Ariana Aimani

Collaborators:
Sanam Aghdaey
Nick Nasirpour
Vasu Gupta

Question 1- COMP 251

We see that in the division method, the size of the hash table will be D but in the multiplication method, the size of the hash table will be $2^r = 2^d$. We also see that d <= 2^d when d >= 1, therefore the size of the hash table in the multiplication method will usually be equal to or greater than the size of the hash table with the division method. The load factor can be taken into account since the load factor is where n is the number of keys and the load factor of the division method will also usually be greater than the load factor of the multiplication method, if there is equal n.

More in depth in regards to load factor:

- the number of collisions in both hash functions are inversely proportional to the size of their hash tables or proportional to the load factor. (These are basically equivalent statements).
- Since 2^d will grow at a much faster rate than d, the load factor in the multiplication method will decrease much faster than the load factor of the division method for increasing d.
    - The multiplication method
    - and division method may cause a very similar number of collisions for a smaller d.
    - the multiplication method can cause fewer collisions for larger d. This is due to there being a larger amount of buckets to hash to for the same number of keys and the same domain.

There are three graphs presented below. Here, we keep w (domain size) and n (number of keys) constant but change the value of d (number of slots in the hash table). In this scenario we have w = 32 and m = 1000 but d is varied.

The important results to look at are:

- The number of conditions actually varies in a linear motion against m for both methods.

From this exercise, we see that the size of the domain will usually be much larger than the size of the hash table. Thus, both the multiplication and division hashing methods perform in a very similar fashion.

