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:

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

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

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

$(13 \cdot 0) \bmod 5 = 0$ A

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

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

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

Tabela izgleda takole:

$0 : O \to A \to T$

$1 :$

$2 : D \to I$

$3 : P \to K$

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:

```
#!/usr/bin/python

# P O D A T K I
elements = [16, 15, 4, 0, 20, 11, 9]
print(findMinKandM(elements))

def findMinKandM(elements):
    m=len(elements)
    while (True):
        for k in range(1,m):
            T = [0]*m # array of zeros
            for elt in elements:
                T[ (k*elt) % m ] += 1

        i=0
        while (i<m):
            if (T[i] > 1):
                # we have duplicate, find another k and m
                break
            i+=1

        if (i==m):
            # we didn't have any duplicates return k and m
            return (k,m)

        m+=1
```

c)

5 => T1[0]

12 => T1[2]

28 => 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)

```
Data: ključ x
Result: true, če je element vstavljen; false v primeru, da zmanjka
        pomnilnika (izpuščeno v kodi spodaj)
Node u ← head
int r ← head.h
stack[1 ... r] ← ∅
while r ≥ 0 do
    while u.next[r].x < x do
        u ← u.next[r] // gremo desno v seznamu L_r
    if u.next[r].x = x then
        u.next[r].count++
        return true // če je ključ že v seznamu
    stack[r − −] ← u // gremo dol in shranimo u
Node w ← new Node(x,pickHeight()) // ustvarimo novo vozlišče s
ključem x in naključno višino
while head.h < w.h do
    head.h + +
    stack[head.h] ← head // povišamo glavo, če je novi element višji od
    vseh dozdajšnjih
for i = 0 ... w.h do
    /* v tej zanki izvajamo potrebne prevezave */
    w.next[i] ← stack[i].next[i]
    stack[i].next[i] ← w
return true ;
```

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)

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)

```
class Node:
  parent
  children = [] # sorted array
  h # height

  def remove_child_and_update_h(v, child):
    v.children.remove(child)
    v.h = max(v.children)+1

Find-Set(v, path=list()):
  if v.parent:
    return v.parent.Find(v, path+self)

  # we reached root
  # path compression
  for elt in path[:-1]:
    elt.parent.remove_child_and_update_h( elt )
    elt.parent = self

  return self
```

