CMSC 420

Advanced Data Structures
Section 0101
Mon/Wed, 3:30pm, CSIC1115

Welcome!

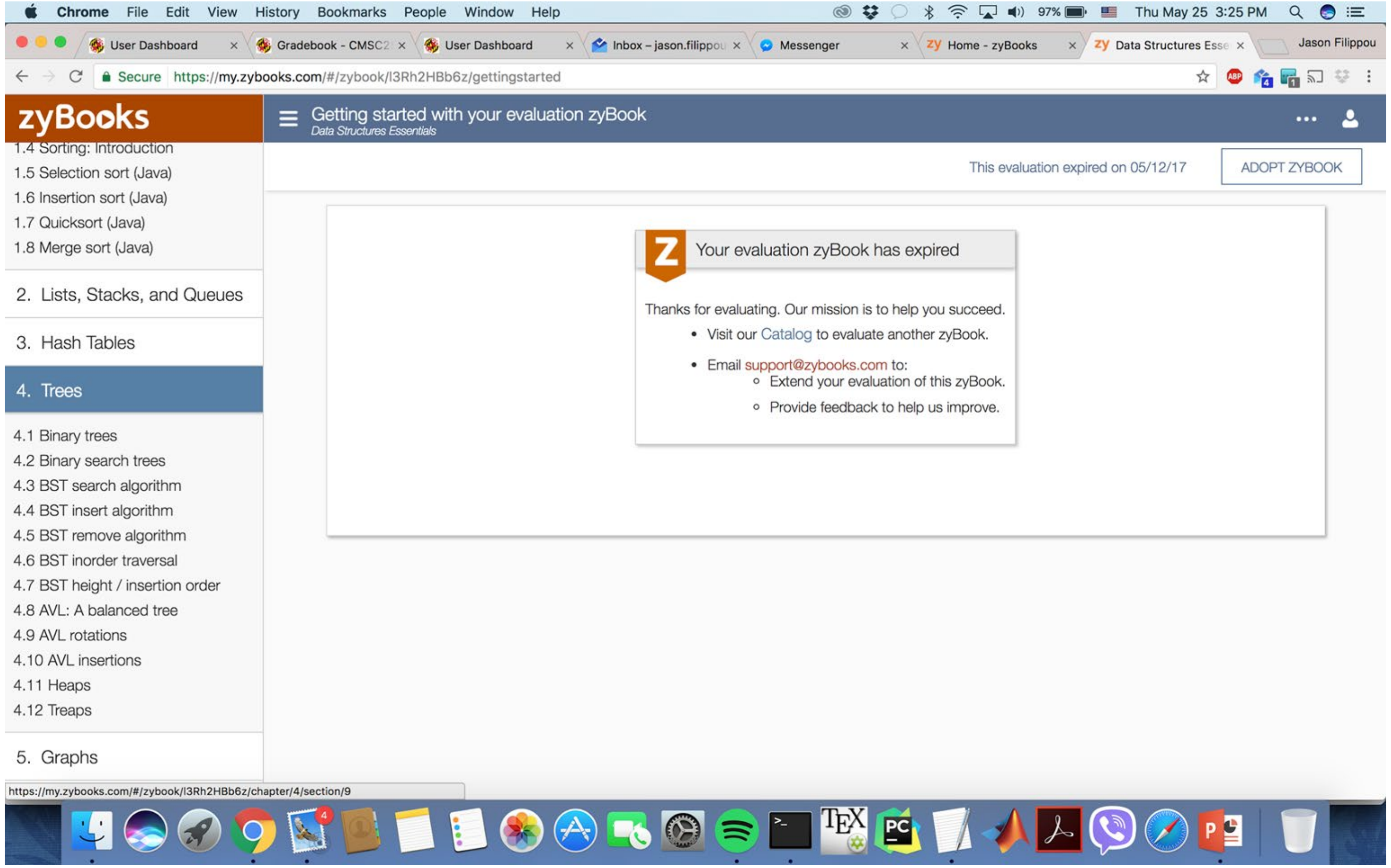
- CMSC420, $\underbrace{\text{“Advanced”}}_{\textit{Will explain….}}$ Data Structures

- Elective, generally junior / senior – level
- You need 351 and 330 to register!
- Quick facts:
  - 5-6 programming projects in Java, tested on submit.cs.
  - Lecture worksheets. (start Wednesday)
  - Homework assignment every 2-3 weeks (5-6 total).
  - 1 midterm, **Wed 10-16, 6-8pm, ESJ (Edward St. John's) 0202**
  - 1 final, probably **Friday Dec 13 from 1:30 - 3:30 based on this.**

