

Domača naloga 2

Naloga 2.1

a)

0	A		13	M
1	B		14	N
2	C		15	O
3	Č		16	P
4	D		17	R
5	E		18	S
6	F		19	Š
7	G		20	T
8	H		21	U
9	I		22	V
10	J		23	Z
11	K		24	Ž
12	L			

Elementi:

$$(13 \cdot 16) \bmod 5 = 3 \text{ P}$$

$$(13 \cdot 15) \bmod 5 = 0 \text{ O}$$

$$(13 \cdot 4) \bmod 5 = 2 \text{ D}$$

$$(13 \cdot 0) \bmod 5 = 0 \text{ A}$$

$$(13 \cdot 20) \bmod 5 = 0 \text{ T}$$

$$(13 \cdot 11) \bmod 5 = 3 \text{ K}$$

$$(13 \cdot 9) \bmod 5 = 2 \text{ I}$$

Tabela izgleda takole:

$$0 : O \rightarrow A \rightarrow T$$

$$1 :$$

$$2 : D \rightarrow I$$

$$3 : P \rightarrow K$$

2

4:

b)

$k = 1$

$m = 13$

Inicializiramo m =število elementov in pogledamo vse smiselne vrednosti k , to je od 1 do $m-1$. Če se pri nobeni vrednosti k ne znebimo sovpadanj, povečamo m za ena in ponovimo postopek. Ko ne dobimo več sovpadanj, smo našli ustrezni, minimalni k in m . Skripta, ki poišče najmanjši k in m :

```
1 #!/usr/bin/python
2
3 # P O D A T K I
4 elements = [16, 15, 4, 0, 20, 11, 9]
5 print(findMinKandM(elements))
6
7 def findMinKandM(elements):
8     m=len(elements)
9     while (True):
10         for k in range(1,m):
11             T = [0]*m # array of zeros
12             for elt in elements:
13                 T[ (k*elt) % m ] += 1
14
15             i=0
16             while (i<m):
17                 if (T[i] > 1):
18                     # we have duplicate, find another k and m
19                     break
20                 i+=1
21
22             if (i==m):
23                 # we didn't have any duplicates return k and m
24                 return (k,m)
25
26         m+=1
```

c)

$5 \Rightarrow T1[0]$

$12 \Rightarrow T1[2]$

$28 \Rightarrow T1[3]$

15 => T1[0], 5 => T2[1]

19 => T1[4]

47 => T1[2], 12 => T2[2]

2 => T1[2], 47 => T2[4]

9 => T1[4], 19 => T2[3]

18 => T1[3], 28 => T2[0]

16 => T1[1]

Končni rezultat:

T1	T2
15	28
16	5
2	12
18	19
9	47

Naloga 2.2

a)

glava (h=3)

4 (h=2)

5 (h=1)

11 (h=1)

15 (h=1)

18 (h=3)

stražar ∞ (h=3)

b)

<p>Data: ključ x</p> <p>Result: true, če je element vstavljen; false v primeru, da zmanjka pomnilnika (izpuščeno v kodi spodaj)</p> <pre> 1 Node u ← head 2 int r ← head.h 3 stack[1...r] ← ∅ 4 while r ≥ 0 do 5 while u.next[r].x < x do 6 u ← u.next[r] // gremo desno v seznamu L_r 7 if u.next[r].x = x then 8 u.next[r].count++ 9 return true // če je ključ že v seznamu 10 stack[r--] ← u // gremo dol in shranimo u 11 Node w ← new Node(x,pickHeight()) // ustvarimo novo vozlišče s ključem x in naključno višino 12 while head.h < w.h do 13 head.h++ 14 stack[head.h] ← head // povišamo glavo, če je novi element višji od vseh dozdajšnjih 15 for i = 0...w.h do 16 /* v tej zanki izvajamo potrebne prevezave */ 17 w.next[i] ← stack[i].next[i] 18 stack[i].next[i] ← w 18 return true ; </pre>

c)

Privzel sem, da kovanec 1 pomeni povišanje elementa in 0 vstavljanje pri dobljeni višini.

pred vstavi (12):

2 (h=2)

3 (h=2)

7 (h=1)

8 (h=3)

13 (h=1)

19 (h=1)

po vstavi(12):

2 (h=2)

3 (h=2)

7 (h=1)

8 (h=3)

12 (h=4)

13 (h=1)

19 (h=1)

pred briši(19):

2 (h=2)

3 (h=2)

7 (h=1)

8 (h=3)

12 (h=4)

13 (h=1)

19 (h=1)

26 (h=2)

po briši(19):

2 (h=2)

3 (h=2)

7 (h=1)

