Homework 2, CSE 232

Due September 10 Note: On most of the problem sets through the semester, I'll put a horizontal line with "Optional" under it. Any problems below this section are encouraged - I think they're interesting and will help you learn the subject - but not necessary to complete in order to get credit for the homework.

Problem 1

This week, we're going to go through TOJ 1037 - Memory Management.

Here's the problem statement (rewritten slightly for improved clarity):

> Our operating system will allocate memory in pieces that we'll call "blocks". The blocks are numbered by integers from 1 to N. When the operating system needs more memory it makes a request to the memory management module. To process this request, the memory management module needs to find the free memory block with the least number. You may assume that there are enough blocks to process all requests. Now we should define the meaning of words "free block". At the moment of the first request to the memory management module all blocks are considered to be free. A block also becomes free when there have been no requests to it for T minutes. The memory management module may also request access to previously allocated blocks. To process this request the memory management module should first check if the requested block is really allocated. If it is, the request is considered to be successful and the block remains allocated for a further T minutes. Otherwise the request fails. You are to implement these algorithms for $N = 30,000$ and $T = 10$ minutes.
>
> **Input** Each line of input contains a request for memory block allocation or memory block access. Memory allocation request has the form:
>
> $Time$ +
>
> where $Time$ is a nonnegative integer number not greater than 65,000. Time is given in seconds. Memory block access request has the form:
>
> $Time$. $BlockNo$
>
> where Time is as described above and $BlockNo$ is an integer value from 1 to N. There will be no more than 80000 requests.
>
> **Output** For each line of input you should print exactly one line with the result of the request processing. For a memory allocation request you are to write an integer with the number of the allocated block. As it was mentioned above you may assume that every request can be satisfied, and there will be no more than N simultaneously allocated blocks. For a memory block access request you should print only the character '+' if the request is successful (i.e. the block is really allocated) or '-' if the request fails (i.e. the block with number given is free, so it can't be accessed). Requests are arranged by their times in an increasing order. Requests with equal times should be processed as they appear in input.

To solve this problem, we'll need some way to quickly perform two different operations -

- Allocate a free memory block, which requires us to quickly identify which free block has the lowest index. E.g., if blocks 1-5 and 10-15 are allocated, it needs to be able to quickly report that block #6 is free.

- Access an allocated memory block, which requires us to quickly access a particular block, tell if it is free or not, and update the time it was last accessed.

a. Assuming that we write functions to perform these two operations, what should be the inputs and outputs for those functions? That is, what input data does each function need to do their job and what do we need them to return in order to print the output requested for this problem?

Here's some dummy functions with the correct input/output:

```
In []: def allocate_block(time):
           """Given the time of the request, return the block ID of the lowest ID block which is free a
           #CODE GOES HERE
           return block_id

       def access_block(time, block_id):
           """Given the time of the request and the block ID reqested, return True or False depending o
           the request was successful."""
           #CODE GOES HERE
           return success
```

After some thought, you've decided to use one data structure to contain blocks that are free and another data structure to contain blocks that have been allocated. You'll then need to move blocks back and forth between these data structures as they become freed and allocated.

b. When are you going to transfer blocks from the freed blocks to the allocated ones? Which blocks are you going to be moving?

One block will be moved from free to allocated every time we call the `allocate_block` function. The block moved will be whichever free block has the lowest ID.

c. When are you going to transfer blocks from the allocated blocks to the freed ones? Which blocks are you going to be moving?

Every time we call either the `allocate_block` or `access_block` functions, we will first need to move all blocks from allocated to freed that have expiration times earlier than the time of the request.

d. Given your answers to parts b. and c. and the data structures you've learned about, what would be good data structures to use to contain the freed and the allocated blocks and how would you organize them (i.e., what should be the primary key?).

We'll use priority queues for both the free and allocated blocks. For free blocks, we'll use a min queue keyed to the block ID, so that it gives quick access to the free block with the lowest ID. For allocated blocks, we'll use a min queue keyed to the time that block was last accessed, so that we can quickly identify the blocks that were accessed more than T minutes ago and move them back to the free blocks.

e. The basic data structures I described in class should be sufficient to solve the problem if only block allocation requests are given. Write up a solution to this problem assuming that there will be no access requests in the input stream and attach it to your homework. Some sample input/ouput following those restrictions is given below.

```
In []: import sys
       import heapq

       DELAY = 10 * 60 #Switch to seconds
       N = 30000

       free_blocks = range(1, N+1)
```

```
heapq.heapify(free_blocks)
allocated_blocks = []

def release_blocks(t):
    while allocated_blocks and allocated_blocks[0][0] <= t:
        entry = heapq.heappop(allocated_blocks)
        block = entry[1]
        heapq.heappush(free_blocks, block)

def allocate_block(t):
    release_blocks(t)
    block = heapq.heappop(free_blocks)
    entry = [t + DELAY, block]
    heapq.heappush(allocated_blocks, entry)
    return block

#Parse the input data
for line in sys.stdin:
    line = line.split()
    t = int(line[0])
    print allocate_block(t)
```