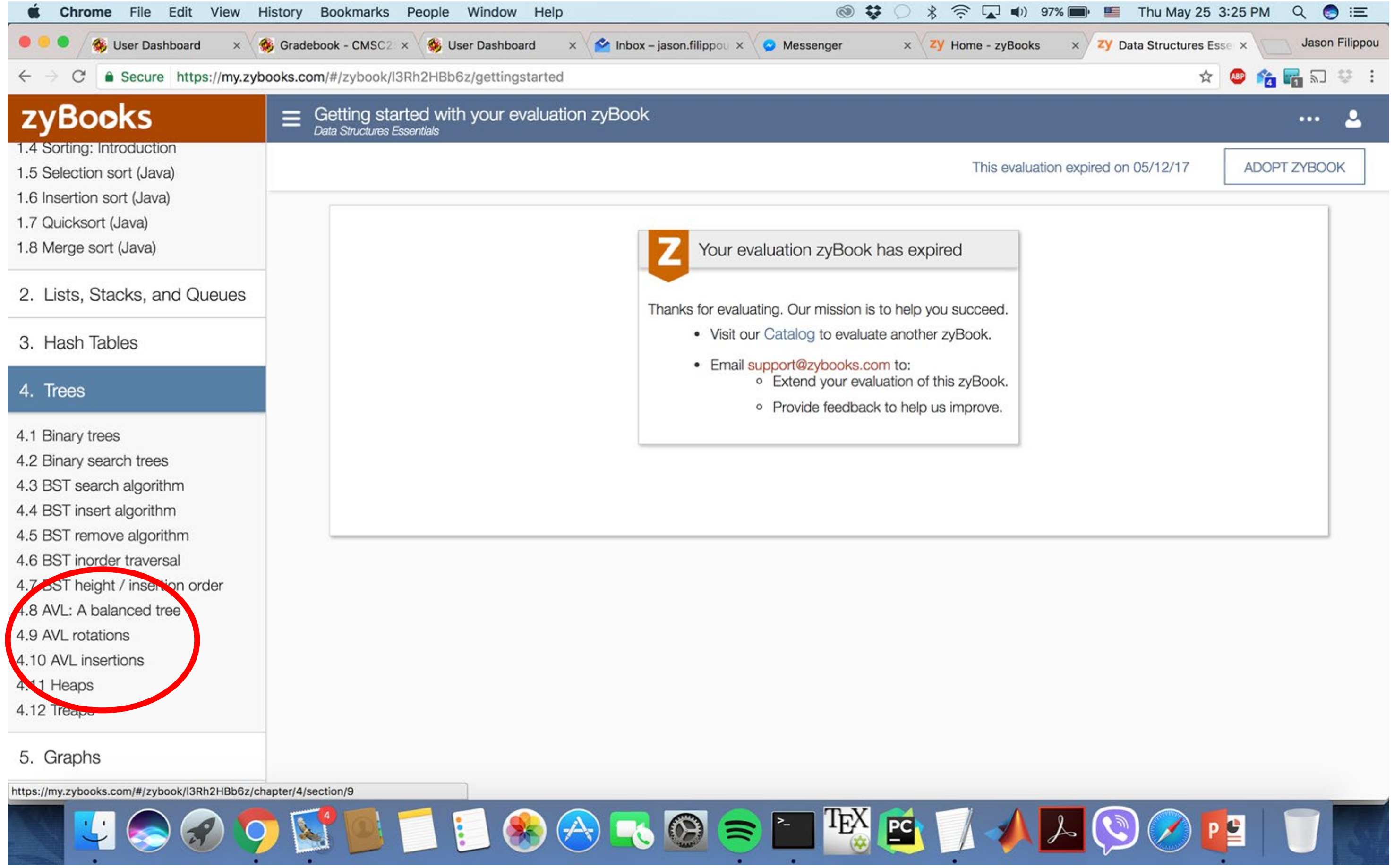
You’ll need

- **Decent** 351 understanding
  - You don’t need to remember NP-completeness or MSTs or the knapsack problem, but you will definitely need $\mathcal{O}(\cdot)$ notation, divide-and-conquer, greedy algorithms, heaps, binary search,…
- **Strong** Java skills
  - 132 and above. Anonymous inner classes, lambdas, iterators, exceptions, interfaces vs abstract classes, tail recursion,…
  - If you have C++ / Scala background, you will learn Java in 2 days.
- Another CS course that is **not** heavy in coding ☺
  - Unless you are really confident you can write correct code, fast!

420 in CS UMD


zyBooks
Getting started with your evaluation zyBook
Data Structures Essentials
This evaluation expired on 05/12/17
ADOPT ZYBOOK
1.4 Sorting: Introduction
1.5 Selection sort (Java)
1.6 Insertion sort (Java)
1.7 Quicksort (Java)
1.8 Merge sort (Java)
2. Lists, Stacks, and Queues
3. Hash Tables
4. Trees
4.1 Binary trees
4.2 Binary search trees
4.3 BST search algorithm
4.4 BST insert algorithm
4.5 BST remove algorithm
4.6 BST inorder traversal
4.7 BST height / insertion order
4.8 AVL: A balanced tree
4.9 AVL rotations
4.10 AVL insertions
4.11 Heaps
4.12 Treaps
5. Graphs
Your evaluation zyBook has expired
Thanks for evaluating. Our mission is to help you succeed.
Visit our Catalog to evaluate another zyBook.
Email support@zybooks.com to:
Extend your evaluation of this zyBook.
Provide feedback to help us improve.

420 in CS UMD


Chrome File Edit View History Bookmarks People Window Help
97%
Thu May 25 3:25 PM
User Dashboard
Gradebook - CMSC2
User Dashboard
Inbox – jason.filippou
Messenger
Home - zyBooks
Data Structures Esse
Jason Filippou
Secure https://my.zybooks.com/#/zybook/l3Rh2HBb6z/gettingstarted
zyBooks
Getting started with your evaluation zyBook
Data Structures Essentials
1.4 Sorting: Introduction
1.5 Selection sort (Java)
1.6 Insertion sort (Java)
1.7 Quicksort (Java)
1.8 Merge sort (Java)
2. Lists, Stacks, and Queues
3. Hash Tables
4. Trees
4.1 Binary trees
4.2 Binary search trees
4.3 BST search algorithm
4.4 BST insert algorithm
4.5 BST remove algorithm
4.6 BST inorder traversal
4.7 BST height / insertion order
4.8 AVL: A balanced tree
4.9 AVL rotations
4.10 AVL insertions
4.11 Heaps
4.12 Treaps
5. Graphs
This evaluation expired on 05/12/17
ADOPT ZYBOOK
Your evaluation zyBook has expired
Thanks for evaluating. Our mission is to help you succeed.
Visit our Catalog to evaluate another zyBook.
Email support@zybooks.com to:
Extend your evaluation of this zyBook.
Provide feedback to help us improve.
https://my.zybooks.com/#/zybook/l3Rh2HBb6z/chapter/4/section/9

420 in CS UMD

4. Trees

4.1 Binary trees
4.2 Binary search trees
4.3 BST search algorithm
4.4 BST insert algorithm
4.5 BST remove algorithm
4.6 BST inorder traversal
4.7 BST height / insertion order
4.8 AVL: A balanced tree
4.9 AVL rotations
4.10 AVL insertions
4.11 Heaps
4.12 Treaps

420 in CS UMD

4. Trees

4.1 Binary trees
4.2 Binary search trees
4.3 BST search algorithm
4.4 BST insert algorithm
4.5 BST remove algorithm
4.6 BST inorder traversal
4.7 BST height / insertion order
4.8 AVL: A balanced tree
4.9 AVL rotations
4.10 AVL insertions
4.11 Heaps
4.12 Treaps

Very basic for 420 (2nd project)

Tower of “core” CS courses

420	451
330	351
132	216
131	250

Tower of "core" CS courses

Lists, queues, stacks, BSTs, simple hashes, graphs, heaps.

Tower of “core” CS courses

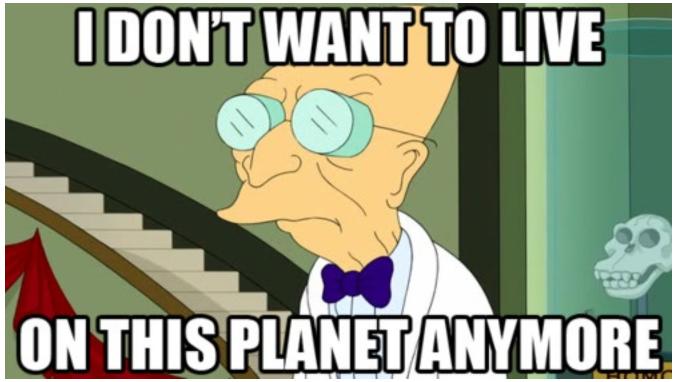
420	451
330	351
132	216
131	250

Lists, queues, stacks, BSTs, simple hashes, graphs, heaps.

Tower of “core” CS courses

Balanced BSTs, B-Trees, SkipLists, hashing in depth, Tries, Huffman, LZW, KMP, Suffix Trees & Arrays, Huffman, LZW, KD-Trees, QuadTrees, MinHash, LSH….

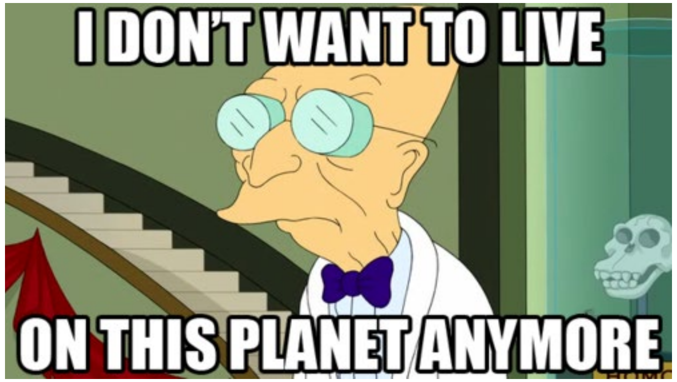
420	451
330	351
132	216
131	250



Tower of “core” CS courses

Balanced BSTs, B-Trees, SkipLists, hashing in depth, Tries, Huffman, LZW, KMP, Suffix Trees & Arrays, Huffman, LZW, KD-Trees, QuadTrees, MinHash, LSH….

