Stack.
- **isEmpty**: gibt an ob der Stack leer ist.
- **pop**: gibt das oberste Element des Stacks zurück und entfernt es vom Stack.
- **push(k)**: Fügt **k** auf den Stack hinzu

Eine mögliche Implementation auf Grundlage eines Arrays wäre:

```
1 class Stack {
2     private int[] arr;
3     private int top;
4
5     Stack(int size) {
6         arr = new int[size];
7         top = -1;
8         // Creates a new array with size
9     }
10
11     boolean isEmpty() {
12         return top < 0;
13         // Returns true if empty
14     }
15
16     int pop() {
17         return arr[top--];
18         // Removes and returns the top element
19     }
20
21     void push(int k) {
22         arr[++top] = k;
23         // Adds an element
24     }
25 }
```

Push und Pop schmeißen Fehlermeldung wenn Stack leer bzw. voll ist. Oft als Stack underflow und Stack overflow benannt. Hier wäre es automatisch IndexOutOfBoundsException.

Oft werden Stacks auch mit variabler Größe implementiert. Dies kann über verschiedene Wege passieren, zum Beispiel Kopieren des Arrays in einen größeren Array oder Implementation über mehrere Arrays (z.B. über Linked List). Häufig wird das erstere so implementiert, dass der Array in einen Array mit doppelter Größe kopiert wird.

5.2 Queues

Queues werden normalerweise mit den folgenden Funktionen erstellt:

- **newn**: Erstellt einen neuen Queue.
- **isEmpty**: gibt an ob der Queue leer ist.
- **enqueue(k)**: Fügt k auf den Queue hinzu
- **dequeue**: gibt das erste Element des Queues zurück und entfernt es vom Queue.

Hier ist die Implementation für Queues wie folgt:

```
1 class Queue {
2
3     private int[] arr;
4     private int front;
5     private int back;
6
7     Queue(int size) {
8         arr = new int[size];
9         front = -1;
10        back = -1;
11    }
12
13    boolean isEmpty() {
14        return back == -1;
15    }
16
17    boolean isFull() {
18        return (front + 1) % arr.length == back;
19        // If front + 1 is equal to back, the queue is full
20        // Modulo makes this usable for cyclic arrays
21    }
22
23    void enqueue(int k) {
24        if (isFull()) {
25            throw new RuntimeException("Queue is full");
26        }
27        else {
28            if (isEmpty()) {
29                front = 0;
30            }
31            back = (back + 1) % arr.length;
32            // Modulo so that cyclic arrays work
33            arr[back] = k;
34        }
35    }
36
37    int dequeue() {
38        if (isEmpty()) {
39            throw new RuntimeException("Queue is empty");
40        }
41        else {
42            int temp = arr[front];
43            front = (front + 1) % arr.length;
44            // Modulo so that cyclic arrays work
45            if (front == back) {
46                front = -1;
47                back = -1;
48            }
49            // If front and back are equal, the queue is empty -> reset
50            return temp;
51        }
52    }
53 }
```

5.3 Linked List

Eine einfache Linked List besteht aus mehreren Elementen, die jeweils immer einen Wert und eine Referenz auf das nächste Element in der Liste haben. Eine einfache Linked List kann wie folgt implementiert werden:

```
1 class LinkedElement {
2     Integer key = null;
3     LinkedElement next = null;
4
5     public LinkedElement(Integer key) {
6         this.key = key;
7     }
8 }
9
10
11
12
13
14
15 class LinkedList {
16     LinkedElement head = null;
17
18     void insert(int k) {
19         LinkedElement elem = new LinkedElement(k);
20         if (head == null) {
21             head = elem;
22         }
23         else {
24             head.next = elem;
25             head = elem;
26         }
27     }
28
29     void delete(int k) {
30         LinkedElement prev = null;
31         LinkedElement curr = head;
32         while (curr != null && curr.key != k) {
33             prev = curr;
34             curr = curr.next;
35         }
36         if (curr == null) {
37             throw new UException("Element not found");
38         }
39         if (prev != null) {
40             prev.next = curr.next;
41         }
42         else {
43             head = curr.next;
44         }
45     }
46
47     LinkedElement search(int k) {
48         LinkedElement curr = head;
49         while (curr != null && curr.key != k) {
50             curr = curr.next;
51         }
52         if (curr == null) {
53             throw new UException("Element not found");
54         }
55         return curr;
56     }
57 }
```

5.4 Binary Search Tree

```
1 class BSTNode {
2     Integer key;
3     BSTNode left;
4     BSTNode right;
5     BSTNode parent;
6
7     BSTNode(Integer k) {
8         key = k;
9     }
10 }