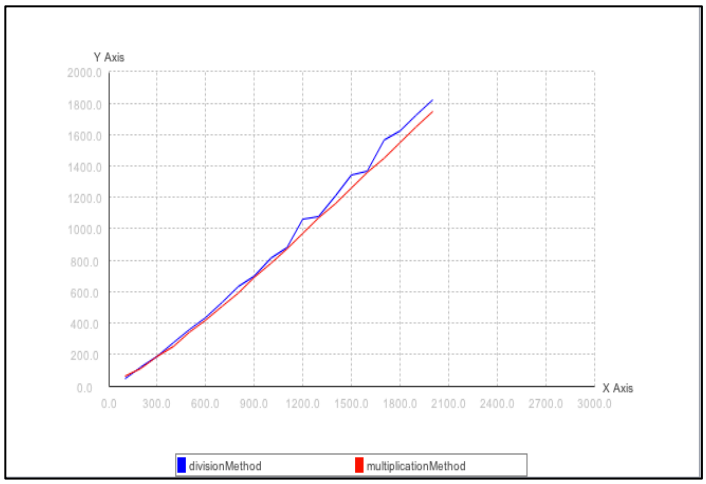


Fig A: w = 32, d =8, m = 1000

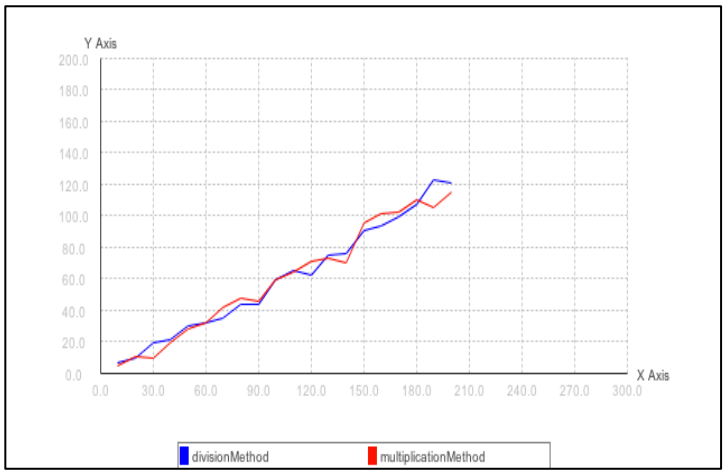


Fig B: w = 32, d = 8, m = 100

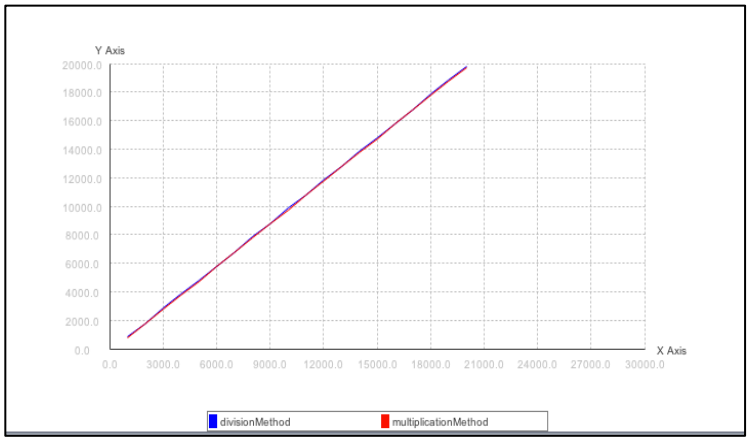
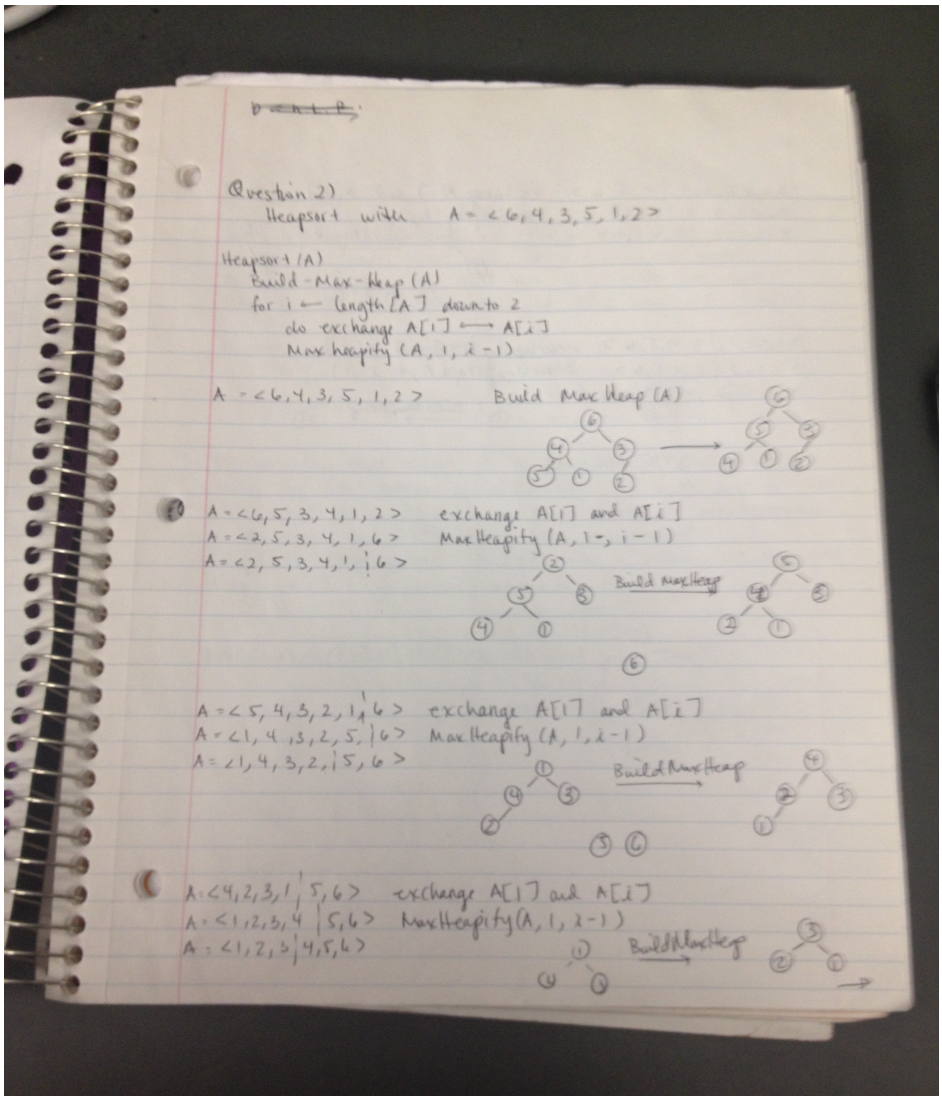


Fig C: w= 32 d =8 m =10000

Question 2



Question 2)
Heapsort with A = <6, 4, 3, 5, 1, 2>

Heapsort (A)
Build-Max-Heap (A)
for i ← length[A] down to 2
do exchange A[1] ⟷ A[i]
Max heapify (A, 1, i-1)