420	451
330	351
132	216
131	250



Hence “***Advanced***” Data Structures

A 351 question

- Given a **`double`** (8 bytes) array A of size N, which among the following sorting algorithms offers the best **time complexity?**

A 351 question

- Given a **`double`** (8 bytes) array A of size N, which among the following sorting algorithms offers the best **time complexity?**

InsertSort

Mergesort

Bubblesort

Selection sort

A 351 question …. with a 351 answer!

- Given a **`double`** (8 bytes) array A of size N, which among the following sorting algorithms offers the best **time complexity?**

InsertSort	Mergesort	Bubblesort	Selection sort

- $\mathcal{O}(n \cdot logn)$ worst **and** average time complexity
- $\mathcal{O}(n)$ spatial cost ☹ (not in-place)
- In-place (but non-stable) variants have been proposed, but we won't talk about them

Another 351 question

- Given a ***linked list*** of 8 byte integers of size N, does mergesort achieve the same runtime?

Yes

No

Another 351 question… with a 351 answer!

- Given a ***linked list*** of 8 byte integers of size N, does mergesort achieve the same runtime?

Yes

No

- If you do it ***bottom-up***!
- For more details with pseudocode, you are directed to the Wikipedia article

Yet another 351 question

- Given a ***sorted linked list*** of 8 byte integers of size N, what's the **best runtime** we can achieve for **binary search**?

$\mathcal{O}(\log_2 n)$	$\mathcal{O}(n \cdot \log_2 n)$	$\mathcal{O}(n)$	Something else (what?)

Yet another 351 question.. <u>**with a 420 answer!**</u>

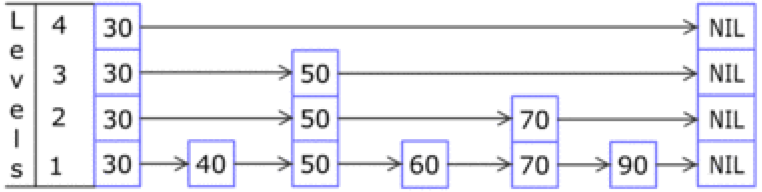
- Given a ***<u>sorted linked list</u>*** of 8 byte integers of size N, what's the **best runtime** we can achieve for **binary search**?

$\mathcal{O}(\log_2 n)$

$\mathcal{O}(n \cdot \log_2 n)$

$\mathcal{O}(n)$

Something else (what?)



- Use a probabilistic data structure known as a **skiplist**!
- We will talk about it later in the semester.

351 question #4

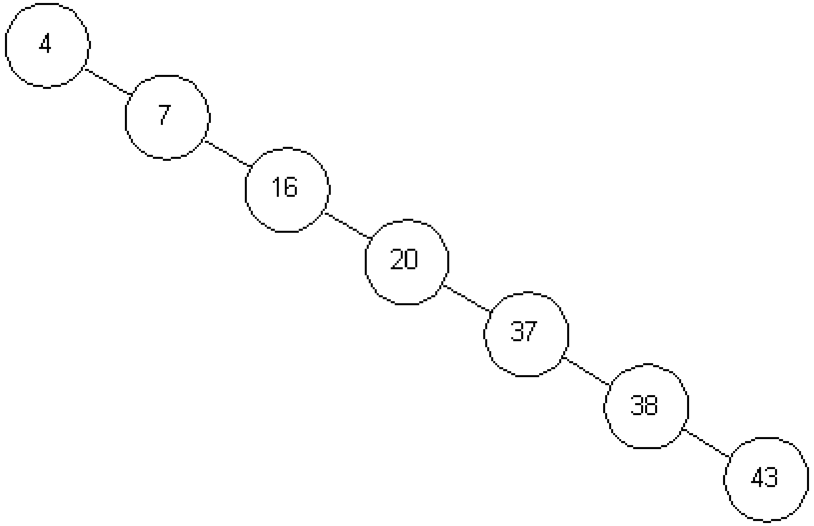
- Suppose I have n integers to store in a binary search tree. In the worst case, **how many links** will I have to traverse to find one of these integers in the tree?

$\log_2 n$	$2n$	n	$\lceil n/2 \rceil$

351 question #4

- Suppose I have n integers to store in a binary search tree. In the worst case, **how many links** will I have to traverse to find one of these integers in the tree?

$\log_2 n$	$2n$	n	$\lceil n/2 \rceil$

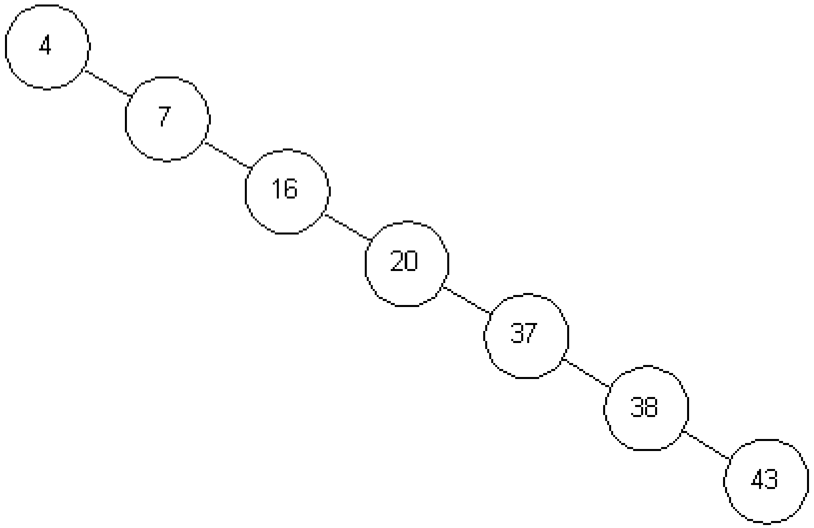


- Worst case: we receive the input in ascending (or descending) order, and our binary search tree devolves into a linked list ☹

351 question #4

- Suppose I have n integers to store in a binary search tree. In the worst case, **how many links** will I have to traverse to find one of these integers in the tree?

$\log_2 n$	$2n$	n	$\lceil n/2 \rceil$



- Worst case: we receive the input in ascending (or descending) order, and our binary search tree devolves into a linked list ☹
- ***<u>On average</u>, how many links will I traverse to find any one of these keys?***

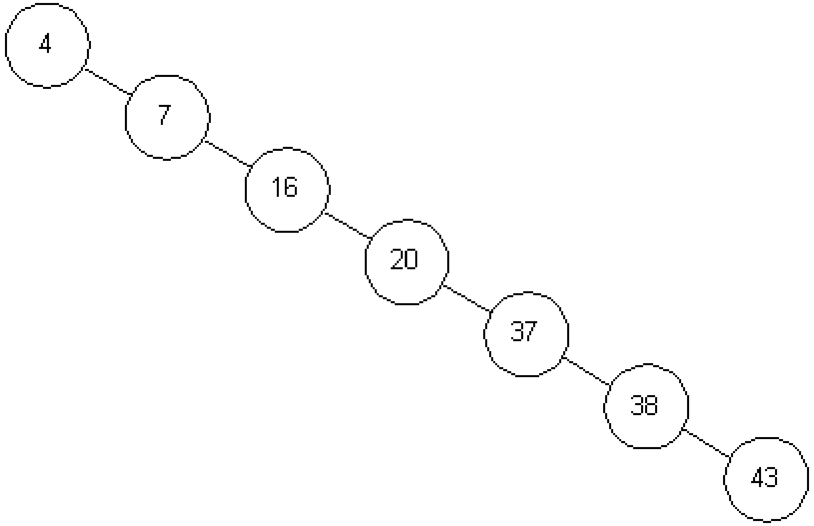
351 question #4

This term will be very important for us, especially in hashing!

