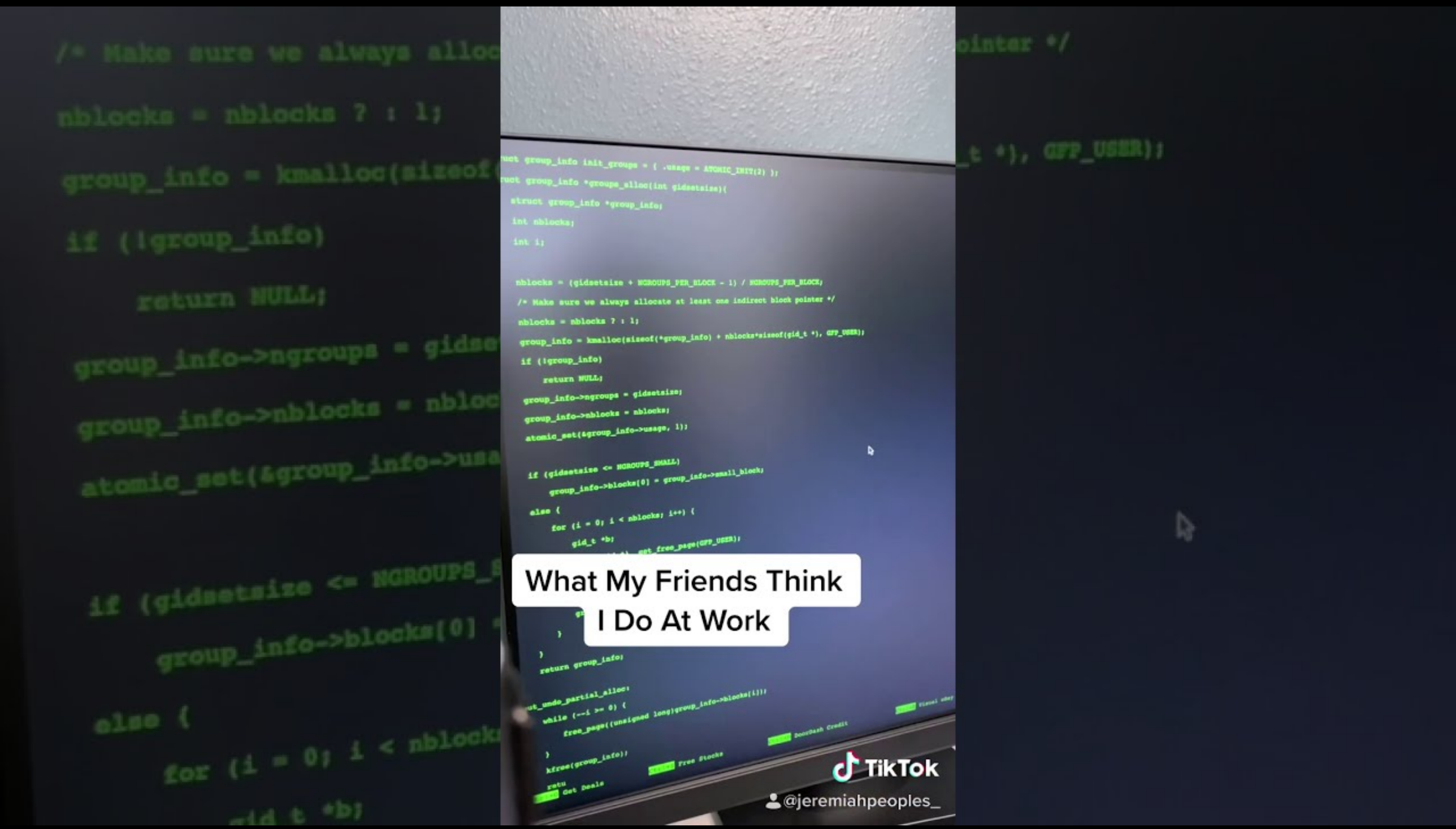
CSC 148

Week 10

Sadia ‘Rain’ Sharmin

Classes begin 10 minutes after the hour

This Week's Feature: Artificial Intelligence Researcher *Laura Caccia*



And a short video just for laughs

BST Efficiency

Efficiency

Multiset ADT

Let's imagine we are implementing the Multiset ADT (a set that allows duplicates; it's not a list -- doesn't care about ordering).

