Sammlung von Musteraufgaben

IntroProg-TutorInnen

WS 2021/22

Vorarnung

Diese Aufgaben wurden von IntroProg-TutorInnen erstellt. Während sie schon eine hohe Ähnlichkeit zu typischen Klausur-Aufgaben anzeigen und daher zum Üben hilfreich sein sollten, guarantiert das IntroProg-Team keinerweise, dass die Aufgaben in den eigentlichen Klausuren tatsächlich zu einem bestimmten Grad den Aufgaben in dieser Sammlung von Musteraufgaben ähneln werden.

Aufgabensammlung

1 Sortieralgorithmen

1.1 Laufzeitverhalten

Geben Sie für alle der unten aufgeführten Sortieralgorithmen die folgenden Eigenschaften an:

a. Worst-Case Laufzeit ($\mathcal{O}(\mathrm{f}(n))$)

b. Beispiel, wann dieser Algorithmus zu nutzen ist (mit Begründung)

c. Beispiel für eine Situation mit Worst-Case-Laufzeitverhalten

1.1.1 Selectionsort

1.1.2 Insertionsort

Aufgabensammlung

1.1.3 Mergesort

1.1.4 Heapsort

Aufgabensammlung

1.1.5 Quicksort

Aufgabensammlung

2 Natural Mergesort

Natural Mergesort (natürliches Mergesort) ist eine Erweiterung von Mergesort, die bereits vor sortierte Teilfolgen, so genannte *Run*, innerhalb der zu sortierenden Startliste ausnutzt. Die Basis für den Mergevorgang bilden hier nicht die rekursiv oder iterativ gewonnenen `Zweiergruppen`, sondern die in einem ersten Durchgang zu bestimmenden `runs`:

Startliste	[3, 4, 2, 1, 7, 5, 8, 9, 0, 6]
`Runs` bilden	[3, 4], [2], [1, 7], [5, 8, 9], [0, 6]
Merge	[2, 3, 4], [1, 5, 7, 8, 9], [0, 6]
Merge	[1, 2, 3, 4, 5, 7, 8, 9], [0, 6]
Merge	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Aufgabensammlung

2.1 Laufzeit

2.1.1 Wie ist die zu erwartende Worst-Case-Laufzeit, wenn die Merge-Funktion wie in der Vorlesung definiert ist ($\mathcal{O}$-Notation)? Begründen Sie ihre Antwort.

2.1.2 Für welche Fälle ist der gegebene Algorithmus gut/schlecht geeignet?

Aufgabensammlung

2.2 Vergleich mit Mergesort

2.2.1 Gibt es Fälle, in denen Natural Mergesort nicht-schneller als Mergesort läuft? Begründen Sie Ihre Aussage (kurz).

3 Korrektheit

Gegeben sei folgender Algorithmus:

Algorithm 1 Summe

```
1: procedure SUM(A)
2:     res ← A
3:     i ← A-1
4:     while i > 0 do
5:         res ← res + i
6:         i ← i − 1
7:     return res
```

welcher die Summe von 0 bis A berechnet. Zeigen Sie seine Korrektheit. Gehen Sie dabei wie folgt vor:

1. Zeigen Sie, dass in Zeile 4 die Invariante $res = \sum_{k=i+1}^{A} k$ gilt:
   (a) Zeigen Sie, dass die Invariante zum Induktionsanfang gilt.
   (b) Zeigen Sie, dass aus einer gültigen Invariante bei $i = i_0$ auch eine gültige Invariante bei $i = i_0 - 1$ für $i_0 \in \{1..A-1\}$ folgt
2. Begründen Sie, weshalb aus der Invariante die Korrektheit mit $SUM(A) = \sum_{k=1}^{A} k$ folgt.

Aufgabensammlung

Aufgabensammlung

4 Datenstrukturen

4.1 Laufzeit

Geben Sie für alle der unten aufgeführten Sortieralgorithmen die folgenden Worst-Case-Laufzeiten an.