Sample Input (only allocation requests) `1 +`

```
4 +
8 +
301 +
602 +
602 +
900 +
1205 +
```

Sample Output (only allocation requests) `1`

```
2
3
4
1
5
2
1
```

Optional

f. Solving the problem with block access requests requires augmenting the data structure used to hold allocated blocks so that you can access blocks based on something other than their primary key. What's the secondary key that you'll need to support block access requests?

The secondary key will have to be the block ID, so that we can update the expiration time of any particular block in the allocated block queue.

g. A good way to augment data structures with a secondary key is to create a dictionary keyed on the secondary key containing pointers to elements in the primary data structure. Augment your solution from part e. like this and see if you can use it to update block access times.

```
In []: #N.B. This code gets the wrong answer!

       import sys
       import heapq

       DELAY = 10 * 60 #Switch to seconds
       N = 30000

       free_blocks = range(1, N+1)
       heapq.heapify(free_blocks)
       allocated_blocks = []
       locations = {}

       def release_blocks(t):
           while allocated_blocks and allocated_blocks[0][0] <= t:
               entry = heapq.heappop(allocated_blocks)
               block = entry[1]
               heapq.heappush(free_blocks, block)
               del locations[block]

       def allocate_block(t):
           release_blocks(t)
           block = heapq.heappop(free_blocks)
           entry = [t + DELAY, block]
           heapq.heappush(allocated_blocks, entry)
           return block

       def access_block(t, b):
           release_blocks(t)
           if not b in locations:
               return '-'
           locations[b][0] = t + DELAY
           return '+'

       #Parse the input data
       for line in sys.stdin:
           line = line.split()
           if line[1] == '+':
               t = int(line[0])
               print allocate_block(t)
           elif line[1] == '.':
               t, b = int(line[0]), int(line[2])
               print access_block(t, b)
```

h. You may have noticed that directly updating elements in your primary data structure using the augmenting dictionary can break the properties that make it useful. Can you think of an alternative way of updating elements? (hint - what would happen if you had the same block in the data structure multiple times?)

It looks like directly updating elements in the priority queue breaks the heap property, so that we're no longer always getting the minimum value when we need it. Instead, we could mark the old block so that we ignore it, keep it in the queue, and insert a new block with the correct ID and expiration time.

```
In []: import sys
       import heapq
```

```
DELAY = 10 * 60 #Swich to seconds to match the input data
N = 30000
CLEARED = 'block_cleared'

#Initialize a priority queue for free block IDs with all blocks starting as free
free_blocks = range(1, N + 1)
heapq.heapify(free_blocks)

#Initialize priority queue for allocated blocks along with the time when they will be freed
allocated_blocks = []

#We'll also store a dictionary of all allocated blocks so we can quickly check whether a block
#is available or not
locations = {}

#Now let's set up some functions to handle allocating, accessing, and releasing blocks
def release_blocks(t):
    """Release any allocated blocks that have expired by time t, pushing them back onto the
    free block priority queue."""
    while allocated_blocks and allocated_blocks[0][0] <= t:
        entry = heapq.heappop(allocated_blocks)
        block = entry[1]
        if block is not CLEARED:
            heapq.heappush(free_blocks, entry[1])
            del locations[block]

def allocate_block(t):
    """Find the first free block and allocate it at time t to be freed at time t + DELAY."""
    release_blocks(t)

    block = heapq.heappop(free_blocks)
    entry = [t + DELAY, block]
    locations[block] = entry
    heapq.heappush(allocated_blocks, entry)
    return block

def access_block(t, b):
    """Access block b at time t, increasing its expiration time by marking the old block as
    CLEARED and inserting a new block with the new expiration time into the priority queue

    Returns True if successful and False if unsuccessful (because block b isn't allocated)."""
    release_blocks(t)

    if not b in locations:
        return False

    entry = locations.pop(b)
    entry[-1] = CLEARED
    entry = [t + DELAY, b]
    locations[b] = entry
    heapq.heappush(allocated_blocks, entry)

    return True
```

```
#And now parse the input and print the output
for line in sys.stdin:
    line = line.split()
    if len(line) == 2:
        t = int(line[0])
        print allocate_block(t)
    elif len(line) == 3:
        t, b = int(line[0]), int(line[2])
        if access_block(t, b):
            print '+'
        else:
            print '-'
```

Other Problems of Interest TOJ 1019. Line Painting (think about segment trees!)