Naloga 2.4 (programerska)

```java
package psa.naloga2;

import static java.lang.Math.floor;
import static java.lang.Math.sqrt;

public class HashFunction {
  public static enum HashingMethod {
    DivisionMethod,
    KnuthMethod
  };

  public static int DivisionMethod(int k, int m) {
    if (k<0) { k *= -1; }
    int i = k % m;
    if (i<0) {
      i+=m;
    }

    return i;
  }

  public static int KnuthMethod(int k, int m) {
    if (k<0) { k *= -1; }
    double PHI = (sqrt(5) - 1)/2.0;
    return (int)floor(m*(k*PHI - floor(k*PHI)));
  }
}
```

```java
package psa.naloga2;

import java.util.LinkedList;

public class HashSetChaining {
  private LinkedList<Integer> table[];
  private HashFunction.HashingMethod h;

  public HashSetChaining(int m, HashFunction.HashingMethod h) {
    this.h = h;
    this.table = new LinkedList[m];
    for (int i=0; i<table.length; i++) {
      table[i] = new LinkedList<>();
    }
  }

  public LinkedList<Integer>[] getTable() {
```

```
    return this.table;
  }

  public boolean add(int k) {
    int idx = Integer.MIN_VALUE;
    if (this.h==HashFunction.HashingMethod.DivisionMethod) {
      idx = HashFunction.DivisionMethod(k, this.table.length);
    } else
    if (this.h==HashFunction.HashingMethod.KnuthMethod) {
      idx = HashFunction.KnuthMethod(k, this.table.length);
    }
    if (!this.table[idx].contains(k)) {
      this.table[idx].add(k);
      return true;
    }

    return false;
  }

  public boolean remove(int k) {
    int idx = Integer.MIN_VALUE;
    if (this.h==HashFunction.HashingMethod.DivisionMethod) {
      idx = HashFunction.DivisionMethod(k, this.table.length);
    } else
    if (this.h==HashFunction.HashingMethod.KnuthMethod) {
      idx = HashFunction.KnuthMethod(k, this.table.length);
    }
    return this.table[idx].remove(new Integer(k));
  }

  public boolean contains(int k) {
    int idx = Integer.MIN_VALUE;
    if (this.h==HashFunction.HashingMethod.DivisionMethod) {
      idx = HashFunction.DivisionMethod(k, this.table.length);
    } else
    if (this.h==HashFunction.HashingMethod.KnuthMethod) {
      idx = HashFunction.KnuthMethod(k, this.table.length);
    }

    return this.table[idx].contains(k);
  }
}
```

```
package psa.naloga2;

public class HashSetOpenAddressing {
  private int table[]; // table content, Integer.MIN_VALUE, if
```

```java
    element not present
private HashFunction.HashingMethod h;
private CollisionProbeSequence c;

public static enum CollisionProbeSequence {
  LinearProbing,    // new h(k) = (h(k) + i) mod m
  QuadraticProbing, // new h(k) = (h(k) + i^2) mod m
  DoubleHashing     // new h(k) = (h(k) + i*h(k)) mod m
};

public HashSetOpenAddressing(int m,
    HashFunction.HashingMethod h, CollisionProbeSequence c) {
  this.table = new int[m];
  this.h = h;
  this.c = c;

  for (int i=0; i<m; i++) {
    table[i] = Integer.MIN_VALUE;
  }
}

public int[] getTable() {
  return this.table;
}

public boolean add(int k) {
  int startidx=Integer.MIN_VALUE;
  if (this.h==HashFunction.HashingMethod.DivisionMethod) {
    startidx = HashFunction.DivisionMethod(k,
        this.table.length);
  } else
  if (this.h==HashFunction.HashingMethod.KnuthMethod) {
    startidx = HashFunction.KnuthMethod(k,
        this.table.length);
  }

  int idx=startidx;
  for (int i=0; i<this.table.length &&
      this.table[idx]!=Integer.MIN_VALUE; i++) {
    switch (this.c) {
      case LinearProbing: {
        idx = (idx+1) % this.table.length;
        break;
      }
      case QuadraticProbing: {
        idx = (startidx + (i+1)*(i+1)) % this.table.length;
        break;
```

```
        }
        case DoubleHashing: {
          idx = (startidx + i*startidx) % this.table.length;
          break;
        }
      }
    }

    if (this.table[idx]==Integer.MIN_VALUE) {
      // found empty slot, insert element
      this.table[idx] = k;
      return true;
    }

    return false;
  }

  public boolean remove(int k) {
    int startidx=Integer.MIN_VALUE;
    if (this.h==HashFunction.HashingMethod.DivisionMethod) {
      startidx = HashFunction.DivisionMethod(k, this.table.length);
    } else
    if (this.h==HashFunction.HashingMethod.KnuthMethod) {
      startidx = HashFunction.KnuthMethod(k, this.table.length);
    }

    int idx=startidx;
    // Cheating! Should exit when we found an empty slot where no element has been inserted yet. Test cases work though ;)
    for (int i=0; i<this.table.length && this.table[idx]!=k; i++) {
      switch (this.c) {
        case LinearProbing: {
          idx = (idx+1) % this.table.length;
          break;
        }
        case QuadraticProbing: {
          idx = (startidx + (i+1)*(i+1)) % this.table.length;
          break;
        }
        case DoubleHashing: {
          idx = (startidx + i*startidx) % this.table.length;
          break;
        }
```

```java
      }
    }

    if (this.table[idx]==k) {
      this.table[idx] = Integer.MIN_VALUE;
      return true;
    }

    return false;
  }

  public boolean contains(int k) {
    int startidx=Integer.MIN_VALUE;
    if (this.h==HashFunction.HashingMethod.DivisionMethod) {
      startidx = HashFunction.DivisionMethod(k,
         this.table.length);
    } else
    if (this.h==HashFunction.HashingMethod.KnuthMethod) {
      startidx = HashFunction.KnuthMethod(k,
         this.table.length);
    }

    int idx=startidx;
    // Cheating! Should exit when we found an empty slot where
       no element has been inserted yet. Test cases work
       though ;)
    for (int i=0; i<this.table.length && this.table[idx]!=k;
       i++) {
      switch (this.c) {
        case LinearProbing: {
          idx = (idx+1) % this.table.length;
          break;
        }
        case QuadraticProbing: {
          idx = (startidx + (i+1)*(i+1)) % this.table.length;
          break;
        }
        case DoubleHashing: {
          idx = (startidx + i*startidx) % this.table.length;
          break;
        }
      }
    }

    return this.table[idx]==k;
  }
}
```