4.1.1 Einfach-verkettete Liste

Einfügen am Anfang

Einfügen am Ende

Suche

Nachfolgersuche

Vorgängersuche

Aufgabensammlung

4.1.2 Doppelt-verkettete Liste

Einfügen am Anfang

Einfügen am Ende

Suche

Nachfolgersuche

Vorgängersuche

Aufgabensammlung

4.1.3 Min-Heap

Einfügen

Suche

Minimum-Suche

Aufgabensammlung

4.1.4 AVL-Baum

Einfügen

Suche

Minimum-Suche

Aufgabensammlung

5 Heap

5.1 Gegeben ist die Liste [3,4,2,1,7,5,8,9,0,6].

5.1.1 Bauen Sie schrittweise einen Max-Heap auf.

Geben Sie alle Zwischenschritte als Baum-Repräsentation an. Geben Sie zusätzlich das Endergebnis in Array-Repräsentation an. Nutzen Sie die in der Vorlesung vorgestellte Methode mit Heapify.

Aufgabensammlung

5.1.2 Geben Sie in Worten wieder, wie Heapsort mit einem Max-Heap sortiert.

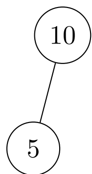
6 AVL-Baum

6.1 Was unterscheidet einen AVL-Baum von einem normalen Binärbaum?

6.2 Was versteht man unter einem Beinahe-AVL-Baum?

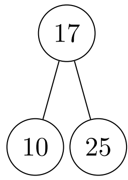
6.3 Führen Sie eine Handsimulation der Einfügeoperation auf die gegebenen AVL-Bäume aus.

6.3.1 Einzufügen: 3



Aufgabensammlung

6.3.2 Einzufügen: 2, 15, 12



Aufgabensammlung

7 Splay-Tree

Ein Splay-Tree ist ein binärer Suchbaum, der sich dadurch auszeichnet, dass der letzte genutzte Knoten stets die Wurzel ist. Damit wird erreicht, dass ein häufig hintereinander angefragtes Element schneller zur Verfügung steht. Unterknoten des angefragten Knotens werden wie bei AVL-Bäumen rotiert (genannt Zick- und Zack-Operation). Damit ergibt sich folgendes Verhalten:

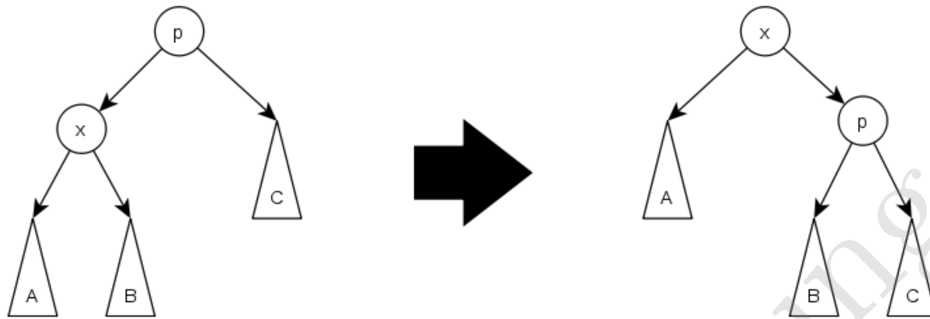


Figure 1: Anfrage an Knoten x (Zick-Operation)