```

```
1 class BSTree {
2
3     BSTNode root = null;
4
5     void insert(BSTNode z) {
6         BSTNode x = root; // Traversal starting from the root
7         BSTNode px = null; // Parent of x, initially null
8
9         while(x != null) {
10             px = x;
11             if (z.key < x.key) {
12                 x = x.left;
13             } else {
14                 x = x.right;
15             }
16         } // Traversing the tree until finding the insertion point
17
18         z.parent = px; // Sets the parent of the node to be inserted
19         if (px == null) { // px only null if the tree is empty -> loop never runs -> z is root
20             root = z;
21         } else if (z.key < px.key) { // Key smaller -> left child
22             px.left = z;
23         } else { // Key bigger -> right child
24             px.right = z;
25         }
26         // May add the same node twice as it doesn't check for duplicates
27     }
28 }
```



```

1  void delete(BSTNode z) {
2      if (z.left == null) { // If z has no left child, transplants the right child to z's position
3          transplant(z, z.right);
4      } else if (z.right == null) { // If z has no right child, transplants the left child to z's
5          position
6          transplant(z, z.left);
7      } else { // If z has both left and right children
8          BSTNode y = z.right;
9          while (y.left != null) {
10             y = y.left;
11         } // Finds the next biggest element of z = smallest in right subtree of z
12         if (y.parent != z) { // If the next biggest element y is not child of z
13             transplant(y, y.right); // Transplants the right child of y to y's position
14             y.right = z.right; // The right child of y becomes the right child of z
15             y.right.parent = y; // The parent of the right child of y becomes y
16         }
17         transplant(z, y); // Transplants y to z's position
18         y.left = z.left; // The left child of y becomes the left child of z
19         y.left.parent = y; // The parent of the left child of y becomes y
20     }
}

```

```

1 void transplant(BSTNode u, BSTNode v) {
2     // Transplants v to the parent of u
3     if (u.parent == null) { // If u is the root, v becomes the new root
4         root = v;
5     } else if (u == u.parent.left) { // If u is a left child, v becomes a left child
6         u.parent.left = v;
7     } else { // If u is a right child, v becomes a right child
8         u.parent.right = v;
9     }
10    if (v != null) { // If v is not null, v becomes a child of u's parent
11        v.parent = u.parent;
12    }
13 }

```

```

1 BSTNode iterativeSearch(int k) {
2     BSTNode curr = root;
3     while (curr != null && curr.key != k) {
4         if (k < curr.key) {
5             curr = curr.left;
6         } else {
7             curr = curr.right;
8         }
9     }
10    return curr;
11    // Returns null if element not found
12 }
13
14 BSTNode recursiveSearch(int k, BSTNode curr) {
15     if (curr == null) {
16         return null;
17     }
18     if (k < curr.key) {
19         return recursiveSearch(k, curr.left);
20     } else if (k > curr.key) {
21         return recursiveSearch(k, curr.right);
22     }
23
24     return curr;
25     // Returns null if element not found
26 }

```

```

1 void traversal(BSTNode curr) {
2     if (curr != null) {
3         return;
4     }
5     // Any actions that should be done in a specific order can be done
6     // Here for preorder traversal
7     traversal(curr.left);
8     // Here for inorder traversal
9     traversal(curr.right);
10    // Here for postorder traversal
11    // Left and right can also be exchanged to traverse in reverse order
12 }
13 }

