Algorithms and Data Structure 9

Yiping Deng

April 23, 2018

Problem 1

a)

```
def h1(k):
    # first hash function
    return k % 5

def h2(k):
    # second hash function
    return (7 * k) % 8

def h(k, i):
    # the double hashing function with size 5
    return (h1(k) + h2(k) * i) % 5

if __name__ == '__main__':
    lst = [3, 10, 2, 4]
    for i in lst:
        print('{} with i = 0 hashes to {}'.format(i, h(i, 0)))
```

With the following python program, we execute the double hashing strategy. After running the program, we found 0 collision.

b)

This is the code of the implementation

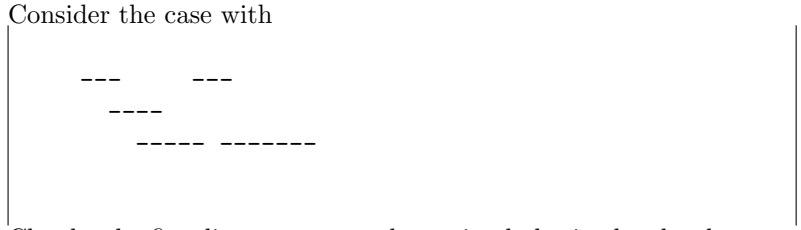
```
class Node:
    def __init__(self, key = None, value = None):
        # just a nod3
        self.key = key
        self.value = value

class HashTable:
    def __init__(self):
        # with initial size of 100
        self.maxSize = 100
        self.currentSize = 0
        self.arr = [None] * self.maxSize

    def hashCode(self, key):
        # use the default hash of python
        return key.__hash__() % self.maxSize

    def insertNode(self, key, value):
        # insert a node into the hashTable
        if self.arr[self.hashCode(key)] is None:
```

```
            # if no conflict
            self.arr[self.hashCode(key)] = Node(key, value)
        else:
            # if conflict
            code = self.hashCode(key)
            while self.arr[code] is not None:
                code = code + 1
                if code >= self.maxSize:
                    raise RuntimeError("overflow")
            self.arr[code] = Node(key, value)

        # increase the size
        self.currentSize = self.currentSize + 1

    def get(self, key):
        # get the hashcode
        code = self.hashCode(key)
        if self.arr[code] is None:
            # entry no found
            return None
        else:
            # check if the key is right
            while self.arr[code].key != key:
                code = code + 1

                # if no found again
                if code >= self.maxSize or self.arr[code] is None:
                    return None
            return self.arr[code].value

    def isEmpty(self):
        return currentsize == 0
```

Problem 2

a)

Consider the case with



Clearly, the first line represents the optimal obtains by the shortest duration. It is not a global optimal, since you can actually do 4 activity if you don't include the second activity in the first line.

b)

```
class Activity:
    def __init__(self, start, end):
        self.start = start
        self.end = end

    def __str__(self):
        return "start:{} end:{}".format(self.start, self.end)

    def overlaps(self, other):
```

```
        # check if two activities overlaps
        if not (self.start >= other.end or self.end <= other.start):
            print("overlaps: first - {} second - {}".format(self, other))
            return True
        else:
            return False

def hasConflict(selected, current):
    # check if it has any collision in the list
    for s in selected:
        if s.overlaps(current):
            return True
    return False

def selectActivity(activities, selectedList = []):
    # base case
    if activities == []:
        return 0

    # break the list into head, which is called current
    # and tail, which is called remaining
    current, *remaining = activities

    if hasConflict(selectedList, current):
        return selectActivity(remaining, selectedList)
    else:
        # either you include the head or not
        included = 1 + selectActivity(remaining, selectedList + [current])
        excluded = selectActivity(remaining, selectedList)

        # find the biggest one
        if included > excluded:
            return included
        return excluded

def includeLastSelection(activities):
    # force inclusion of the last activity
    return 1 + selectActivity(activities[:-1], activities[-1:])
```

We have a concrete implementation of such a algorithm. We recursively select activities, one-by-one, and find the maximum activities you can include.