- Suppose I have n integers to store in a binary search tree. In the worst case, **how many links** will I have to traverse to find one of these integers in the tree?

$\log_2 n$	$2n$	n	$\lceil n/2 \rceil$

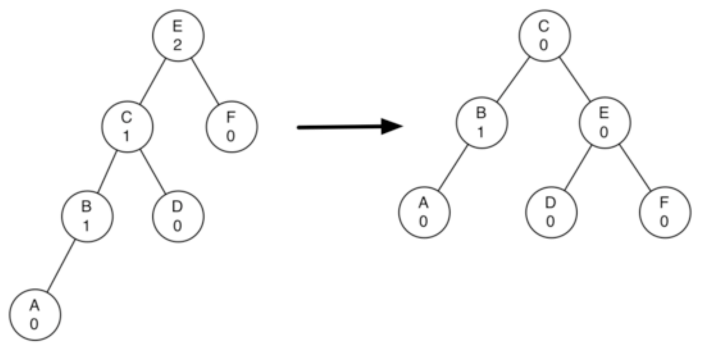
Both "absolute" worst-case **and** ***average worst case*** search in a linked list is linear on n.



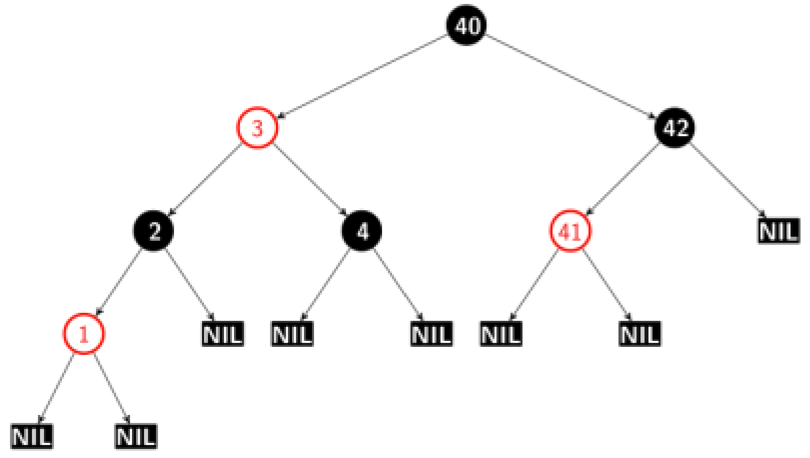
- Worst case: we receive the input in ascending (or descending) order, and our binary search tree devolves into a linked list ☹
- ***On average, how many links will I traverse to find any one of these keys?***
- $\frac{1+2+3+\ldots+n}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2} = \frac{1}{2}n + \frac{1}{2} = \mathcal{O}(n)!$

A 420 solution

- To ensure a global height in $\mathcal{O}(\log_2 n)$ **at all times**, we should enhance our BST and make it into an AVL or Red-**Black** Tree!



Example of rotating a node (in this case, the root) in an AVL tree to restore its balance.



A red-black tree

Take-home message #1 : ***changing structures***

- Take linked list mergesort scenario.
- You moved from an array to a linked list.

Take-home message #1 : ***changing structures***

- Take linked list mergesort scenario.
- You moved from an array to a linked list.
  - ***Why? What's the chief reason?***

Take-home message #1 : ***changing structures***

- Take linked list mergesort scenario.
- You moved from an array to a linked list.
  - ***Why? What's the chief reason?***
  - **Non-contiguous storage in memory.**

Take-home message #1 : ***changing structures***

- Take linked list mergesort scenario.
- You moved from an array to a linked list.
  - ***Why? What's the chief reason?***
  - **Non-contiguous storage in memory.**
- But what if that breaks your existing operations, or makes them less efficient?
  - You have to make sure this is avoided.
  - Sometimes (e.g mergesort) this might need a **radical change to your algorithm.**
  - Sometimes (e.g binary search in sorted linked list) you might need a new **data structure**!

Take 2 to chat 420 and each other

What do you hope to achieve in 420?

Why'd you register for this class?

How'd you do in 351?

Olive oil: healthy and delicious or a big Greek conspiracy?

What have you heard about the projects?

What's your name?

What other courses are you taking?

Any internships?

Where do you come from?

Any thoughts of grad school?

How do you make a Computer Scientist cry?

Favorite programming language?

Arrays and Bags…

- Perhaps the simplest abstract data structure that we can come up with is a bag
  - Abstract can mean a lot of things
  - We think of it as "a conceptual piece of data storage that has to have some properties and time/space constraints for common operations"

Arrays and Bags…

- Perhaps the simplest abstract data structure that we can come up with is a bag
  - Abstract can mean a lot of things
  - We think of it as "a conceptual piece of data storage that has to have some properties and time/space constraints for common operations"

Requirements:

1. We want to be able to **put stuff in the bag**
2. We want to be able to check if the bag is empty.
3. We want to check what its size is.
4. **We can "shake" the bag , randomly perturbing the order of its elements *(key operation)***
5. We can **loop through (iterate over)** all of its elements, one at a time.

Arrays and Bags…

```java
package edu.umd.cs.datastructures.bags;

/**...*/
public interface Bag<Item> extends Iterable<Item>{ // So classes implementing it have to expose a fail-safe iterator.


    /**...*/
    void add(Item i);

    /**...*/
    boolean isEmpty();

    /**...*/
    void shake();

    /**...*/
    int size();

    /**...*/
    @Override
   String toString();

}
```

Requirements:

1. We want to be able to **put stuff in the bag**
2. We want to be able to check if the bag is empty.
3. We want to check what its size is.
4. **We can "shake" the bag , randomly perturbing the order of its elements *(key operation)***
5. We can **loop through (iterate over)** all of its elements, one at a time.

Arrays and Bags…

```java
package edu.umd.cs.datastructures.bags;

/**...*/
public interface Bag<Item> extends Iterable<Item>{ // So classes implementing it have to expose a fail-safe iterator.

    /**...*/
    void add(Item i);

    /**...*/
    boolean isEmpty();

    /**...*/
    void shake();

    /**...*/
    int size();

    /**...*/
    @Override
   String toString();

}
```

Requirements:

1. We want to be able to **put stuff in the bag**
2. We want to be able to check if the bag is empty.
3. We want to check what its size is.
4. **We can "shake" the bag , randomly perturbing the order of its elements *(key operation)***
5. We can **loop through (iterate over)** all of its elements, one at a time.