8 (h=3)

12 (h=4)

13 (h=1)

26 (h=2)

pred vstavi(16):

3 (h=2)

7 (h=1)

8 (h=3)

12 (h=4)

13 (h=1)

26 (h=2)

6

29 (h=1)

po vstavi(16):

3 (h=2)

7 (h=1)

8 (h=3)

12 (h=4)

13 (h=1)

16 (h=1)

26 (h=2)

29 (h=1)

Naloga 2.3

a)

(4 (5))

(1 (6 3 10 (9)))

(7 (8))

(2)

Find(2):

isto

Find(10):

isto

Find(9):

(4 (5))

(1 (6 3 10 9))

(7 (8))

(2)

Find(8):

isto

Dobimo štiri disjunktne množice.

b)

```

Union(1,2) // paroma sosede
Union(3,4)
Union(5,6)
...
Union(n-1, n)

```

```

Union(1,3) // paroma sosede s po dvema elementoma
Union(5,7)
Union(9,11)
...
Union(n-3, n-1)

```

```

Union(1,5) // paroma sosede s po štiri elementi
Union(9,13) ... Union(n-7, n-3)

```

itd.

Najslabša možna višina tako dobljene disjunktne množice je $\lg n$. Najdražja operacija je iskanje predstavnika od listov, npr. Find-Set(n).

c)

```

1 class Node:
2     parent
3     children = [] # sorted array
4     h # height
5
6     def remove_child_and_update_h(v, child):
7         v.children.remove(child)
8         v.h = max(v.children)+1
9
10 Find-Set(v, path=list()):
11     if v.parent:
12         return v.parent.Find(v, path+self)
13
14     # we reached root
15     # path compression
16     for elt in path[:-1]:
17         elt.parent.remove_child_and_update_h( elt )
18         elt.parent = self
19
20     return self

```

Naloga 2.4 (programerska)

```

1 package psa.naloga2;
2
3 import static java.lang.Math.floor;
4 import static java.lang.Math.sqrt;
5
6 public class HashFunction {
7     public static enum HashingMethod {
8         DivisionMethod,
9         KnuthMethod
10    };
11
12    public static int DivisionMethod(int k, int m) {
13        if (k<0) { k *= -1; }
14        int i = k % m;
15        if (i<0) {
16            i+=m;
17        }
18
19        return i;
20    }
21
22    public static int KnuthMethod(int k, int m) {
23        if (k<0) { k *= -1; }
24        double PHI = (sqrt(5) - 1)/2.0;
25        return (int)floor(m*(k*PHI - floor(k*PHI)));
26    }
27 }

1 package psa.naloga2;
2
3 import java.util.LinkedList;
4
5 public class HashSetChaining {
6     private LinkedList<Integer> table[];
7     private HashFunction.HashingMethod h;
8
9     public HashSetChaining(int m, HashFunction.HashingMethod h) {
10        this.h = h;
11        this.table = new LinkedList[m];
12        for (int i=0; i<table.length; i++) {
13            table[i] = new LinkedList<>();
14        }
15    }
16
17    public LinkedList<Integer>[] getTable() {

```



```

18     return this.table;
19 }
20
21 public boolean add(int k) {
22     int idx = Integer.MIN_VALUE;
23     if (this.h==HashFunction.HashingMethod.DivisionMethod) {
24         idx = HashFunction.DivisionMethod(k, this.table.length);
25     } else
26     if (this.h==HashFunction.HashingMethod.KnuthMethod) {
27         idx = HashFunction.KnuthMethod(k, this.table.length);
28     }
29     if (!this.table[idx].contains(k)) {
30         this.table[idx].add(k);
31         return true;
32     }
33
34     return false;
35 }
36
37 public boolean remove(int k) {
38     int idx = Integer.MIN_VALUE;
39     if (this.h==HashFunction.HashingMethod.DivisionMethod) {
40         idx = HashFunction.DivisionMethod(k, this.table.length);
41     } else
42     if (this.h==HashFunction.HashingMethod.KnuthMethod) {
43         idx = HashFunction.KnuthMethod(k, this.table.length);
44     }
45     return this.table[idx].remove(new Integer(k));
46 }
47
48 public boolean contains(int k) {
49     int idx = Integer.MIN_VALUE;
50     if (this.h==HashFunction.HashingMethod.DivisionMethod) {
51         idx = HashFunction.DivisionMethod(k, this.table.length);
52     } else
53     if (this.h==HashFunction.HashingMethod.KnuthMethod) {
54         idx = HashFunction.KnuthMethod(k, this.table.length);
55     }
56
57     return this.table[idx].contains(k);
58 }
59 }

1 package psa.naloga2;
2
3 public class HashSetOpenAddressing {
4     private int table[]; // table content, Integer.MIN_VALUE, if