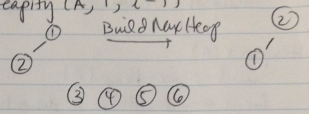
A = <6, 4, 3, 5, 1, 2> Build Max Heap (A)

A = <6, 5, 3, 4, 1, 2> exchange A[1] and A[i]
A = <2, 5, 3, 4, 1, 6> Max Heapify (A, 1, i-1)
A = <2, 5, 3, 4, 1 | 6>
Build Max Heap

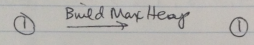
A = <5, 4, 3, 2, 1 | 6> exchange A[1] and A[i]
A = <1, 4, 3, 2, 5 | 6> Max Heapify (A, 1, i-1)
A = <1, 4, 3, 2, 5 | 6>
Build Max Heap

A = <4, 2, 3, 1 | 5, 6> exchange A[1] and A[i]
A = <1, 2, 3, 4 | 5, 6> Max Heapify (A, 1, i-1)
A = <1, 2, 3 | 4, 5, 6>
Build Max Heap

A = ⟨3, 2, 1, | 4, 5, 6⟩ exchange A[1] and A[i]
A = ⟨1, 2, 3, | 4, 5, 6⟩ MaxHeapify (A, 1, i-1)
A = ⟨1, 2, | 3, 4, 5, 6⟩

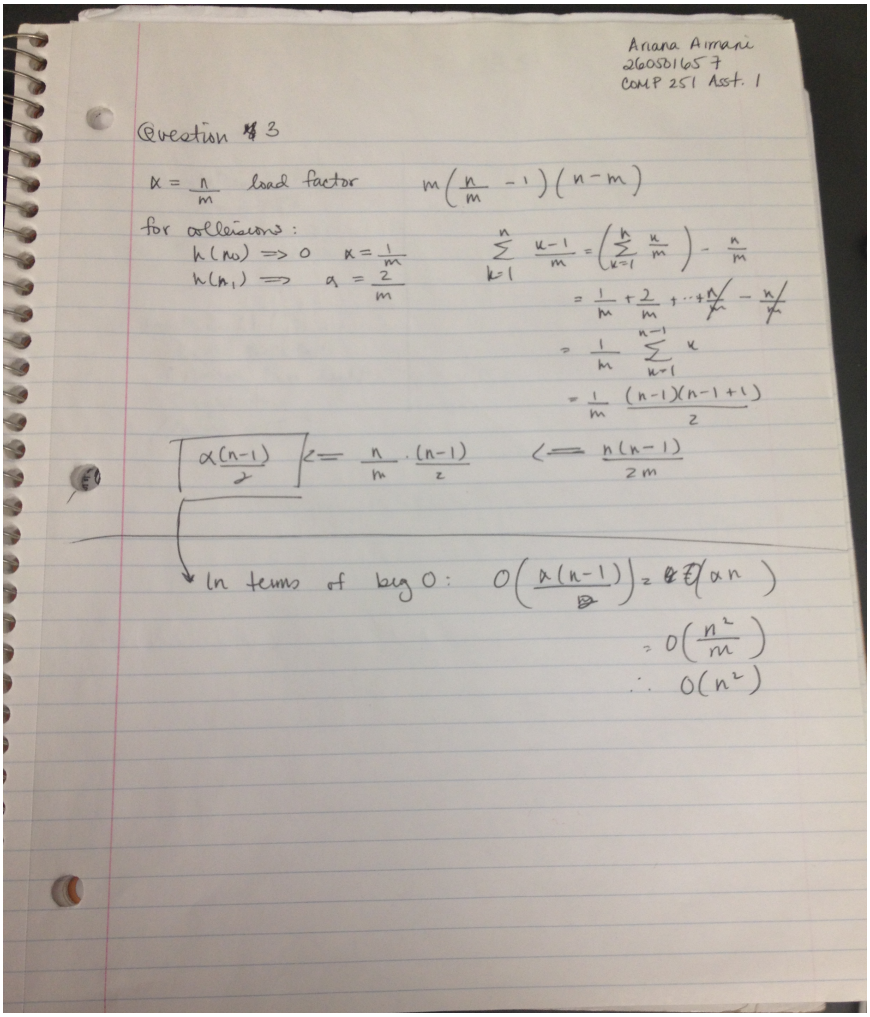


A = ⟨2, 1, | 3, 4, 5, 6⟩ exchange A[1] and A[i]
A = ⟨1, 2 | 3, 4, 5, 6⟩ MaxHeapify (A, 1, i-1)
A = ⟨1, | 2, 3, 4, 5, 6⟩