Arrays and Bags…

```java
package edu.umd.cs.datastructures.bags;

/**...*/
public interface Bag<Item> extends Iterable<Item>{ // So classes implementing it have to expose a fail-safe iterator.

    /**...*/
    void add(Item i);

    /**...*/
    boolean isEmpty();

    /**...*/
    void shake();

    /**...*/
    int size();

    /**...*/
    @Override
    String toString();
}
```

Adding to an interface is fine, as long as you **<u>document</u>** this addition! ☺

Requirements:

1. We want to be able to **put stuff in the bag**
2. We want to be able to check if the bag is empty.
3. We want to check what its size is.
4. **We can "shake" the bag , randomly perturbing the order of its elements *<u>(key operation)</u>***
5. We can **loop through (iterate over)** all of its elements, one at a time.

Arrays and Bags…

```java
package edu.umd.cs.datastructures.bags;

/**...*/
public interface Bag<Item> extends Iterable<Item>{ // So classes implementing it have to expose a fail-safe iterator.

    /**...*/
    void add(Item i);

    /**...*/
    boolean isEmpty();

    /**...*/
    void shake();

    /**...*/
    int size();

    /**...*/
    @Override
    String toString();
}
```

Adding to an interface is fine, as long as you **document** this addition! ☺

Let's switch gears to a Java demo real quick!

Requirements:

1. We want to be able to **put stuff in the bag**
2. We want to be able to check if the bag is empty.
3. We want to check what its size is.
4. **We can "shake" the bag , randomly perturbing the order of its elements *(key operation)***
5. We can **loop through (iterate over)** all of its elements, one at a time.

Take-home message #2: ***Hardware Considerations***

- In this demo we saw a classic difference between CS Theory and (hardware) practice…
- Sure, in theory, arrays are constant storage *(A[i] requires one to two multiplications and one addition)*
- But in practice, if they get too large, not all of them can fit in registers, or even cache!
  - Cache misses make your application (the **RandomAccessBag**) hurt.
- So **even for the simplest possible data structure** a language gives us (1D array), theory and practice **diverge**!

Not allowed in lecture!

If you make a mistake on a worksheet, ~~**cross it out**~~ with your pen or pencil so that:

(a) We can grade you fairly and
(b) You can see your mistake when you study again and remember *why* you made it (the worksheets will assist you with this)

Worksheets

- Some of our slides will have this icon: 

- It means that the question should be answered in your worksheets.
- If you get a question wrong, write the right answer under the line we provide. Also include the reasons for which you got it wrong!
  - *Do not erase or cross out the answer!* Write down the right answer under the line!

Another example: Java Arraylists

- Let’s look at the `ArrayList’`s docs for a moment…

Another example: Java Arraylists

- Let's look at the `ArrayList`'s docs for a moment…
- What's with this "amortized constant time" required for additions? Since the underlying implementation is an array, we'd expect just constant!

Another example: Java Arraylists

- Let’s look at the `ArrayList`’s docs for a moment…
- What’s with this “amortized constant time” required for additions? Since the underlying implementation is an array, we’d expect just constant!
- Note the docs for the constructors… default starting capacity at 10. When inserting 11th element, we need linear time to copy over to new internal array. ☹
  - But! For the next 9 elements, we have constant time! ☺

Another example: Java Arraylists

- Let's look at the `ArrayList`'s docs for a moment…
- What's with this "amortized constant time" required for additions? Since the underlying implementation is an array, we'd expect just constant!
- Note the docs for the constructors… default starting capacity at 10. When inserting 11^{th} element, we need linear time to copy over to new internal array. ☹
  - But! For the next 9 elements, we have constant time! ☺
  - So, on **average,** we have $\frac{30}{20} \approx 1.5$ unit cost for inserting elements!
  - This is an example of a technique called ***amortized analysis.***

Practice

- Suppose that we have an `ArrayList` `list1` which begins with 10 cells and adds 10 empty cells every time it gets full.
  - Concretely, it will resize itself ***at the first insertion that happens while it is already full.***
  - So, the first resizing will happen at the eleventh insertion.

Practice

- Suppose that we have an `ArrayList` `list1` which begins with 10 cells and adds 10 empty cells every time it gets full.
  - Concretely, it will resize itself ***at the first insertion that happens while it is already full.***
  - So, the first resizing will happen at the eleventh insertion.
- What is the amortized cost of inserting 50 elements into `list1`?

Practice

- Suppose that we have an `ArrayList` `list1` which begins with 10 cells and adds 10 empty cells every time it gets full.
  - Concretely, it will resize itself ***at the first insertion that happens while it is already full.***
  - So, the first resizing will happen at the eleventh insertion.
- What is the amortized cost of inserting 50 elements into `list1`?

Worksheet time!

Practice

- Suppose now that we have an `ArrayList` `list2` which begins with 10 cells and doubles its size every time it gets full!
- What is the amortized cost of inserting 50 elements into `list2`?

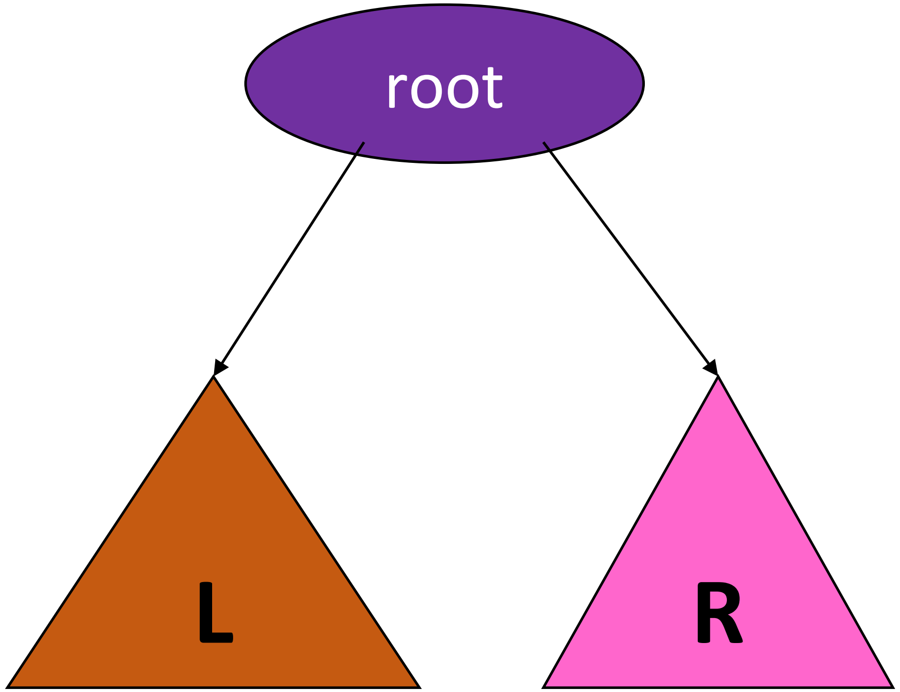
Take-home message #3: ***Amortized Complexity***

- If most of the time our operation runs very fast, are we willing to accept bad performance once in a while?
  - Amortized analysis can answer this question.
- `ArrayLists`, Splay Trees and Self-organizing hash tables are all data structures where certain operations have amortized constant complexity!

