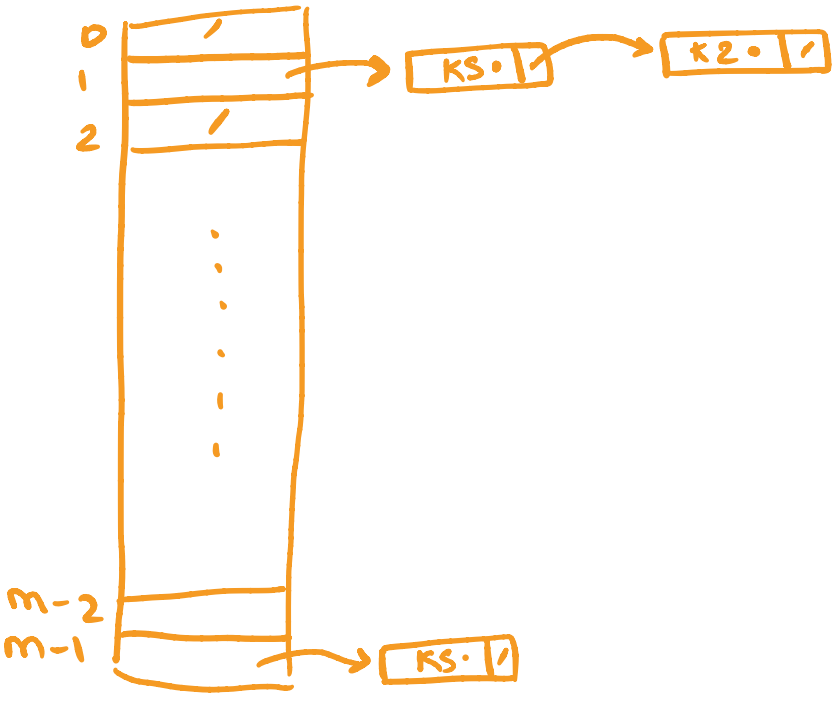
Chalk Talk

Chaining or Closed Addressing

Each slot of T stores a linked list of items that hash to this bucket

h(k1) = h(k4) = h(k6) = 2 h(k2) = h(k5) = 1 h(k3) = m-1



Worksheet

Worksheet Questions 1-2: One Solution

Worst case : all keys hash to same bucket

As long as $|U| > n(n-1)$ it is possible to pick n keys that hash same bucket

Worst case $\Theta(n)$

Solution in Worksheet

This solution assumes that new elements are inserted at the head of the chain. You could also insert at the tail.

Chalk Talk

Average-case Runtime of Search with Chaining

Simple Uniform Hashing Assumption

assume key is equally likely to hash to any bucket

consequence

expected no. of keys in each bucket is the same

$\frac{n}{m}$ → no. of items / slots

Load factor $\frac{n}{m} = \alpha$

Distribution over input:

We are equally likely to search for any key in U

Chalk Talk

Average-case Runtime of Search with Chaining

Running time = time to compute hash function + number of keys to compare

Part 1: assume k is not in the table

Compute $h(k)$ then traversed entire list in $T[h(k)]$

$E[\text{length of list in } h(k)] = \frac{n}{m} = \alpha$

$E[T] = 1 + \alpha$

$= \Theta(1 + \alpha)$

Part 2: assume k is in the table $\longrightarrow$ values in table $x_1, x_2, \ldots, x_n$

Search for x_i

Probability we select $x_i = P(x_i) = \frac{1}{n}$

time = 1 + # of elements we examine

How many items were inserted into T after $i = n - i$

How many of these $n-i$ items do we expect in each bucket? $\frac{n-i}{m}$

$$E[T] = 1 + \sum_{i=1}^{n} P(x_i) \cdot T(x_i)$$

$$= 1 + \frac{1}{n}\sum_{i=1}^{n}\left(\frac{n-i}{m} + 1\right)$$

applying hash function

comparisons with elements in my chain before me

comparison with x_i

$$= 1 + 1 + \frac{1}{n}\sum_{i=1}^{n}\left(\frac{n-i}{m}\right)$$

$$= 2 + \frac{1}{n}\sum_{i=1}^{n}\left(\frac{n}{m}\right) - \frac{1}{n}\sum_{i=1}^{n}\left(\frac{i}{m}\right)$$

$$= 2 + \frac{n^{\not x}}{\not n m} - \frac{1}{\not n}\left[\frac{\not n(n+1)}{2m}\right]$$

Chalk Talk

$$= 2 + \frac{2n}{2m} - \frac{n+1}{2m} = 2 + \frac{2n - n - 1}{2m}$$

$$= 2 + \frac{n-1}{2m}$$

$\alpha = \frac{n}{m} \Rightarrow n = \frac{n}{\alpha}$

$$= 2 + \frac{(n-1)\alpha}{2n}$$

$$= 2 + \frac{\alpha}{2} - \frac{\alpha}{2n}$$

$\frac{\alpha}{2n}$ → AS $n \to \infty$, tends to 0

$$= \Theta(1+\alpha)$$

Final conclusion : Search $\Theta(1+\alpha)$ average case

Open Addressing

Keep all records IN the table

When the first bucket is already taken, go to another bucket

needs to be a value from 0 to m-1

buckets in hash table

h(k) row h(k,i) i = 0, 1, 2, ... m-1

Probe sequence: The sequence of buckets where we try to store k. (h(k,0), h(k,1), h(k,2) ..., h(k,m-1))