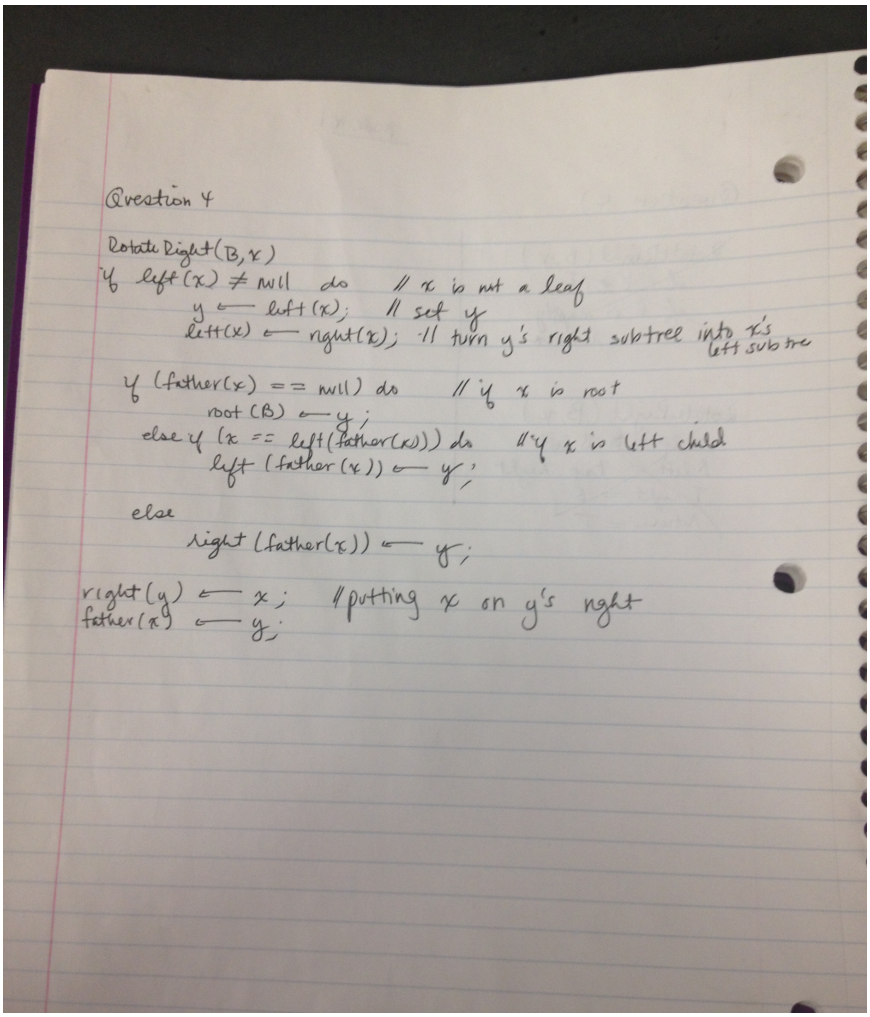


A = ⟨1, 2, 3, 4, 5, 6⟩ □

Question 3


Ariana Aimani
260501657
COMP 251 Asst. 1
Question #3
α = n/m load factor
for collisions:
h(x0) => 0 α = 1/m
h(x1) => α = 2/m
m(n/m - 1)(n - m)
In terms of big O:

