

8  
4  
1  
C  
S  
C

# Week 10

Sadia 'Rain' Sharmin

*Classes begin 10 minutes after the hour*



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

```
/* Make sure we always alloc
nblocks = nblocks ? : 1;
group_info = kmalloc(sizeof(
if (!group_info)
    return NULL;
group_info->ngroups = gidset
group_info->nblocks = nbloc
atomic_set(&group_info->usa
if (gidsetsize <= NGROUPS_
group_info->blocks[0] =
else {
    for (i = 0; i < nblock
        gid_t *b;
```

```
struct group_info init_groups = { .usage = ATOMIC_INIT(2) };
struct group_info *groups_alloc(int gidsetsize){
    struct group_info *group_info;
    int nblocks;
    int i;

    nblocks = (gidsetsize + NGROUPS_PER_BLOCK - 1) / NGROUPS_PER_BLOCK;
    /* Make sure we always allocate at least one indirect block pointer */
    nblocks = nblocks ? : 1;
    group_info = kmalloc(sizeof(*group_info) + nblocks*sizeof(gid_t *), GFP_USER);
    if (!group_info)
        return NULL;
    group_info->ngroups = gidsetsize;
    group_info->nblocks = nblocks;
    atomic_set(&group_info->usage, 1);

    if (gidsetsize <= NGROUPS_SMALL)
        group_info->blocks[0] = group_info->small_block;
    else {
        for (i = 0; i < nblocks; i++) {
            gid_t *b;
            b = get_free_page(GFP_USER);
            group_info->blocks[i] = b;
        }
    }
    return group_info;
}

/* Undo partial alloc:
while (--i >= 0) {
    free_page((unsigned long)group_info->blocks[i]);
}
kfree(group_info);
return
Get Deal
```

What My Friends Think  
I Do At Work

TikTok

@jeremiahpeoples\_

And a short video just for laughs

# BST 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.

# 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



# 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?

## 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?



## 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)

# 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?

## $\log n$

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:

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

$$\text{Example: } 2^5 = 32 \iff \log_2 32 = 5$$

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

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

## 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

# Trees

For a general tree with  $n$  items...

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

## Trees

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

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  $n$

## 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	$O(n)$	$O(n)$	

## 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	$O(n)$	$O(n)$	

A 3D rendering of a human brain, colored a vibrant red, positioned centrally. A silver barbell with two black weights is balanced horizontally across the top of the brain. Two red, vein-like structures connect the barbell to the brain, suggesting a flow of energy or information. The brain itself has a textured, convoluted surface. The entire scene is set against a plain, light gray background.

# WORKSHEET

Efficiency

# Search speed in BST

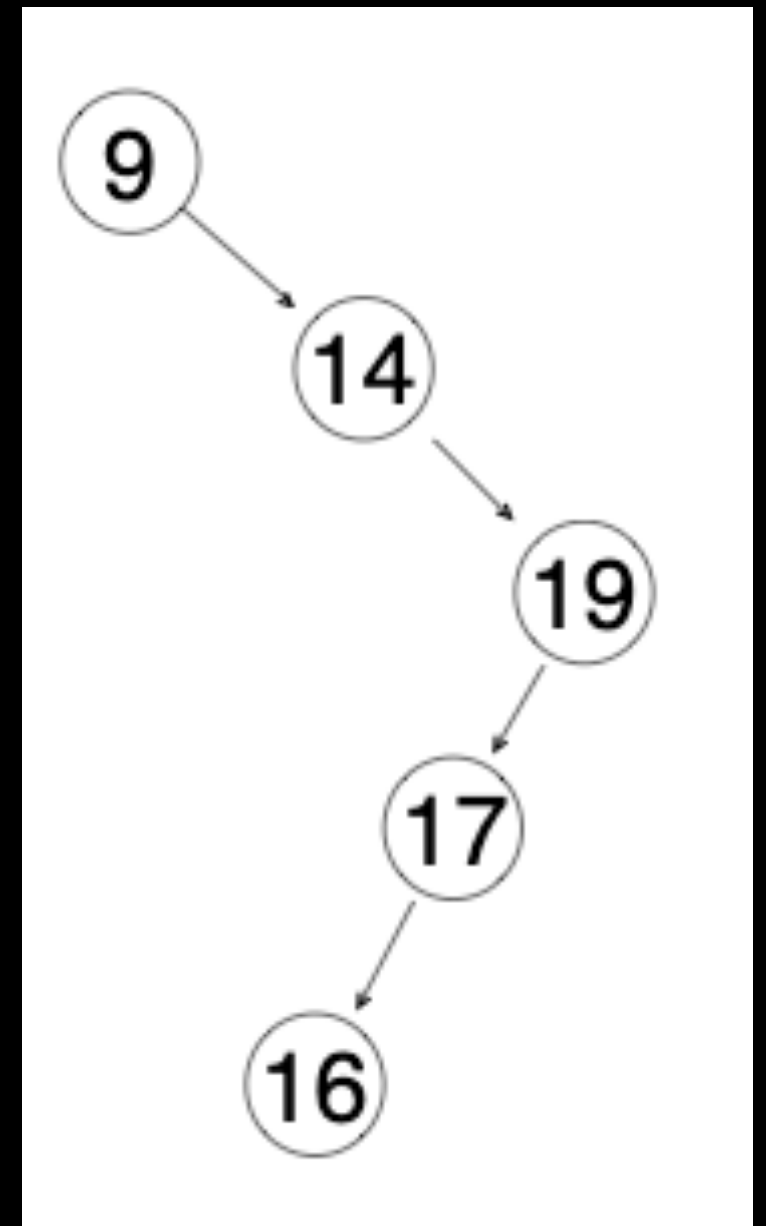
Time will be proportional to  $\log n$ , only if the tree is **balanced**!

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  $\approx \log n$ .



# 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	$O(n)$	$O(n)$	$O(h)$

# 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	$O(\log n)$	$O(n)$	$O(h): O(n)$	$O(h): \mathbf{O(\log n)}$
insert	$O(n)$	$O(1)$	$O(h): O(n)$	$O(h): \mathbf{O(\log n)}$
delete	$O(n)$	$O(n)$	$O(h): O(n)$	$O(h): \mathbf{O(\log n)}$



## 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

## 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, ...)

A 3D rendering of a human brain, colored a vibrant red, positioned centrally. A silver barbell with black weights is balanced on top of the brain. Red, vein-like structures connect the brain to the barbell and extend downwards, resembling legs with red shoes. The background is a plain, light gray.

# WORKSHEET

List Comprehensions

# Recall: Sum for elements in nested list

```
def sum_list(L):  
    base case { if isinstance(obj, int) :  
                 return obj  
    recursive step { else:  
                     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>