Want (require?) the probe sequence

to be a permutation of 0, 1, 2, ..., m-1

hit every bucket

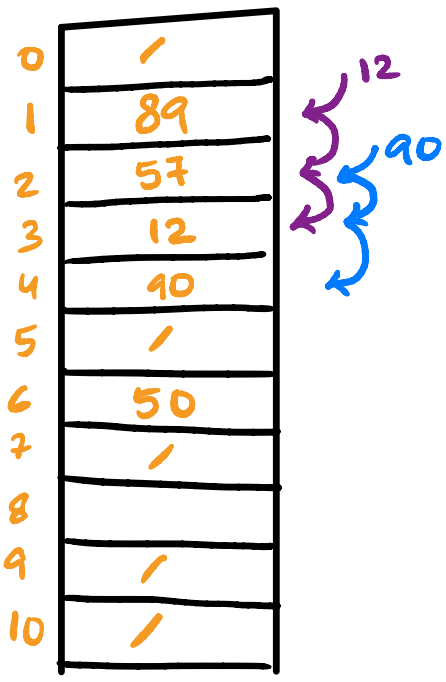
Worksheet

Q3 & Q4: Open Addressing Example

m = 11

$h(k,i) = (h'(k) + i) \bmod m \quad i = 0, 1, 2, \ldots, m-1$

$h'(k) = k \bmod m$ (home bucket)



Search until we get a null

	home	actual	comparison keys
IN(57)	2	2	1
IN(89)	1	1	1
IN(50)	6	6	1
IN(12)	1	3	3
IN(90)	2	4	3
Search(90)	2	4	3
Search(16)	5	x	1
search(12)	1	3	3
Search(34)	1	x	5

Problem with searching 34?
If all buckets are filled, it has to go and compare from position 1 to m-1.
- CLUSTERING

Zoom Poll

Challenges with Linear Probing

Primary clustering

clusters in the table of filled slots. Once we hash to anywhere along the cluster, we extend it

Q: Would using a larger step than 1 help?

Nah.

Q. How about a random step size?

X Nope!

We can't find stuff easily then

Quadratic Probing

offset

$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$ $i = 0, 1, 2, \ldots,$ ~~m-1~~ m - 1

$h'(k) = k \bmod m$

Challenge? Still need a permutation of $(0, 1, 2, \ldots, m-1)$ pick c_1, c_2, m carefully

two keys that have the same home bucket have the same probe sequence

Double Hashing

home bucket — offset — step

$h(k,i) = (h_1(k) + h_2(k)i) \bmod m$ $\quad i = 0, 1, 2, \ldots, m-1$

$h_1(k) = k \bmod m$ — home bucket

$h_2(k)$ — another hash function

step size for 1 key is different than the step size for another even though they have the same home bucket

Constraints on hash functions?

Want $h(k,i)$ for $i = 0, 1, 2, \ldots, m-1$ to be a permutation of $(0, 1, 2, \ldots, m-1)$

We need $h_2(k)$ and m to be relatively prime

One technique is to make m a power of 2.
Make $h_2(k)$ be odd.

What about deleting in open-addressing?

delete(57) $h(57) = 2$

search(43) $h(43) = 10$

43 is not in table

search(90) $h(90) = 2$

90 is not in the table

But it is !!!

Put a delete marker (tombstone), and keep searching till node is None, not a tombstone.

$\alpha \leq 1$ now

Insert(68) $h(68) = 2$.

So replace tombstone

Augmenting Data Structures

CSC263 Week 5 Friday Lecture

Augmented data structure = existing data structure modified to store additional information and/or perform additional operations

1. Choose data structure to augment
2. Determine the additional information
3. Check that additional information can be maintained during each original operation
4. Implement new operations

Zoom Chat

Example: Ordered Sets

rank 1

{144, 20, 100, 30, 17, 150, 55}

Rank: position within an ordered list -- typically from lowest to highest

RANK(k): return rank of key k

SELECT(r): return the key with rank r

also support INSERT, DELETE and SEARCH

Rank(17) = 1

Rank(100) = 5

Select(1) = 17

Select(4) = 55

Select(3) = 30

Select(20) = ERROR

Chalk Talk

Approach 1: AVL tree without modification

Queries: Carry out inorder traversal of tree & keep track of # of nodes visited until we reach either desired key or desired rank

Q: What will be the time for the new queries?

A: $\Theta(n)$ worst case

Q: Will the other operations (SEARCH/INSERT/DELETE) take any longer?

A: No, because we have no new information to update

Problem: Query time. Can we do better?

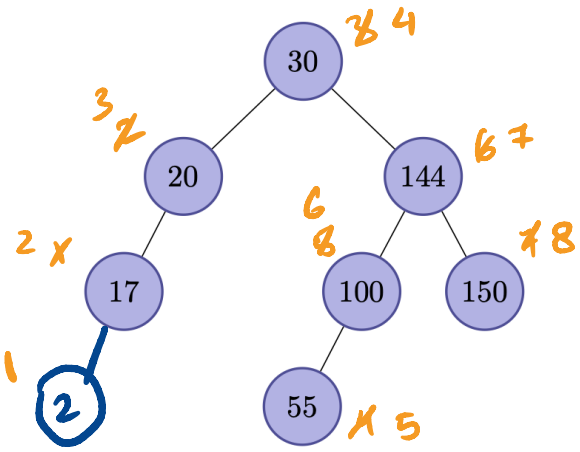
↓ for RANK/SELECT

$\Theta(n)$ too slow!

Chalk Talk

Approach 2

Augment AVL tree so each node has an additional field rank[x] that stores its rank in the tree.



Insert(2)

Chalk Talk

Approach 2: AVL storing rank and key

Q: What will be the time for the new queries?

A:

$\Theta(\log n)$

Q: Will the other operations (SEARCH/INSERT/DELETE) take any longer?

A:

SEARCH: $\Theta(\log n)$ ↓

INSERT/DELETE: $\Theta(n)$

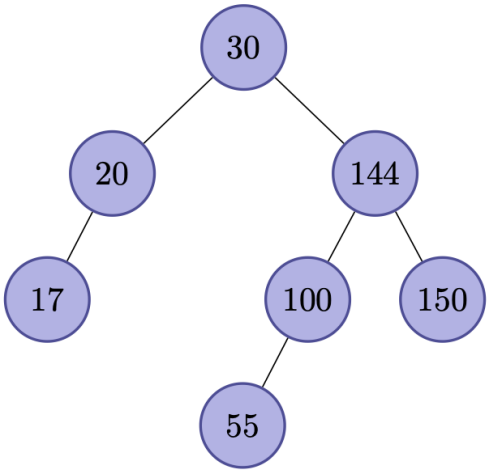
Problem: Query time. Can we do better?