We will discuss the efficiency of this Multiset's search, insert and delete operations if we implement it using a Python list, a general tree, or a BST.

Efficiency

Multiset ADT

What if we keep track of all our multiset elements within a sorted Python list?

For a sorted list with *n* items…

insert and delete can be slow, if inserting/removing from the *front* of the list – O(*n*) in the worst case

Efficiency

Multiset ADT

What if we keep track of all our multiset elements within a sorted Python list?

For a sorted list with *n* items...

insert and delete can be slow, if inserting/removing from the *front* of the list – O(*n*) in the worst case

how about search?

Efficiency

Searching a list

In a sorted list: What strategy would you use to get to the value (if it exists) in a lot less steps than linear search?

Efficiency

Searching a list

In a sorted list: What strategy would you use to get to the value (if it exists) in a lot less steps than linear search?

Binary Search! Look at the middle of the list, and eliminate half of the list depending on if the list item is less than or greater than what you're searching for. (If it's equal to, you've found it)

Efficiency

Searching a list

Binary Search! Look at the middle of the list, and eliminate half of the list depending on if the list item is less than or greater than what you're searching for. (If it's equal to, you've found it)

So, we keep dividing in half the total elements we may have to look at. In the worst-case we will divide in half over and over, until we just have one element left.

How many times can we divide n by 2 until we have one item left?

Efficiency

log n

Key insight: the number of times I repeatedly divide n in half, before we are down to 1 element, is the same as the number of times I double 1 before I reach (or exceed) n.

$\log_2 n$, often known in CS as log n

Aside: logarithms

Recall:

$\log_a x = y \iff a^y = x$

Example: $2^5 = 32 \iff \log_2 32 = 5$

$log_2\ n$, often known in CS as *log n*

After all, base 2 is our favourite base in CS .. :)

Efficiency

In conclusion

For a sorted list with *n* items…

search is fast: O(log *n*) worst case, because of binary search

For an n-element list, it takes time proportional to n steps to decide whether the list contains a value, but only time proportional to log(n) to do the same thing on a sorted list

insert and delete can be slow, if inserting/removing from the *front* of the list – O(*n*) in the worst case

Efficiency

Trees

For a general tree with *n* items…

```
for subtree in self.subtrees:
    if subtree.__contains__(item):
        return True
return False
```

Trees

Efficiency

For a general tree with *n* items…

insert can be fast, if you insert as a child of the root – O(1)

search and delete can be slow, since you might need to check every item in the tree – O(*n*) in the worst case

General Trees

Efficiency

Strategy – similar to unsorted lists:

We must look at elements one by one

Each time we eliminate only one element from consideration

In the worst case, we look at all n elements

Either way, it takes time proportional to n

Efficiency

Worst case running times so far...

operation	Sorted List	Tree	Binary Search Tree
search	O(log *n*)	O(*n*)	
insert	O(*n*)	O(1)	
delete	O(*n*)	O(*n*)	

Efficiency

Worst case running times so far...

Better?

operation	Sorted List	Tree	Binary Search Tree
search	O(log *n*)	O(*n*)	
insert	O(*n*)	O(1)	
delete	O(*n*)	O(*n*)	