Take-home message #3: ***Amortized Complexity***

- If most of the time our operation runs very fast, are we willing to accept bad performance once in a while?
  - Amortized analysis can answer this question.
- `ArrayLists`, Splay Trees and Self-organizing hash tables are all data structures where certain operations have amortized constant complexity!
- Amortized complexity: An expression in big-Oh notation or English
  - Examples: $\mathcal{O}(n), \mathcal{O}(n^2)$, constant, linear, exponential….
- Amortized cost: The **exact** cost in terms of elementary operations (e.g swaps, reference assignments) that we incur for an operation.
  - Examples: $2n$, ,2.5 ,$1.53n^2 + 2n$

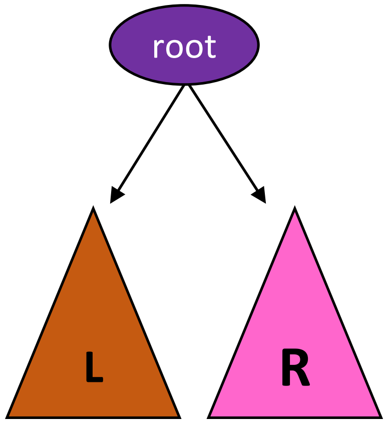
Reminder: Binary tree traversals



Reminder: Binary tree traversals

- ***Pre**order traversal is…*

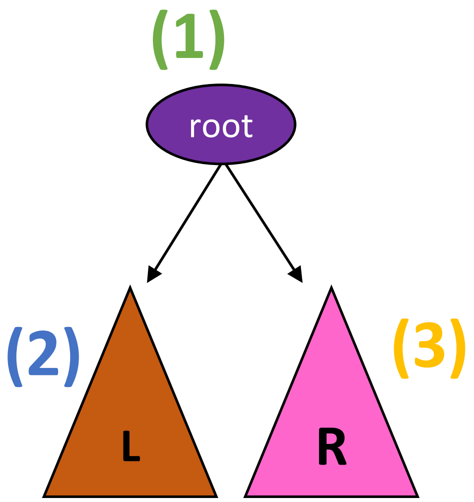
  - (a) L r R
  - (b) r L R
  - (c) r R L
  - (d) L R r



Reminder: Binary tree traversals

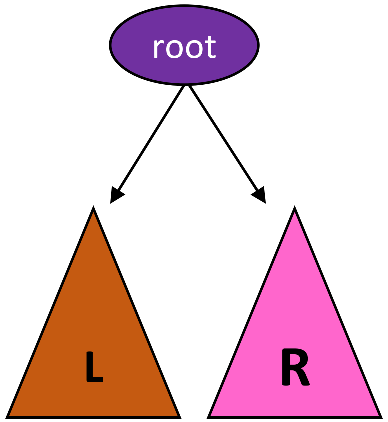
- ***<u>Pre</u>***order traversal is…
  - (a) L r R
  - (b) r L R
  - (c) r R L
  - (d) L R r

Mnemonic rule (maybe?):
pre-order traversal, as in **pre-visit the root** before the subtrees



Reminder: Binary tree traversals

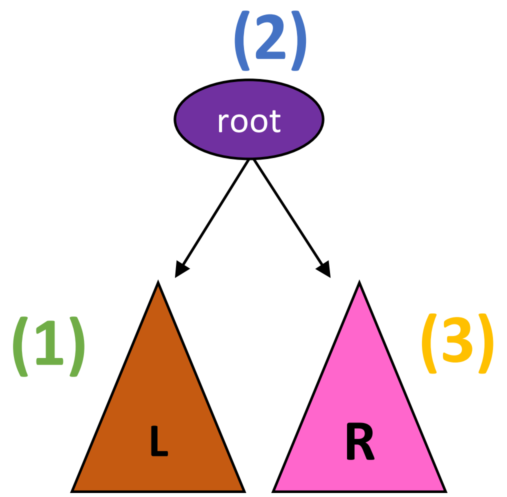
- ***In***order traversal is…
  - (a) L r R
  - (b) r L R
  - (c) r R L
  - **(d)** L R r



Reminder: Binary tree traversals

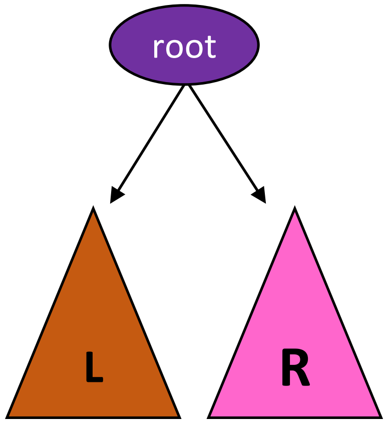
- ***In***order traversal is…
  - (a) L r R
  - (b) r L R
  - (c) r R L
  - (d) L R r

Mnemonic rule (maybe?):
in-order traversal, as in "visit the root ***in***-between the subtrees"



Reminder: Binary tree traversals

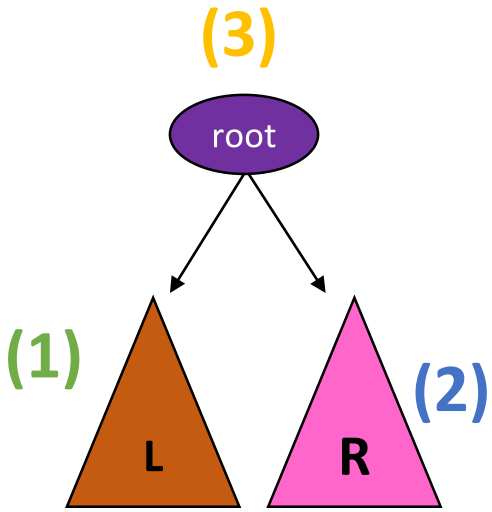
- ***Post*****order traversal is…**
  - (a) L r R
  - (b) r L R
  - (c) r R L
  - (d) L R r



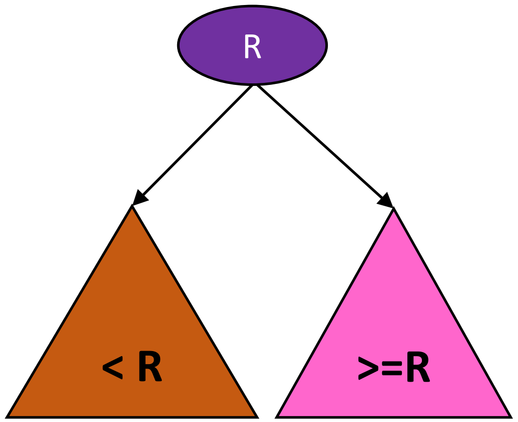
Reminder: Binary tree traversals

- ***Post*order traversal is…**
  - (a) L r R
  - (b) r L R
  - (c) r R L
  - (d) L R r

Mnemonic rule (maybe?):
post-order traversal, as in "visit the root ***post***-visiting the subtrees"