↳ for Insert + Delete are now too slow

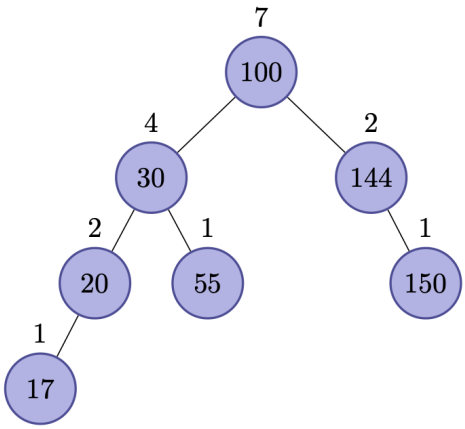
Chalk Talk

Approach 3: store subtree size

Augment an AVL tree so that each node n has an additional field n.size() that stores the number of keys in the subtree rooted at n (including n itself)



Zoom chat

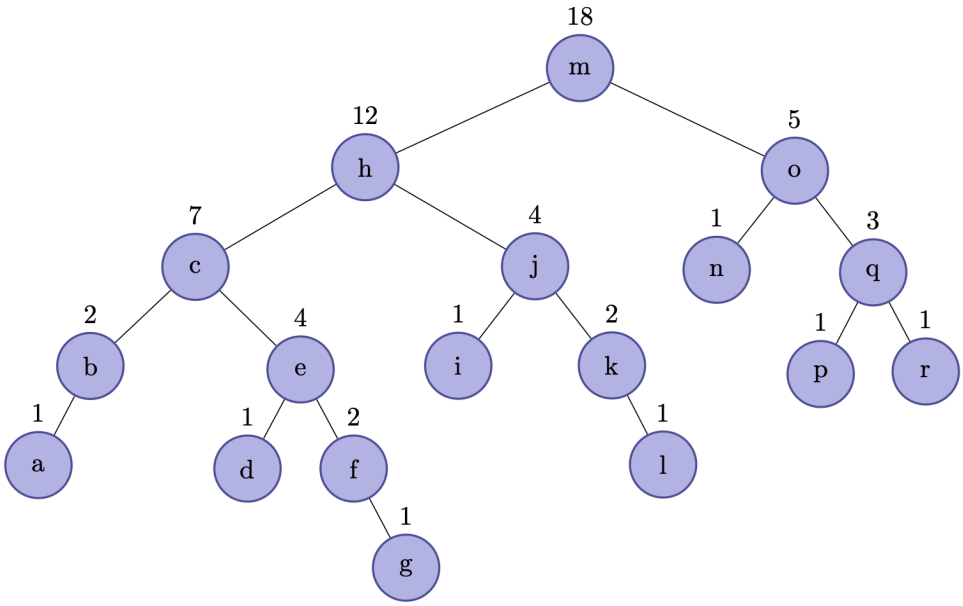


Rank(100) = 5

Rank(30) = 3

Rank(20) = 2

Zoom chat



Chalk Talk

Approach 3: store subtree size

Q: How do we quickly find the rank of the root of the tree?

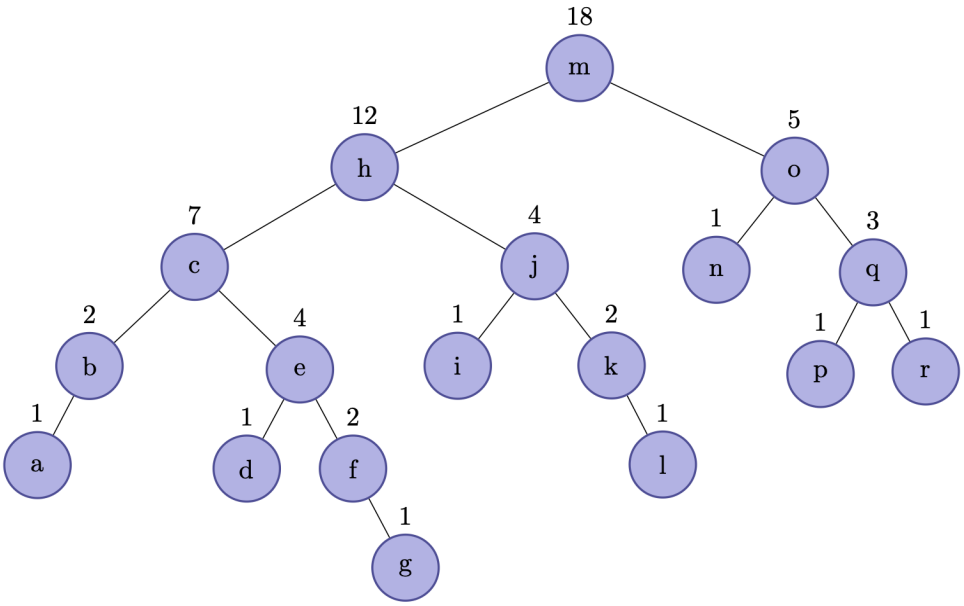
A: root.left.size + 1

Q: How do we quickly find the local (i.e. relative) rank of a node m in the tree rooted at m?