WORKSHEET

Efficiency

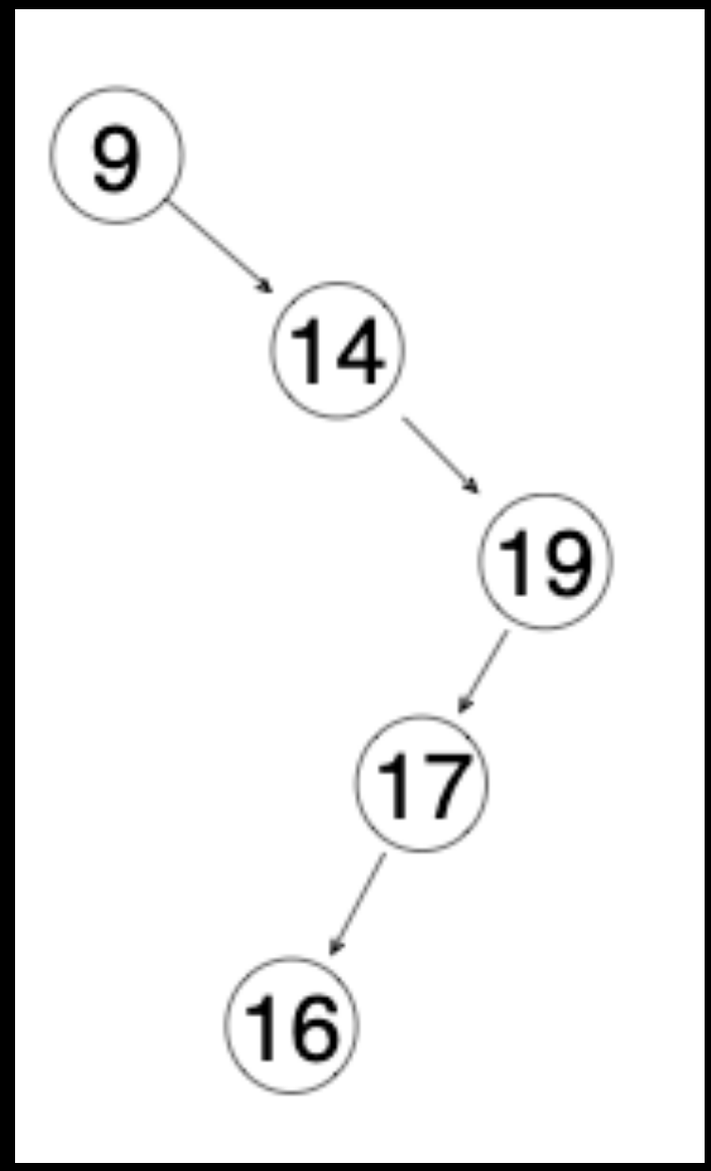
Efficiency

Search speed in BST

Time will be proportional to log n, only if the tree is balanced!

Example (imbalanced tree):

Time takes proportional to n in this case



We say that a BST is balanced if its left and right subtrees have roughly equal heights, and these subtrees are also balanced.

Balanced BSTs have height ≈ log n.

Recursion

Efficiency of BST operations

In a binary search tree, each Multiset operation's worst-case running time is proportional to the height h of the tree (where $\log n \leq h \leq n$).

operation	Sorted List	Tree	Binary Search Tree
search	O(log *n*)	O(*n*)	**O(*h*)**
insert	O(*n*)	O(1)	**O(*h*)**
delete	O(*n*)	O(*n*)	**O(*h*)**

Efficiency

To be continued in later courses...

In later courses, you will learn about balanced trees (AVL trees, red-black trees, etc.)

AVL trees ...

operation	Sorted List	Tree	Unbalanced BST	Balanced BST
search	O(log *n*)	O(*n*)	O(*h*): O(*n*)	O(*h*): **O(log *n*)**
insert	O(*n*)	O(1)	O(*h*): O(*n*)	O(*h*): **O(log *n*)**
delete	O(*n*)	O(*n*)	O(*h*): O(*n*)	O(*h*): **O(log *n*)**

Efficiency

Other trees and getting to a leaf faster..

If you have enormous amounts of data, binary trees won't cut it (getting to a leaf is still expensive)

Databases are such examples (large volumes of data, must fit in memory)

Increase the arity/branching factor!

Make heavy use of B-trees..

You will see this in later courses (CSC343, CSC443)

List Comprehensions

List Comprehensions

New lists from old

Suppose L is a list of the first hundred natural numbers

```
L = list(range(100))
```

If I want a new list with the squares of all the elements of L, I could:

```
new_list = []
for x in L:
      new_list.append(x * x)
```

Or I could use the equivalent list comprehension

```
new_list = [x * x  for x in L]
```

General comprehension pattern

```
[expression for name in iterable]
```

expression evaluates to a value

name refers to each element in the iterable (list, tuple, dict, ...)

WORKSHEET

List Comprehensions

List Comprehension

Recall: Sum for elements in nested list

```
def sum_list(L):
    if  isinstance(obj, int) :          # base case
        return obj
    else:                               # recursive step
        s = 0
        for elem in L:
            # calculate the sum of the sublist "elem" recursively
            s += sum_list(elem)
        return s
```

We could rewrite the recursive step using list comprehensions:

```
else:
    return  sum([sum_list(elem)  for elem in L])
```

Quizizz time!

Recursion Practice

https://quizizz.com/admin/quiz/6238dbdde00013001d0a3973/recursion-practice