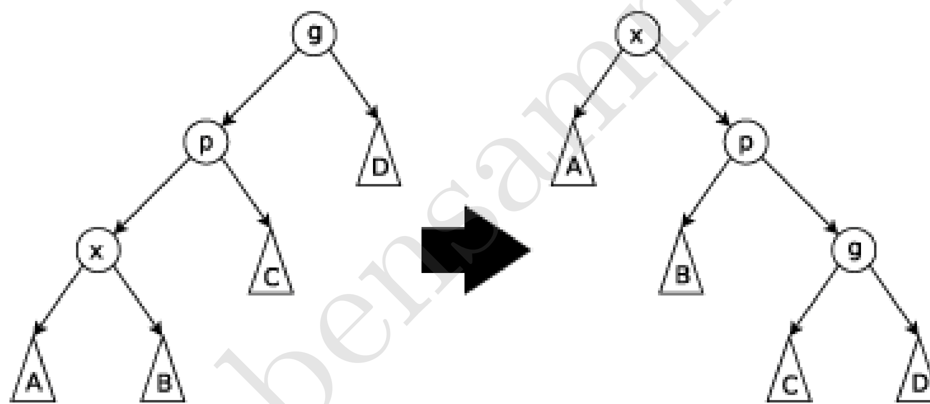


Figure 2: Anfrage an Knoten x (Zick-Zick-Operation)

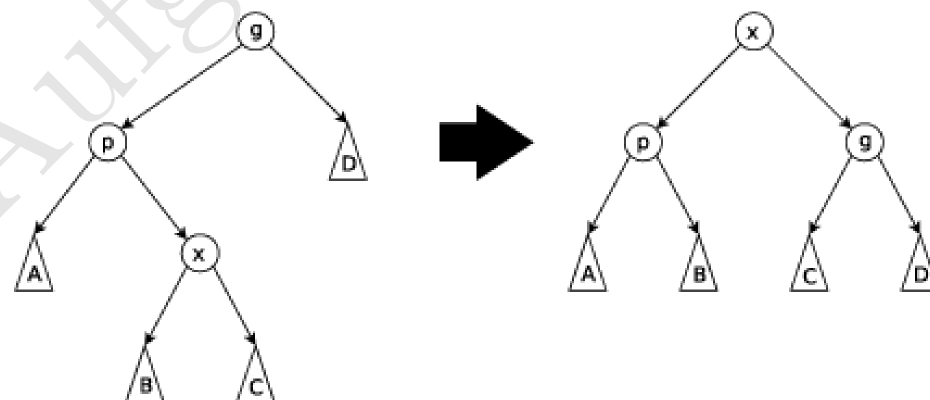


Figure 3: Anfrage an Knoten x (Zack-Zick-Operation)

Aufgabensammlung

7.1 Gegeben ist der folgende Code. Vervollständigen Sie die Funktion `Zack`.

```c
struct node
{
    int key;
    struct node *left, *right;
};

struct node* newNode(int key)
{
    struct node* node = (struct node*) malloc(sizeof(struct node));
    node->key   = key;
    node->left  = node->right  = NULL;
    return (node);
}

struct node *zick(struct node *x)
{
    struct node *y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}

struct node *zack(struct node *x)
{









}
```

