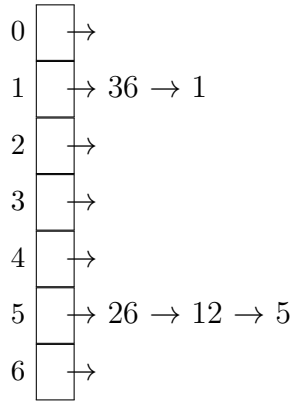
CS2210 Assignment 3

Adam Gale

October 25, 2023

Question 1

Consider a hash table of size $M = 7$ where we are going to store integer key values. The hash function is $h(k) = k \bmod 7$. Draw the table that results after inserting, in the given order, the following values: 5, 12, 1, 36, 26. Assume that collisions are handled by separate chaining.



Note: Assumed that our separate chaining solution added to the head of the linked list.

Question 2

Show the result of the previous exercise, assuming collisions are handled by linear probing. For reference: The hash function is $h(k) = k \bmod 7$. Draw the table that results after inserting, in the given order, the following values: 5, 12, 1, 36, 26.

0	26
1	1
2	36
3	
4	
5	5
6	12

Question 3

Repeat exercise (1) assuming collisions are handled by double hashing, using a secondary hash function $h'(k) = 5 - (k \bmod 5)$. For reference: The hash function is $h(k) = k \bmod 7$. Draw the table that results after inserting, in the given order, the following values: 5, 12, 1, 36, 26.

0	
1	12
2	1
3	26
4	
5	5
6	36

Showing my work:

$h(5) = 5 \bmod 7 = 5$ — no collision → put 5 in 5

$h(12) = 12 \bmod 7 = 5$ — collision → check $h(12) + h'(12)$

$h(12) + h'(12) = 5 + (5 - (12 \bmod 5)) = 5 + 3 = 8 \bmod 7 = 1$ — no collision → put 12 in 1

$h(1) = 1 \bmod 7 = 1$ — collision → check $h(1) + h'(1)$

$h(1) + h'(1) = 1 + (5 - (1 \bmod 5)) = 1 + 4 = 5$ — collision → check $h(1) + 2h'(1)$

$h(1) + 2h'(1) = 1 + 2(4) = 1 + 8 = 9 \bmod 7 = 2$ — no collision → put 1 in 2

$h(36) = 36 \bmod 7 = 1$ — collision → check $h(36) + h'(36)$

$h(36) + h'(36) = 1 + (5 - (36 \bmod 5)) = 1 + 4 = 5$ — collision → check $h(36) + 2h'(36)$

$h(36) + 2h'(36) = 1 + 2(4) = 1 + 8 = 9 \bmod 7 = 2$ — collision → check $h(36) + 3h'(36)$

$h(36) + 3h'(36) = 1 + 3(4) = 1 + 12 = 13 \bmod 7 = 6$ — no collision → put 36 in 6

$h(26) = 26 \bmod 7 = 5$ — collision → check $h(26) + h'(26)$

$h(26) + h'(26) = 5 + (5 - (26 \bmod 5)) = 5 + 4 = 9 \bmod 7 = 2$ — collision → check $h(26) + 3h'(26)$

$h(26) + 2h'(26) = 5 + 2(4) = 5 + 8 = 13 \bmod 7 = 6$ — collision → check $h(26) + 3h'(26)$

$h(26) + 3h'(26) = 5 + 3(4) = 5 + 12 = 17 \bmod 7 = 3$ — no collision → put 26 in 3

so many collisions, argh!

Question 4

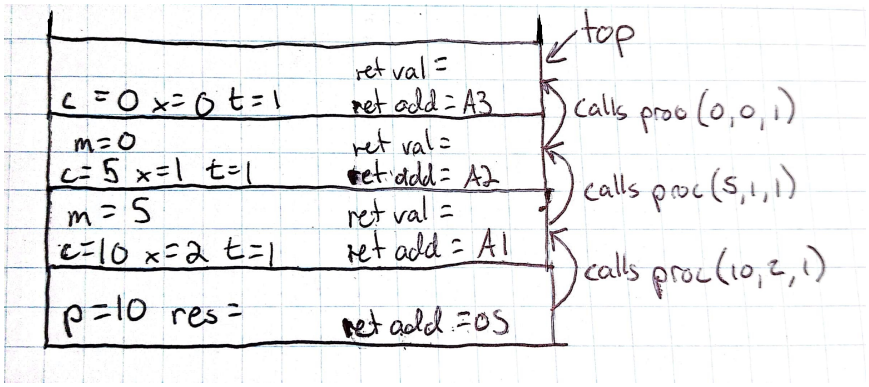
Consider the following algorithms.