A: m.left.size + 1

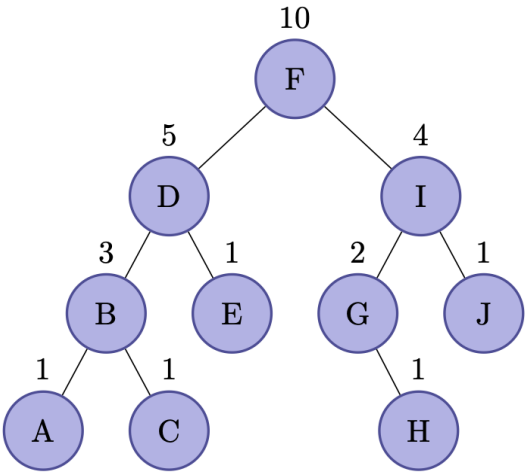
Zoom chat

Identify the Local (Relative) Rank



Worksheet

Worksheet Questions 1 & 2: implement SELECT



Worksheet

Worksheet Questions 1 & 2

```
SELECT(T, k):
    if T.left != NULL:
        local_rank = T.leftsize + 1
    else:
        local_rank = 1
    if local_rank = k:
            return T.root
    if k < local_rank:
            return SELECT(T.left, k)
    else:
        return SELECT(T.right, k - local_rank)
```

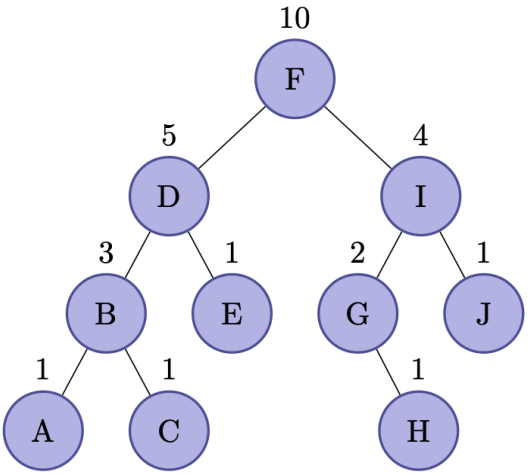
Chalk Talk

Implement RANK

Idea – Perform SEARCH on k and keep track of the current rank. Each time you go down a level by making a right turn, add the size of the subtrees to the left that you skipped. This is the local rank of the parent p you just left in the subtree rooted at p. Keep track of accumulating ranks.

Worksheet

Worksheet Questions 3 & 4 RANK



rank(E) → 5

rank(I) → 9

((5+1) + (2+1))

Chalk Talk

Q: How do we calculate the local rank of a node m in the tree rooted at m?
A: m.left.size + 1

Q: When we recurse on the left subtree what do we add?
A: 0

Q: When we recurse on the right subtree what do we add?
A: parent local rank

Q: When we find x, how do we determine its true rank?
A: current rank + local rank

```
RANK(T, k):
        return TREE-RANK(T.root, k, 0)

TREE-RANK(root, k, current_rank):
        if root is NIL: # if k not in S
            pass
        if root.left == null:
            local_rank = 1
        else:
            local_rank = root.left.size + 1

        if k < root.item.key:
            # moving left so don't deduct anything
            return TREE-RANK(root.left, k, current_rank)
        else if k > root.item.key:
            # moving right so add items skipped
            return TREE-RANK(root.right, k,
                             current_rank + local_rank)

        else: # x.key == root.item.key
            # since we have found the item, return
            # cumulative rank

            return current_rank + local_rank
```

Chalk Talk

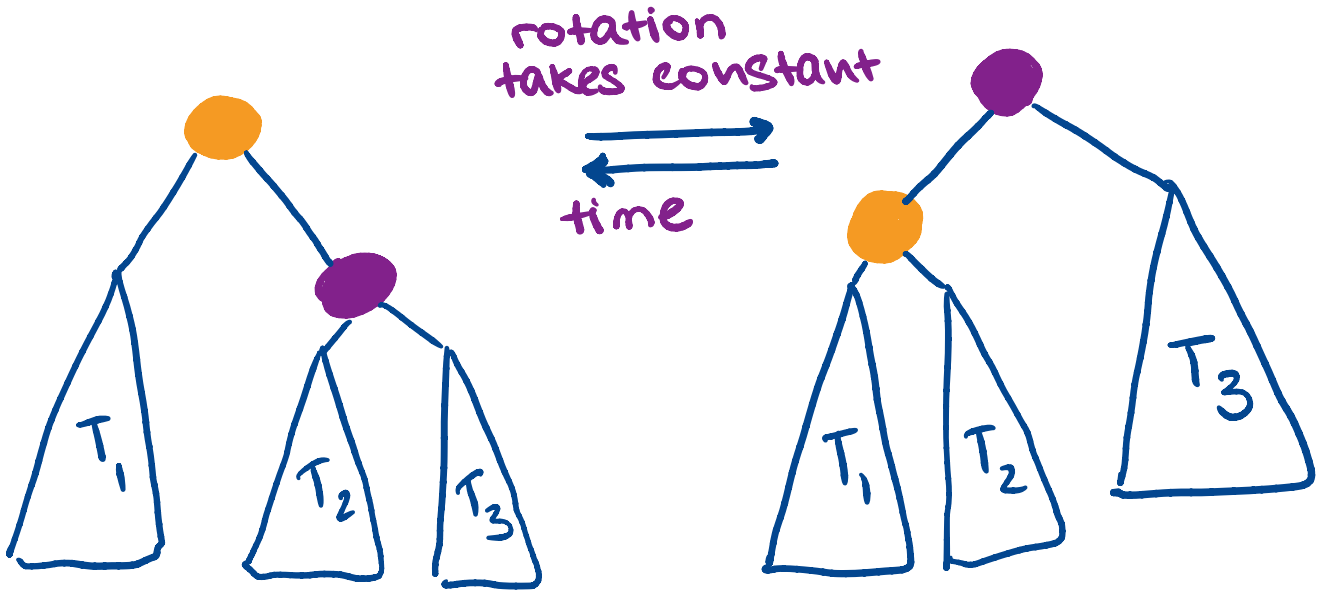
Q: Time for the new queries?

A: $\Theta(\log n)$

Q: What about updates of new information in old operations?