```

5.5 Red-Black Tree

```
1 class RBNode {
2     Integer key;
3     RBNode left;
4     RBNode right;
5     RBNode parent;
6     Color color;
7
8     RBNode(Integer k) {
9         key = k;
10    }
11 }

1 class RBTree {
2     RBNode sent;
3     RBNode root = null;
4
5     RBTree() {
6         sent = new RBNode(null);
7         sent.color = Color.BLACK;
8         sent.left = sent;
9         sent.right = sent;
10        // Sentinel always points to itself ->
11        // node.parent.parent and its children will never result in null references
12    }
13
14    // Traversal and search are the same as BSTree
15
16    void insert(RBNode z) {
17        // Very similar to BSTree, with addition of color and parent of sentinel instead of null
18        RBNode x = root; // Traversal starting from the root
19        RBNode px = sent; // Parent of x, initially sentinel unlike BST
20
21        while (x != null) {
22            px = x;
23            if (z.key < x.key) {
24                x = x.left;
25            } else {
26                x = x.right;
27            }
28        } // Traversing the tree until finding the insertion point
29
30        z.parent = px; // Sets the parent of the node to be inserted
31        if (px == sent) { // px only sentinel if the tree is empty -> loop never runs -> z is root
32            root = z;
33        } else if (z.key < px.key) { // Key smaller -> left child
34            px.left = z;
35        } else { // Key bigger -> right child
36            px.right = z;
37        }
38
39        z.color = Color.RED; // Sets color of new Node to red, will not necessarily stay red
40        fixColorsAfterInsertion(z); // Fixes colors in tree after insertion to maintain RB properties
41        // May add the same node twice as it doesn't check for duplicates
42    }

43    void fixColorsAfterInsertion(RBNode z) {
44        while (z.parent.color == Color.RED) { // While z's parent is red
45            if (z.parent == z.parent.parent.left) { // If z's parent is a left child
46                RBNode y = z.parent.parent.right; // Gets sibling of z's parent
47                if (y != null && y.color == Color.RED) { // If sibling exists and is red
48                    z.parent.color = Color.BLACK; // Set z's parent to black
49                    y.color = Color.BLACK; // Set z's sibling to black
50                    z.parent.parent.color = Color.RED; // Set z's grandparent to red
51                    z = z.parent.parent; // Set z to z's grandparent
52                } else { // If z doesn't have a sibling or sibling is black
53                    if (z == z.parent.right) { // If z is a right child
54                        z = z.parent; // Set z to z's parent
55                        rotateLeft(z); // Rotate new z to left
56                    }
57                }
58            }
59        }
60    }
61 }
```

```

57         z.parent.color = Color.BLACK; // Set z's parent to black
58         z.parent.parent.color = Color.RED; // Set z's grandparent to red
59         rotateRight(z.parent.parent); // Rotate z's grandparent to right
60     }
61     } else {
62         // Same as above but with right and left exchanged
63         RBNode y = z.parent.parent.left;
64         if (y != null && y.color == Color.RED) {
65             z.parent.color = Color.BLACK;
66             y.color = Color.BLACK;
67             z.parent.parent.color = Color.RED;
68             z = z.parent.parent;
69         } else {
70             if (z == z.parent.left) {
71                 z = z.parent;
72                 rotateRight(z);
73             }
74             z.parent.color = Color.BLACK;
75             z.parent.parent.color = Color.RED;
76             rotateLeft(z.parent.parent);
77         }
78     }
79 }
80 root.color = Color.BLACK; // Set root to black, as it always should be
81 // Never needs to check nodes below z as their properties will not change after insertion
82 }

```

```

83 void rotateLeft(RBNode x) {
84     RBNode y = x.right;
85
86     x.right = y.left; // Set x's right child to y's left child
87     if (y.left != null) { // If y has a left child
88         y.left.parent = x; // Set y's left child's parent to x
89     }
90     y.parent = x.parent; // Set y's parent to x's parent
91     if (x.parent == sent) { // If x is the root, set y to be the root
92         root = y;
93     } else if (x == x.parent.left) { // If x is a left child, set x's parent's left child to y
94         x.parent.left = y;
95     } else { // If x is a right child, set x's parent's right child to y
96         x.parent.right = y;
97     }
98     y.left = x; // Set y's left child to x
99     x.parent = y; // Set x's parent to y
100 }
101 void rotateRight(RBNode x) {
102     // Same as rotateLeft but with right and left exchanged
103     RBNode y = x.left;
104
105     x.left = y.right;
106     if (y.right != null) {
107         y.right.parent = x;
108     }
109     y.parent = x.parent;
110     if (x.parent == sent) {
111         root = y;
112     } else if (x == x.parent.right) {
113         x.parent.right = y;
114     } else {
115         x.parent.left = y;
116     }
117     y.right = x;
118     x.parent = y;
119 }

```

```

121 void delete(RBNode z) {
122     RBNode a = z.parent; // a represent node with black depth imbalance
123     int dbh = 0; // delta black height, -1 for right, 1 for left leaning
124
125     if (z.left == null && z.right == null) { // If z is a leaf
126         if (z.color == Color.BLACK && z != root) { // If z is black

```

```

127         if (z == z.parent.left) { // If z is a left child
128             dbh = -1; // Set delta black height to -1
129         } else { // If z is a right child
130             dbh = 1; // Set delta black height to 1
131         }
132     }
133     transplant(z, null); // Transplant z to null
134 } else if (z.left == null) { // If z only has a right child
135     RBNode y = z.right;
136     transplant(z, z.right);
137     y.color = z.color;
138 } else if (z.right == null) { // If z only has a left child
139     RBNode y = z.left;
140     transplant(z, z.left);
141     y.color = z.color;
142 } else { // If z has two children
143     RBNode y = z.right;
144     a = y;
145     boolean wentLeft = false;
146     while (y.left != null) { //
147         a = y;
148         y = y.left;
149         wentLeft = true;
150     }
151     if (y.parent != z) { // Loop didn't run
152         transplant(y, y.right);
153         y.right = z.right;
154         y.right.parent = y;
155     }
156     transplant(z, y);
157     y.left = z.left;
158     y.left.parent = y;
159     if (y.color == Color.BLACK) {
160         if (wentLeft) { // Tree imbalanced depending on y location
161             dbh = -1;
162         } else {
163             dbh = 1;
164         }
165     }
166     y.color = z.color;
167 }
168 if (dbh != 0) { // If black height imbalance
169     fixColorsAfterDeletion(a, dbh);
170 }
171 }