Algorithm main()
  p← 10
  res = proc(p,2,1) (A1)
  print(res)

Algorithm proc(c, x, t)
  if $x = 0$ **then**
    return c (***)
  $m \leftarrow c - 5$
  if $m > t$ **then**
    return proc $(m, x - 1, t)$ (A2)
  else return proc $(2 * m, x - 1, t)$ (A3)

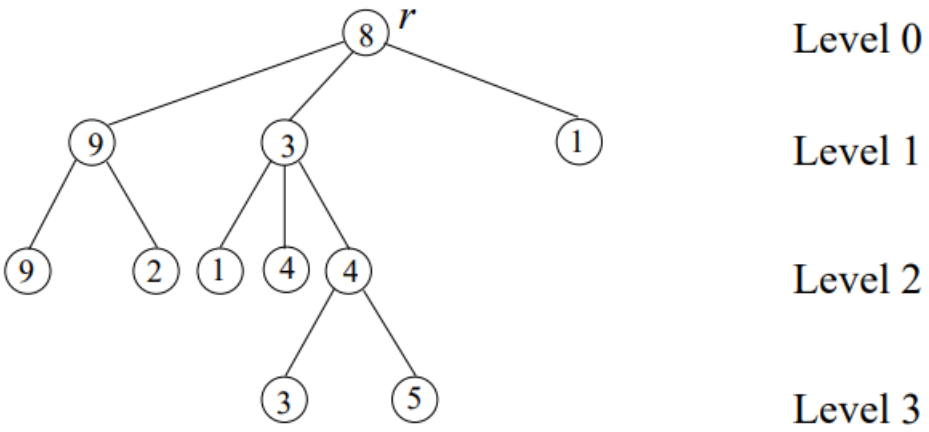
Draw the execution stack up to the moment just before the instruction marked (***) is about to be executed, i.e. run the algorithm and draw the activation records in the execution stack and stop the execution of the algorithm just before the value of c is to be returned. Show what the execution stack looks like at that moment. The addresses of statements where algorithms are invoked have been marked in the pseudocode as A1, A2, and A3. To help you out we have drawn the first activation record, for algorithm main.

I couldn't figure out how to do this in latex lol:



Question 5

Write in detailed pseudocode, like the one used in class, an algorithm `find(r,k,v)` that receives as input the root `r` of a tree and two integer values `k` ≥ 0, and `v`, and it returns the number of nodes at level `k` storing the value `v`. For example, for the tree below with root `r` the invocation `find(r,2,4)` must return the value 2 as there are two nodes as level 2 that store the value 4, the invocation `find(r,3,9)` must return 0 as no node at level 3 stores the value 9, and `find(r,5,4)` must also return 0 as there are no nodes at level 5 in this tree.



For a node r let r.isLeaf()[1] return *true* if r is a leaf and $false$ otherwise. To get the value stored in a node r use r.getValue(). To access the children of a node r use the following pseudocode:

for each child c of r **do**

Hint. In the initial call to the algorithm the value of the first parameter is the root of the tree and the value of the second parameter is the target level. Every time that the algorithm is invoked recursively the value of the second parameter must be updated how?

[1]Note: I am assuming that .isLeaf() returns True for the nodes (9, 2, 1, 4, 3, 5, 1) in the above tree - that is, nodes without non-null children

Pseudocode:

Algorithm:`find(r,k,v)`
In: Root r of a tree, value $k \geq 0$, and value v
Out: Number of nodes at level k storing the value v

```
count ← 0                                                  initialize counter
if r is null then return 0                                 edgecase of root being null
if k = 0                                                   check to see if this is the level we want to check
    if r.getValue() = v return 1                           if this is the right level and data matches, return 1
    return 0                                               if this is the right level and data doesn't match, return 0
else if r.isLeaf() = false                                 if not the correct level, check if r has children to iterate over
    for each child c of r do                               iterate through children nodes
        count ← count + find(c, k − 1, v)                  sums returns from recursive child calls with decremented k
return count                                               returns final sum of matching nodes
```

Question 6

Compute the time complexity of the following algorithm. You must explain how many operations are performed per call, how many calls to the algorithm are made, how many operations the algorithm performs in total, and what the order of the time complexity is.

Algorithm func(n) performs $c_1 \log n$ operations, where c_1 is a constant.

$$c_2 = \left\{ \quad \textbf{if } r \text{ is a leaf } \textbf{then return } n + \texttt{func}(n) \right.$$

$$c_3 = \left\{ \begin{array}{l} \textbf{else } \{ \\ \quad v \leftarrow \texttt{func}(n) \\ \quad v \leftarrow v + \texttt{algo}(r.\text{leftChild}()) + \texttt{algo}(r.\text{rightChild}()) \\ \quad \textbf{return } v \\ \} \end{array} \right.$$

Compute the complexity ignoring recursive calls (complexity of one call):
- $c_2 + c_1 \log n$ if r is a leaf
- $c_3 + c_1 \log n$ if r is not a leaf

Compute the number of recursive calls:
- Each internal node calls the alogrithm twice
- Leaf nodes do not make recursive calls
- There are $\frac{n-1}{2}$ internal nodes in a proper binary tree, so $n-1$ recursive calls are made
- Including the inital call, n calls are made to the algorithm in total

Combine number of calls with the complexity of each call:
- n calls are made, each with a constant amount of operations and a $c_1 \log n$ call to func
- We know this will be consolidated to $n \log n$ after removing constants - let's prove it

$$f(n) = \sum_{leaves} (c_2 + c_1 \log n) + \sum_{internal\ nodes} (c_3 + c_1 \log n)$$

$$f(n) = \sum_{leaves} (c_2) + \sum_{leaves} (c_1 \log n) + \sum_{internal\ nodes} (c_3) + \sum_{internal\ nodes} (c_1 \log n)$$

$$f(n) = leaves * \cancel{c_2} + leaves * c_1 \log n + internal * \cancel{c_3} + internal * c_1 \log n$$

$$f(n) = leaves + internal + (leaves + internal) c_1 \log n$$

$$f(n) = n + n * \cancel{c_1} \log n$$

$f(n)$ is $O(n \log n)$