INSERT → add as we go down

DELETE → subtract 1 on every node of path to leaf from root



Final Reminders

Things to take away

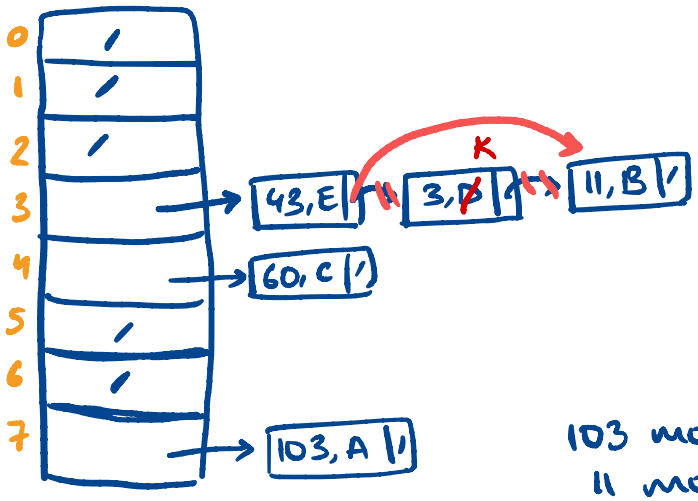
Augmenting an AVL tree with size of subtree rooted at m, allowed us to efficiently implement RANK and SELECT.

General strategy:

- Use known data-structure with extra information
- Extra information must be maintained on original operations

1. Draw a hash table with 8 slots. Using the hash function $h(k) = k \bmod 8$, do the following operations in order. Notice that each tuple contains a key and a value. Often we don't show the value, but this time put them into your table. As you do each operation, count the number of times you examine a key. There are some different options about how to order the chains depending on what you assume about whether insert and delete have already searched for the item. As a group discuss how you are defining the operations and then be consistent with your approach.

operation	keys examined
insert(103, A)	
insert(11, B)	
insert(60, C)	
insert(3, D)	
search(3)	
insert(43, E)	
search(103)	
search (43)	
search (83)	
search (5)	
insert (3, K)	already there
delete (3)	



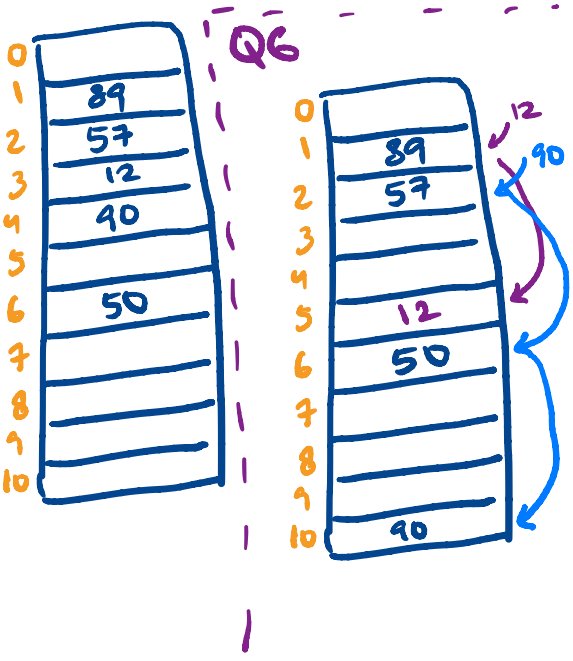
103 mod 8 = 7
11 mod 8 = 3
60 mod 8 = 4
43 mod 8 = 3

2. Work out the worst-case complexity for insert, delete, and search in a hash-table using chaining and enter them in to the breakout-room Google doc at https://tinyurl.com/breakouts-263

3. For this question, use a hashtable with 11 slots that is implementing open addressing using the hash function $h(k, i) = (h(k) + i) \bmod m$ for $i = 0, 1, 2, , m - 1$. Use $h'(k) = k \bmod m$. Start with an empty table and draw the values in the table as you execute the following operations.

m=11

operation	home bucket	actual bucket	keys examined
insert(57)	2	2	1
insert(89)	1	1	1
insert(50)	6	6	1
insert(12)	1	3	3
insert(90)	2	4	3
search(90)	2	4	3
search(16)	5	X	1
search(12)	1	X	4
search(34)	1	X	4



4. Discuss with your group the problem with searching for 34? At this point bucket 0 is empty, but if we were to insert one item into it and then try to insert a second, how many element comparisons would the second insert take? Clustering

empty then wrong

5. Delete item with key 57 from the hashtable from question 3. Now search for 90. What is the problem?

6. (Homework or optional problem) Redo the operations from Question 3, using a quadratic probing. Use $h(k, i) = (h(k) + i + 3i^2) \bmod m$ to determine the probe sequence.

1. Draw a hash table with 8 slots. Using the hash function $h(k) = k \bmod 8$, do the following operations in order. Notice that each tuple contains a key and a value. Often we don't show the value, but this time put them into your table. As you do each operation, count the number of times you examine a key. There are some different options about how to order the chains depending on what you assume about whether insert and delete have already searched for the item. As a group discuss how you are defining the operations and then be consistent with your approach.

   SOLUTION: Taken up in lecture and posted in annotated lecture slides.

2. Work out the worst-case complexity for insert, delete, and search in a hash-table using chaining and enter them in to the breakout-room Google doc at https://tinyurl.com/breakouts-263

   SOLUTION: In the worst-case all of the n inserted keys in a hash-table using chaining could map to the same bucket. This would give us a linked list in that bucket and the performance would be worse than just using a linked-list because we would have to compute the hash function first plus do whatever we would otherwise do for an unsorted linked list.

   This means that search is $\Theta(n)$. The additional time for insert and delete is constant but they both require a search as part of their implementation.

3. For this question, use a hashtable with 11 slots that is implementing open addressing using the hash function $h(k, i) = (h(k) + i) \bmod m$ for $i = 0, 1, 2, , m - 1$. Use $h'(k) = k \bmod m$. Start with an empty table and draw the values in the table as you execute the following operations.

operation	**home bucket**	**actual bucket**	**keys examined**
insert(57)			
insert(89)			
insert(50)			
insert(12)			
insert(90)			
search(90)			
search(16)			
search(12)			
search(34)			

4. Discuss with your group the problem with searching for 34? At this point bucket 0 is empty, but if we were to insert one item into it and then try to insert a second, how many element comparisons would the second insert take?