```

```

        element not present
5  private HashFunction.HashingMethod h;
6  private CollisionProbeSequence c;
7
8  public static enum CollisionProbeSequence {
9      LinearProbing,      // new  $h(k) = (h(k) + i) \bmod m$ 
10     QuadraticProbing,   // new  $h(k) = (h(k) + i^2) \bmod m$ 
11     DoubleHashing      // new  $h(k) = (h(k) + i * h(k)) \bmod m$ 
12 };
13
14 public HashSetOpenAddressing(int m,
    HashFunction.HashingMethod h, CollisionProbeSequence c) {
15     this.table = new int[m];
16     this.h = h;
17     this.c = c;
18
19     for (int i=0; i<m; i++) {
20         table[i] = Integer.MIN_VALUE;
21     }
22 }
23
24 public int[] getTable() {
25     return this.table;
26 }
27
28 public boolean add(int k) {
29     int startidx=Integer.MIN_VALUE;
30     if (this.h==HashFunction.HashingMethod.DivisionMethod) {
31         startidx = HashFunction.DivisionMethod(k,
            this.table.length);
32     } else
33     if (this.h==HashFunction.HashingMethod.KnuthMethod) {
34         startidx = HashFunction.KnuthMethod(k,
            this.table.length);
35     }
36
37     int idx=startidx;
38     for (int i=0; i<this.table.length &&
        this.table[idx]!=Integer.MIN_VALUE; i++) {
39         switch (this.c) {
40             case LinearProbing: {
41                 idx = (idx+1) % this.table.length;
42                 break;
43             }
44             case QuadraticProbing: {
45                 idx = (startidx + (i+1)*(i+1)) % this.table.length;
46                 break;

```

```

47         }
48         case DoubleHashing: {
49             idx = (startidx + i*startidx) % this.table.length;
50             break;
51         }
52     }
53 }
54
55 if (this.table[idx]==Integer.MIN_VALUE) {
56     // found empty slot, insert element
57     this.table[idx] = k;
58     return true;
59 }
60
61 return false;
62 }
63
64 public boolean remove(int k) {
65     int startidx=Integer.MIN_VALUE;
66     if (this.h==HashFunction.HashingMethod.DivisionMethod) {
67         startidx = HashFunction.DivisionMethod(k,
68             this.table.length);
69     } else
69     if (this.h==HashFunction.HashingMethod.KnuthMethod) {
70         startidx = HashFunction.KnuthMethod(k,
71             this.table.length);
72     }
73
74     int idx=startidx;
75     // Cheating! Should exit when we found an empty slot where
76     // no element has been inserted yet. Test cases work
77     // though ;)
78     for (int i=0; i<this.table.length && this.table[idx]!=k;
79         i++) {
80         switch (this.c) {
81             case LinearProbing: {
82                 idx = (idx+1) % this.table.length;
83                 break;
84             }
85             case QuadraticProbing: {
86                 idx = (startidx + (i+1)*(i+1)) % this.table.length;
87                 break;
88             }
89             case DoubleHashing: {
90                 idx = (startidx + i*startidx) % this.table.length;
91                 break;
92             }
93         }
94     }
95 }

```

```

89     }
90 }
91
92 if (this.table[idx]==k) {
93     this.table[idx] = Integer.MIN_VALUE;
94     return true;
95 }
96
97 return false;
98 }
99
100 public boolean contains(int k) {
101     int startidx=Integer.MIN_VALUE;
102     if (this.h==HashFunction.HashingMethod.DivisionMethod) {
103         startidx = HashFunction.DivisionMethod(k,
104             this.table.length);
105     } else
106     if (this.h==HashFunction.HashingMethod.KnuthMethod) {
107         startidx = HashFunction.KnuthMethod(k,
108             this.table.length);
109     }
110
111     int idx=startidx;
112     // Cheating! Should exit when we found an empty slot where
113     // no element has been inserted yet. Test cases work
114     // though ;)
115     for (int i=0; i<this.table.length && this.table[idx]!=k;
116         i++) {
117         switch (this.c) {
118             case LinearProbing: {
119                 idx = (idx+1) % this.table.length;
120                 break;
121             }
122             case QuadraticProbing: {
123                 idx = (startidx + (i+1)*(i+1)) % this.table.length;
124                 break;
125             }
126             case DoubleHashing: {
127                 idx = (startidx + i*startidx) % this.table.length;
128                 break;
129             }
130         }
131     }
132     return this.table[idx]==k;
133 }

```