Aufgabensammlung

7.2 Gegeben ist die `splay`-Funktion, welche ein gesuchtes Element als Wurzel setzt. Finden Sie 3 Logik und 3 Syntaxfehler. Gehen Sie davon aus, dass die Zeilen 36 und 61 korrekt sind.

```c
// Diese Funktion setzt key als Wurzel, falls key im Baum vorhanden ist.
// Falls nicht, setzt sie den letzten genutzten Knoten als Wurzel.
// Diese Funktion modifiziert den Baum und gibt die neue Wurzel zurueck.
struct node *splay(struct node *root, int key)
{
    // Base cases: Wurzel ist Null oder das gesuchte Element
    if (root = NULL || root->key == key)
        return root;

    // key liegt im linken Subbaum
    if (root->key > key)
    {
        // Key ist nicht im Baum
        if (root->left == NULL) return root;

        // Zick-Zick
        if (root->left->key > key)
```

```
18         {
19             // Rekursiv den key als Wurzel von links-links setzten
20             root->left->left = splay(root->left->left, key);
21 
22             // Erst Wurzel-Rotation, zweite Rotation folgt nach else
23             root = zick(root);
24         }
25         else if (root->left->key < key) // Zick-Zack
26         {
27             // Rekursiv den key als Wurzel von links-rechts setzen
28             root->left->right = splay(root->left->right, key)
29 
30             // Hier als erstes die Rotation fuer root->left machen
31             if (root->left->right != NULL)
32                 root->left = zack(root->left);
33         }
34 
35         // zweite Wurzel-Rotation
36         return (root->left == NULL)? root: zick(root);
37     }
38     else // key liegt rechts
39     {
40         // key ist nicht im Baum
41         if (root->right == NULL) return NULL;
42 
43         // Zack-Zick
44         if (root->right->key > key)
45         {
46             // key als Wurzel von rechts-links setzen
47             root->right->right = splay(root->right->left, key);
48 
49             // erst die Rotation fuer root->right machen
50             if (root->right->left != NULL)
51                 root->right = zick(root->right);
52         }
53         else if root->right->key < key// Zack-Zack
54         {
55             // key als Wurzel von rechts-rechts setzen und erste Rotation
56             root->right->right = splay(root->right->right, key);
57             root = zack(root);
58 
59 
60         // Zweite Rotation der Wurzel
61         return (root->right == NULL)? root: zack(root);
62     }
63 }
```

Aufgabensammlung

#Zeile	Fehlertyp	Verbesserung

Table 1: Fehlerkorrektur

7.3 Gegeben ist die Definition der `search`-Funktion, welche das gesuchte Element zurückgeben soll. Falls das Element nicht gefunden wird, soll `NULL` zurückgegeben werden. Implementieren Sie die Funktion.

Hinweis: Sie dürfen davon ausgehen, dass `splay` korrekt funktioniert.

2. Hinweis: Der Baum darf sich während der Suche verändern.

```
1  struct node *search(struct node *root, int key)
2  {
3
4
5
6
7
8
9
10
11
12
13
14
15 }
```

Aufgabensammlung

8 Komplexität

8.1 Geben Sie in eigenen Worten die Bedeutung der Landau-Symbole wieder.

Landau-Symbol	Bedeutung
$\mathcal{O}(n)$	
$o(n)$	
$\Omega(n)$	
$\omega(n)$	
$\Theta(n)$	

Table 2: Landau-Symbole

8.2 Was unterscheidet die echte Laufzeit ($T(n)$) von der in Aufgabe 8.1 behandelten Symbolik?

8.3 Insertion-Sort

Gegeben sei folgender Pseudocode für Insertion-Sort.

Algorithm 2 InsertSort

```
1: procedure InsertSort(A)
2:     for j ← 2 to length(A) do
3:         key ← A[j]
4:         i ← j − 1
5:         while i > 0 and A[i] > key do
6:             A[i + 1] ← [i]
7:             i ← i − 1
8:         A[i + 1] ← key
```

Aufgabensammlung

8.3.1 Geben Sie für jede Zeile an, wie oft diese im schlechtesten Fall aufgerufen wird. Nennen Sie die genutzte Zeitvariable n.

8.3.2 Welche genaue Worst-Case-Laufzeit ergibt sich damit?

$T(n) =$

Aufgabensammlung

8.3.3 Zu welcher Komplexitätsklasse gehört Insertion-Sort? Geben Sie die niedrigste $\mathcal{O}(n)$-Grenze an. Begründen Sie Ihre Antwort.

8.4 Ist ein Algorithmus mit $T(n) = 1000n^2$ Teil der $\mathcal{O}(n)$-Algorithmen? Begründen Sie Ihre Antwort.

8.5 Ist ein Algorithmus mit $T(n) = 1000n^2$ Teil der $\mathcal{O}(n^2)$-Algorithmen? Begründen Sie Ihre Antwort.

8.6 Ist ein Algorithmus mit $T(n) = 1000n^2$ Teil der $\mathcal{O}(n^3)$-Algorithmen? Begründen Sie Ihre Antwort.

Aufgabensammlung