   SOLUTION: 34 initially hashes to bucket 1, but that's filled with 89 and so we start down the probe sequence and end up having the examine a lot of keys before we find an empty slot. The problem is that the keys in the table start to cluster together. Once we hash to anywhere on an existing cluster of filled buckets, we extend the cluster.

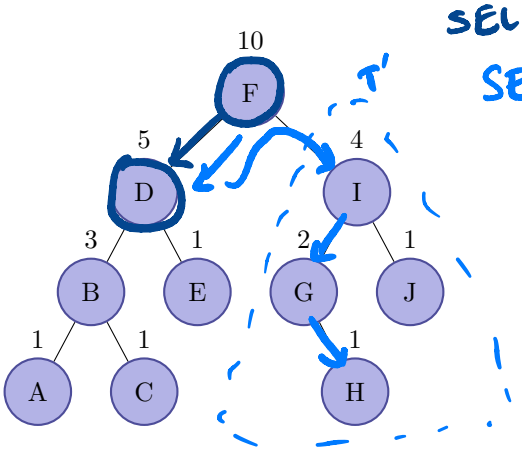
5. Delete item with key 57 from the hashtable from question 3. Now search for 90. What is the problem?

   SOLUTION: The problem is that when we inserted 90 we travelled along the probe sequence past bucket 2 because bucket 2 was occupied by 57. But now that 57 has been deleted, when we search for 90 and look in bucket 2, we find that there is no record there and we might incorrectly conclude that 90 is not in the table.

6. (Homework or optional problem) Redo the operations from Question 3, using a quadratic probing. Use $h(k, i) = (h(k) + i + 3i^2) \bmod m$ to determine the probe sequence.

   SOLUTION: Left to the student to work out.

1. The following BST has been augmented so that each node stores the size of the subtree rooted at that node, including itself. Trace the operations SELECT(T, 5) and SELECT(T, 7) on this tree T. When you have finished, indicate this in the Google doc at `http://tinyurl.com/breakouts-263` by putting in the return values for the two operations. Make sure that everyone in your group understands the algorithm not only the final value.



SELECT(T,5) = D

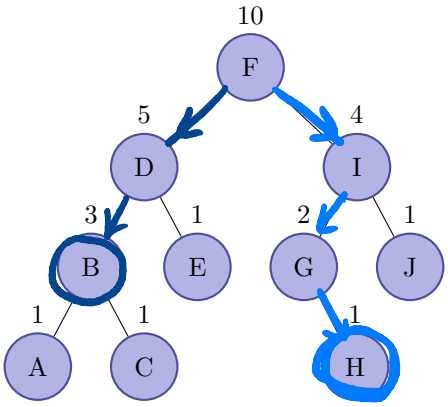
SELECT(T,7):
local rank = 6
↓
select(T.right, 7-6)

= H

2. Write pseudocode for the SELECT operation. Leave the special case of k not in tree T until the end (to complete only if you have extra time in the breakout session.) When you are finished the pseudocode for the cases of k in the tree, indicate this in the Google doc and then keep discussing how to handle k not in tree T.

```
SELECT(T, k):
```

3. On another copy of the tree from Question 1, trace RANK(T, B) and RANK(T, H) to make sure you understand the algorithm for implementing RANK.



Rank(T,B):
B.left + 1
= 2

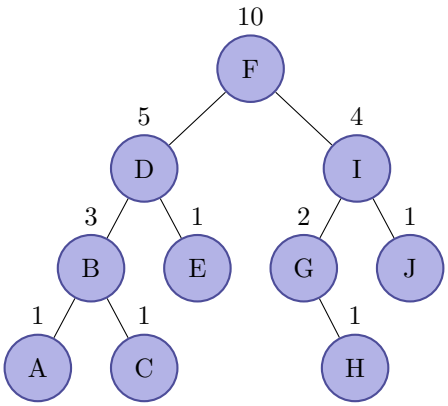
Rank(T,H)
4+2+1 = 7

4. Complete the pseudocode for RANK(T, k).

```
RANK(T, k):
    return RANK(T.root, k, 0)

TREE-RANK(root, k, current_rank):
    # determine my local_rank in the tree rooted at root
    if root.left == null:
        local_rank = 1
    else:
        local_rank =
```

1. The following BST has been augmented so that each node stores the size of the subtree rooted at that node, including itself. Trace the operations SELECT(T, 5) and SELECT(T, 7) on this tree T. When you have finished, indicate this in the Google doc at `http://tinyurl.com/breakouts-263` by putting in the return values for the two operations. Make sure that everyone in your group understands the algorithm not only the final value.



2. Write pseudocode for the SELECT operation. Leave the special case of k not in tree T until the end (to complete only if you have extra time in the breakout session.) When you are finished the pseudocode for the cases of k in the tree, indicate this in the Google doc and then keep discussing how to handle k not in tree T.

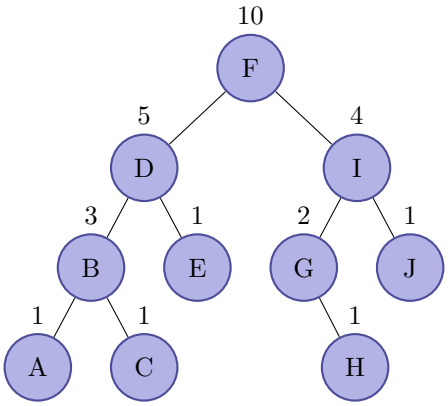
   SOLUTION:

```
SELECT(T, k):
    # logic to handle degenerate case of rank k not in T
    if k > T.size:
        report rank k is not in T

    # calculate rank of root in subtree rooted at T
    local_rank = T.left.size + 1

    if local_rank == k:
        return T.root
    if k < local_rank:
        return SELECT(T.left, k)
    else:
        return SELECT(T.right, k - local_rank)
```

3. On another copy of the tree from Question 1, trace RANK(T, B) and RANK(T, H) to make sure you understand the algorithm for implementing RANK.