```

```

173 void fixColorsAfterDeletion(RBNode a, int dbh) {
174     if (dbh == -1) { // Extra black node on the right
175         RBNode x = a.left;
176         RBNode b = a.right;
177         RBNode c = b.left;
178         RBNode d = b.right;
179         if (x != null && x.color == Color.RED) {
180             // Easy case: x is red
181             x.color = Color.BLACK;
182         } else if (a.color == Color.BLACK
183             && b.color == Color.RED) {
184             // Case 1: a black, b red
185             rotateLeft(a);
186             a.color = Color.RED;
187             b.color = Color.BLACK;
188             fixColorsAfterDeletion(a, dbh);
189         } else if (a.color == Color.RED
190             && b.color == Color.BLACK
191             && (c == null || c.color == Color.BLACK)
192             && (d == null || d.color == Color.BLACK)) {
193             // Case 2a: a red, b black, c and d black
194             a.color = Color.BLACK;
195             b.color = Color.RED;
196         } else if (a.color == Color.BLACK
197             && b.color == Color.BLACK

```

```

198         && (c == null || c.color == Color.BLACK)
199         && (d == null || d.color == Color.BLACK)) {
200     // Case 2b: a black, b black, c and d black
201     b.color = Color.RED;
202     if (a == a.parent.left) {
203         dbh = 1;
204     } else if (a == a.parent.right) {
205         dbh = -1;
206     } else {
207         dbh = 0;
208     }
209     fixColorsAfterDeletion(a.parent, dbh);
210 } else if (b.color == Color.BLACK
211     && c != null && c.color == Color.RED
212     && (d == null || d.color == Color.BLACK)) {
213     // Case 3: a either, b black, c red, d black
214     rotateRight(b);
215     c.color = Color.BLACK;
216     fixColorsAfterDeletion(a, dbh);
217 } else if (b.color == Color.BLACK
218     && d != null && d.color == Color.RED) {
219     // Case 4: a either, b black, c either, d red
220     rotateLeft(a);
221     b.color = a.color;
222     a.color = Color.BLACK;
223     d.color = Color.BLACK;
224 }
225 } else { // Extra black node on the left
226     // Same as above but with right and left exchanged
227     RBNode x = a.right;
228     RBNode b = a.left;
229     RBNode c = b.right;
230     RBNode d = b.left;
231     if (x != null && x.color == Color.RED) {
232         // Easy case: x is red
233         x.color = Color.BLACK;
234     } else if (a.color == Color.BLACK
235         && b.color == Color.RED) {
236         // Case 1: a black, b red
237         rotateRight(a);
238         a.color = Color.RED;
239         b.color = Color.BLACK;
240         fixColorsAfterDeletion(a, dbh);
241     } else if (a.color == Color.RED
242         && b.color == Color.BLACK
243         && (c == null || c.color == Color.BLACK)
244         && (d == null || d.color == Color.BLACK)) {
245         // Case 2a: a red, b black, c and d black
246         a.color = Color.BLACK;
247         b.color = Color.RED;
248     } else if (a.color == Color.BLACK
249         && b.color == Color.BLACK
250         && (c == null || c.color == Color.BLACK)
251         && (d == null || d.color == Color.BLACK)) {
252         // Case 2b: a black, b black, c and d black
253         b.color = Color.RED;
254         if (a == a.parent.right) {
255             dbh = 1;
256         } else if (a == a.parent.left) {
257             dbh = -1;
258         } else {
259             dbh = 0;
260         }
261         fixColorsAfterDeletion(a.parent, dbh);
262     } else if (b.color == Color.BLACK
263         && c != null && c.color == Color.RED
264         && (d == null || d.color == Color.BLACK)) {
265         // Case 3: a either, b black, c red, d black
266         rotateLeft(b);
267         c.color = Color.BLACK;
268         fixColorsAfterDeletion(a, dbh);

```

```

269         } else if (b.color == Color.BLACK
270             && d!= null && d.color == Color.RED) {
271             // Case 4: a either, b black, c either, d red
272             rotateRight(a);
273             b.color = a.color;
274             a.color = Color.BLACK;
275             d.color = Color.BLACK;
276         }
277     }
278 }

```

```

280 void transplant(RBNode u, RBNode v) {
281     // Transplants v to the parent of u
282     if (u.parent == sent) { // If u is the root, v becomes the new root
283         root = v;
284     } else if (u == u.parent.left) { // If u is a left child, v becomes a left child
285         u.parent.left = v;
286     } else { // If u is a right child, v becomes a right child
287         u.parent.right = v;
288     }
289     if (v != null) { // If v is not null, v becomes a child of u's parent
290         v.parent = u.parent;
291     }
292 }
293 }

```