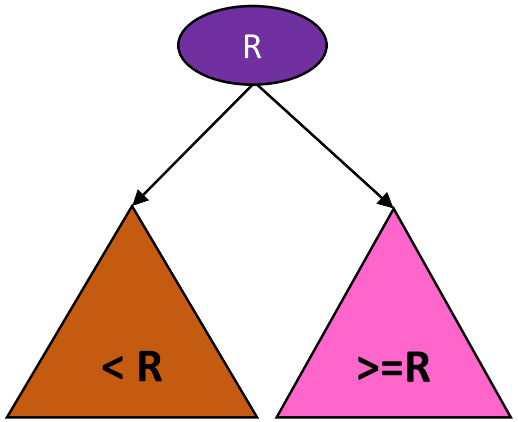


Question for you



- Suppose that our binary tree is a binary **search** tree

Question for you



- Suppose that our binary tree is a binary **search** tree

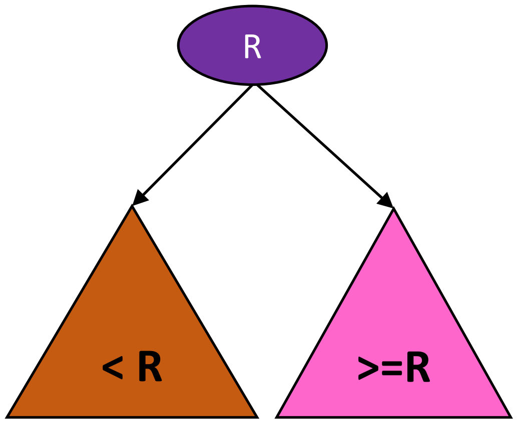
- Which tree traversal is the most **useful** one for BSTs, and **why**?

Preorder

Inorder

Postorder

Question for you



- Suppose that our binary tree is a binary **search** tree

- Which tree traversal is the most **useful** one for BSTs, and **why**?

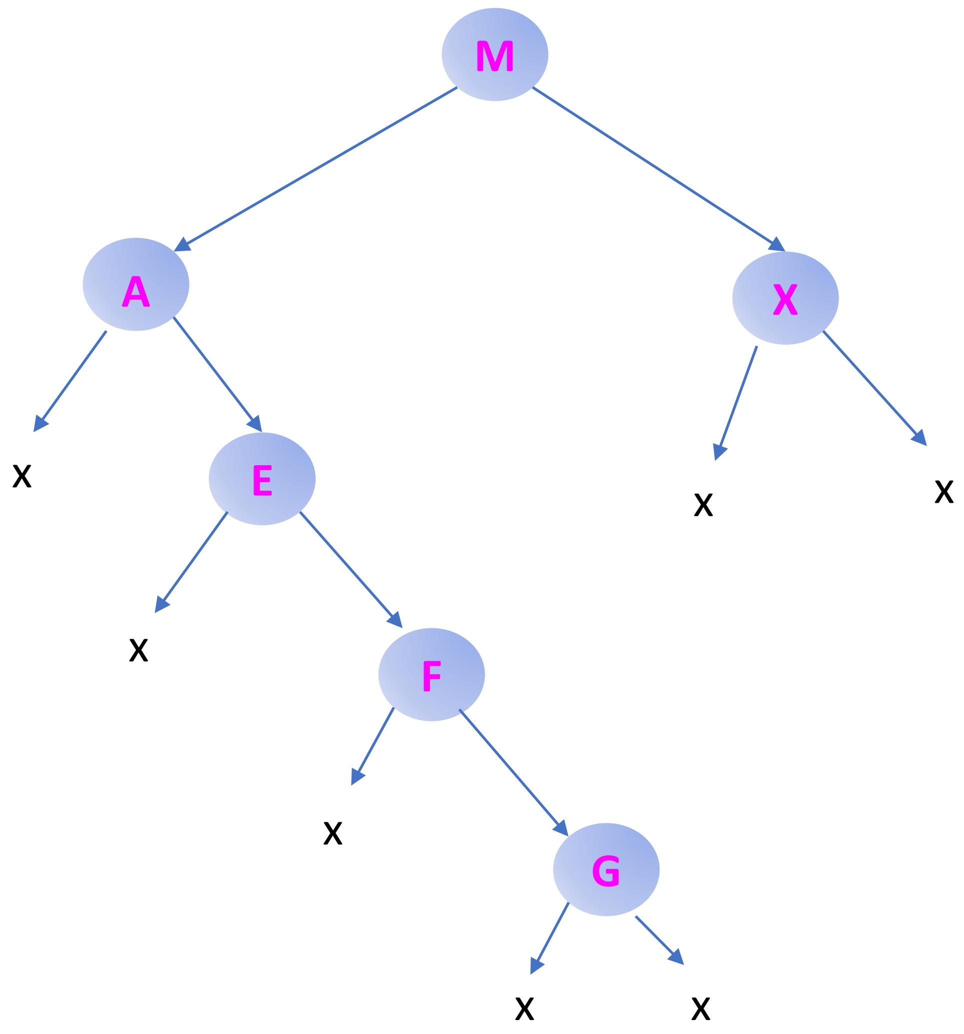
Preorder

Inorder

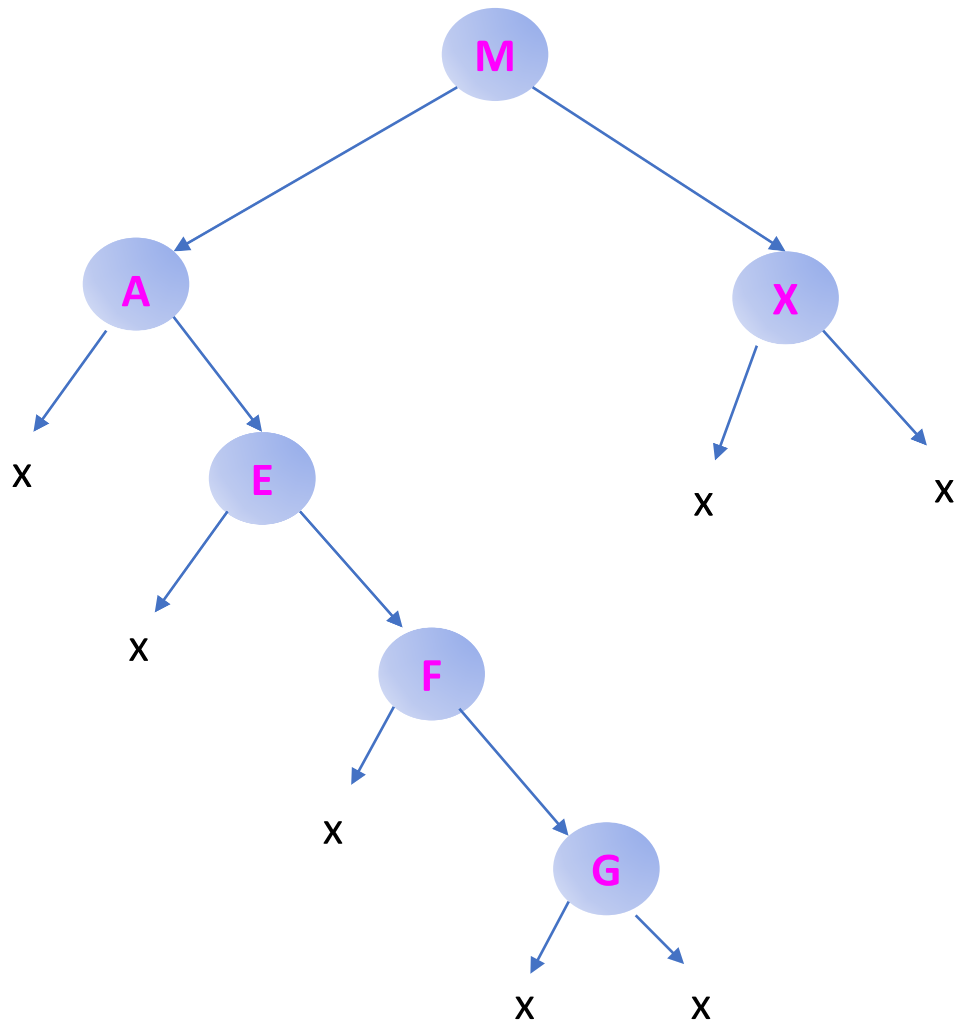
Postorder

In BSTs, Inorder is Sorted order! :)