4. Complete the pseudocode for RANK(T, k).

   SOLUTION:

```
RANK(T, k):
    return TREE-RANK(T.root, k, 0)

TREE-RANK(root, k, current_rank):
    # determine my local_rank in the tree rooted at root
    if root.left == null:
        local_rank = 1
    else:
        local_rank =  root.left.size + 1

    if root is NIL:     #k not in T
       raise or report an error   k not in T
    else if k < root.item.key:
         # moving left
         return TREE-RANK(root.left, k, current_rank)
    else if k > root.item.key:
        # moving right so add for items skipped
        return TREE-RANK(root.right, k, current_rank + local_rank)
    else:   # x.key == root.item.key
         # since we have found the item return our accumulated current rank
         return current_rank + local_rank
```

WEEK 5 PREP

1. Quicksort

[5, 2, 1, 6, 4]

(a) p = 5
smaller = [5, 6]
bigger = [2, 1, 4]

(b) p = 2
smaller = [1]
bigger = [4]

2. Probability 2 elements are compared in randomised quicksort $= \frac{2}{(d+2)}$ where d is no. of integers b/w x and y

3. [2, 9, 5, 4, 6]

$P(2, 9 \text{ compared}) = \frac{2}{3+2} = 0.4$

$P(2 \text{ \& } 4 \text{ compared}) = \frac{2}{0+2} = \frac{2}{2} = 1$

Hashing and Quicksort

Finishing up some hashing odds and ends

Announcements

- PS1 to be returned today (Tuesday)
- Solutions posted for Q1 and Q4d
- Do the Practice Test

Test 1

- This Friday – Feb 18th
- Everything from first 3 weeks of the course
- Aid Sheet Required - must be submitted with your test answers
- No other aids - not notes, text, people or websites
- 50 ~~hours~~ mins to write within a 2-hour window
- On MarkUs
- You can ask questions during the test through Piazza private messages
- Typed or hand-written
- Submission: type restricted to PDF or photo. Please name your files properly if you can
- Need to start before 11:10:00 AM.
- DO THE PRACTICE TEST

Hash Functions

Finding a good hash function appears essential to the assumption of simple uniform hashing

What do we want in a good hash function?

- h(x) effecient to compute
- h(x) "spread" out value (not cluster them)
- h(x) depend on every part of the key
  - ↳ even for compler object

In practice, getting all 3 is difficult

Chalk Talk

Step 1: from string or key to natural number k

"key" k → 107 e → 101 y → 121

⇝ 107 + 101 + 121 = 329

"yek" → 329

so can't just add them up

Instead use each position as a digit in base 128

"key" = $107(128)^2 + 101(128) + 121(128)$

= ~~~

But too much space!

Chalk Talk

Step 2: from natural number k to m

- Division Method $h(k) = k \bmod m$

  usually avoid m that is a power of 2 since that is equivalent to taking lower-order bits of k

- Multiplication Method $0 < A < 1$

  step 1 multiply k by A

  step 2 keep only fractional part

  step 3 multiply by m

  step 4 take floor

- Lots of Heuristics (see textbook)

$$0 \le \lfloor m(kA - \lfloor kA \rfloor) \rfloor < m$$

$$0, 1, 2, \ldots, m-1$$

Chalk Talk

Big Picture: Hashing vs. Balanced Tree

If we make the table an appropriate size, we control α.

Hashing is $\Theta(1)$ average case

So why would we ever use a tree for a dictionary?

1. If we can't tolerate worst case for hashing
2. If we need a list/set of all the keys

Chalk Talk

Quicksort: Worst Case – Upper bound

```python
def quickSort(array):
    if len(array) < 2:
        return array[:]  # makes a copy
    else:
        pivot = array[0]
        smaller, bigger = partition(array[1:], pivot)
        smaller = quickSort(smaller)
        bigger = quickSort(bigger)
        return smaller + [pivot] + bigger

def partition(array, pivot):
    smaller = []
    bigger = []
    for item in array:
        if item <= pivot:
            smaller.append(item)
        else:
            bigger.append(item)
    return smaller, bigger
```

Count comparisons

- each element of S is pivot at most once
- at most all other elements are compared to the pivot
- So every pair is compared at most once

$\binom{n}{2}$ pairs if $|S| = n$

$T(n) \leq \binom{n}{2} = O(n^2)$

Ω for a particular family of inputs

Chalk Talk

Quicksort: Worst Case – Lower Bound

Let C(n) denote # of comparisons on input [n, n-1, n-2, ..., 2, 1]

```
1   def quickSort(array):
2       if len(array) < 2:
3           return array[:]  # makes a copy
4       else:
5           pivot = array[0]
6           smaller, bigger = partition(array[1:], pivot)
7           smaller = quickSort(smaller)
8           bigger = quickSort(bigger)
9           return smaller + [pivot] + bigger
10
11  def partition(array, pivot):
12      smaller = []
13      bigger = []
14      for item in array:
15          if item <= pivot:
16              smaller.append(item)
17          else:
18              bigger.append(item)
19      return smaller, bigger
```

In line 6, pivot n gets compared in partition helper function to all other n-1 values yields

smaller = [n-1, n-2, n-3, ..., 2, 1]

bigger = []

All other comparisons happen in recursive call quicksort(smaller) which takes C(n-1) comparisons by our definition of C.

Hence C(n) satisfies recurrence

$C(n) = n-1 + C(n-1) \quad n > 1$

$C(n) = (n-1) + (n-2) + (n-3) + \ldots + 2 + 1$

$C(1) = 0$

$= \frac{n(n-1)}{2}$

$\Omega(n^2)$ $\therefore T(n)$ is in $\Theta(n^2)$

Chalk Talk

Quicksort: Average Case

Inputs = all permutations of $[1, 2, 3, \ldots, n-1, n]$

Assume uniform distribution.

```
1   def quickSort(array):
2       if len(array) < 2:
3           return array[:]  # makes a copy
4       else:
5           pivot = array[0]
6           smaller, bigger = partition(array[1:], pivot)
7           smaller = quickSort(smaller)
8           bigger = quickSort(bigger)
9           return smaller + [pivot] + bigger
10
11  def partition(array, pivot):
12      smaller = []
13      bigger = []
14      for item in array:
15          if item <= pivot:
16              smaller.append(item)
17          else:
18              bigger.append(item)
19      return smaller, bigger
```

T = random variable counting # comparisons.