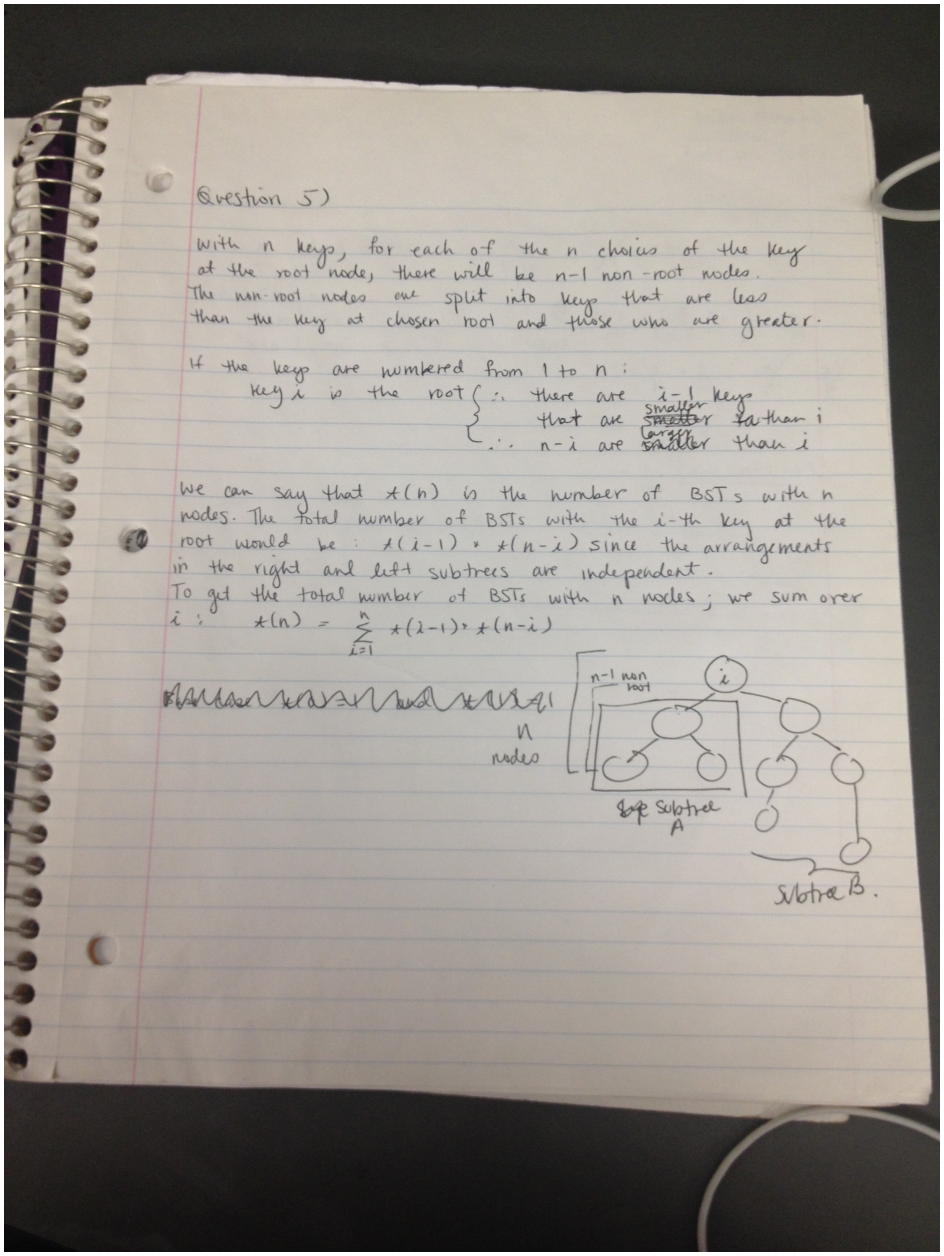
Question 4:



Question 4

```
RotateRight(B, x)
if left(x) ≠ null do    // x is not a leaf
    y ← left(x);    // set y
    left(x) ← right(x);  // turn y's right subtree into x's left subtree
if (father(x) == null) do    // if x is root
    root(B) ← y;
else if (x == left(father(x))) do    // if x is left child
    left(father(x)) ← y;
else
    right(father(x)) ← y;
right(y) ← x;    // putting x on y's right
father(x) ← y;
```

Question 5


Question 5)
With n keys, for each of the n choices of the key at the root node, there will be n-1 non-root nodes. The non-root nodes are split into keys that are less than the key at chosen root and those who are greater.
If the keys are numbered from 1 to n:
Key i is the root
∴ there are i-1 keys that are smaller than i
∴ n-i are greater than i
We can say that t(n) is the number of BSTs with n nodes. The total number of BSTs with the i-th key at the root would be: t(i-1) * t(n-i) since the arrangements in the right and left subtrees are independent.
To get the total number of BSTs with n nodes; we sum over i: t(n) = Σ t(i-1) * t(n-i)
n nodes
n-1 non root
i
Left Subtree A
Subtree B.