We want $E[T]$, we define an indicator random variable

$i, j \in \{1, 2, \ldots, n\}$ $i < j$

$$X_{ij} = \begin{cases} 1 & \text{if } i \text{ and } j \text{ are compared to each other} \\ 0 & \text{otherwise} \end{cases}$$

$$T = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

$$E[T] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]$$

$E[X_{ij}]$ → probability that i is compared to j

$[\ldots\ i\ \ldots\ j\ \ldots]$

if we pick a value from here as a pivot before we pick i or j as a pivot. Then i and j will never be compared

$\{i, i+1, i+2, \ldots, j-1, j\}$ we are equally likely to pick any 1 of these values as a pivot (uniform distribution)

$j - i + 1$ values

$$E[T] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

↳ probability that i and j will be compared

Let $k = j - i$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-1} \frac{2}{k+1}$$

to show $O(n \log n)$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$$

↳ added a term; ↳ decreased denominator

Chalk Talk

$$= 2(n-1) \sum_{k=1}^{n} \frac{1}{k}$$

↳ harmonic series $\sim O(\log n)$

do again to show $\Theta(n \log n)$

$$E[T] = 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n-1} \frac{1}{k+1}$$

when $i=1$ $k = 1, 2, 3, \ldots, n-3, n-2, n-1$

$i=2$ $k = 1, 2, 3, \ldots, n-3, n-2$

$i=3$ $k = 1, 2, 3, \ldots, n-3$

$i = n-2$ $k = 1, 2$ ($n-3$ of these)

$i = n-1$ $k = 1$ ($n-2$ of these)

($n-1$ of these)

generally $n-k$ copies of $\frac{1}{k+1}$

$$= 2 \sum_{k=1}^{n-1} \frac{1}{k+1} (n-k)$$

$$= 2 \sum_{k=1}^{n-1} \frac{n - k - 1 + 1}{k+1} = 2 \sum_{k=1}^{n-1} \frac{n+1}{k+1} - 2(n-1)$$

$$= 2(n+1) \sum_{k=1}^{n-1} \frac{1}{k+1} - 2(n-1)$$

↳ harmonic series $\Theta(\log n)$

$$= \Theta(n \log n)$$

Zoom Poll Review

Classic Quicksort Summary

Quicksort performs well on random input

Quicksort performs poorly on nearly sorted input

Why? Cuz we are partitioning badly - not dividing & conquering

Big Question: Can we guarantee the "average" performance on "average"?

If we are picking pivot in the middle → it works slightly better for sorted input

Random pivot? Yes, it'll work great

Randomized Quicksort

- Average-case analysis depended on each permutation equally likely
- Instead of relying on distribution of inputs, randomize algorithm by picking random element as pivot
- Random behavior of algorithm on any fixed input is equivalent to fixed behavior of algorithm on uniformly random input

Expected worst-case time of randomized algorithm on every single input is θ (n log (n))

→ Average of worst case times

If we ran quicksort on any input multiple times and average the time it would perform $\Theta(n \log n)$

Takeaway

- Randomised algorithms are good when? Lots of good choices but hard to find a choice that is guaranteed to be good.

1. Given two unsorted arrays of integers, one of length n, the other of length m, determine whether the first array is a subset of the second one using time at most $O(n+m)$.

   (a) First give an algorithm that will work under the assumption that each array contains no duplicate elements.

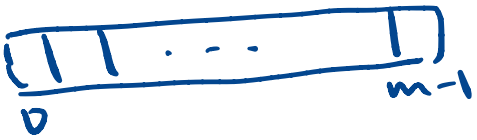
   (b) Explain how you to could adapt your approach to work with arrays containing duplicate elements. In this case the algorithm should compute that $[1,3,4,1]$ is a subset of $[3,4,1,7,1]$ but is not a subset of $[4,1,3]$.

2. Many programming contests include problems that require the competitors to implement data-structures. Here is a problem from the Canadian Computing Competition from 2007 that involves hashing

   https://dmoj.ca/problem/cco07p2

   If you are really keen, you can submit your code online but for the purposes of this course, I recommend you just think about the algorithm that you will use. Then later (perhaps in late April after all your final assessments are done) you can play more with the code.

(a)



1. Given two unsorted arrays of integers, one of length n, the other of length m, determine whether the first array is a subset of the second one using time at most $O(n+m)$.

   (a) First give an algorithm that will work under the assumption that each array contains no duplicate elements.
   SOLUTION: Hash the second array and then hash the first array and check if every element of the first array creates a collision when hashed.

   (b) Explain how you to could adapt your approach to work with arrays containing duplicate elements. In this case the algorithm should compute that $[1,3,4,1]$ is a subset of $[3,4,1,7,1]$ but is not a subset of $[4,1,3]$.
   SOLUTION: To handle duplicates of the same element we can associate a counter with each hashed element of the second array and decrement it every time we encounter that element in the first array.

2. Many programming contests include problems that require the competitors to implement data-structures. Here is a problem from the Canadian Computing Competition from 2007 that involves hashing

   https://dmoj.ca/problem/cco07p2

   If you are really keen, you can submit your code online but for the purposes of this course, I recommend you just think about the algorithm that you will use. Then later (perhaps in late April after all your final assessments are done) you can play more with the code.

   SOLUTION: You can read an excellent discussion of a solution for this problem in a clever data structures textbook written by Dan Zingaro (a CS teaching stream faculty member at UTM). Dan's book is available at https://nostarch.com/algorithmic-thinking. You can download the chapter on hashing for free and then if you love it, you will have to pay for the other chapters.