Here's a BST

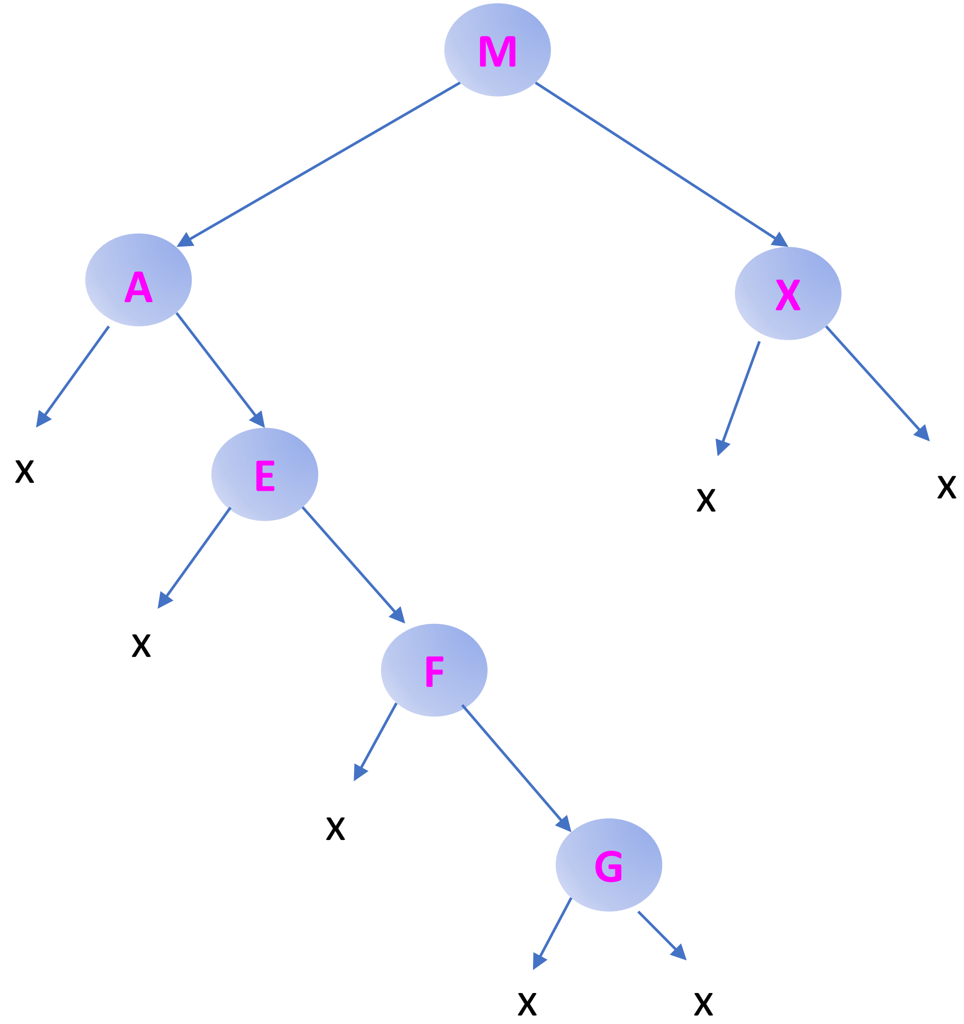


Here’s a BST



Tell me **THREE** things you don’t like about this tree.

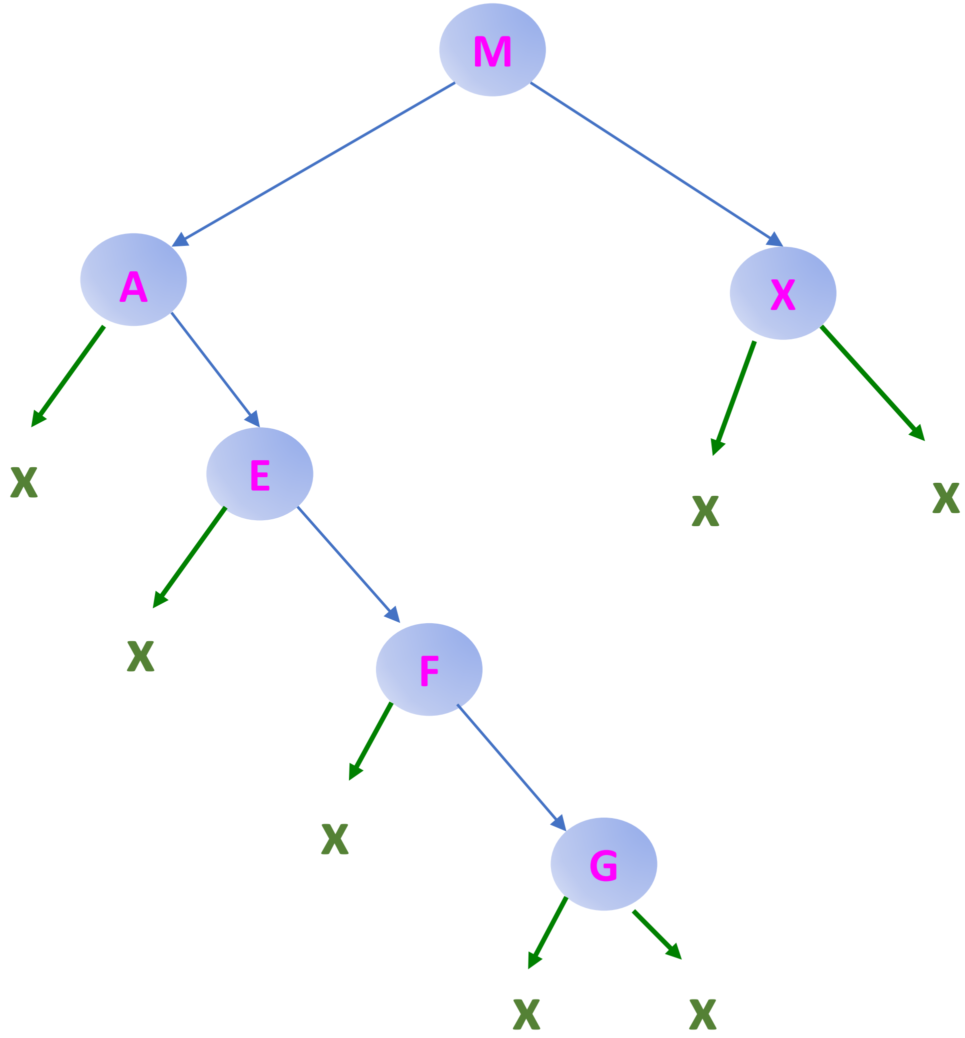
Here's a BST



Here are three things Jason doesn't like.

(1) Very unbalanced, say goodbye to $\approx \log_2 n$ search…. ☹

Here's a BST

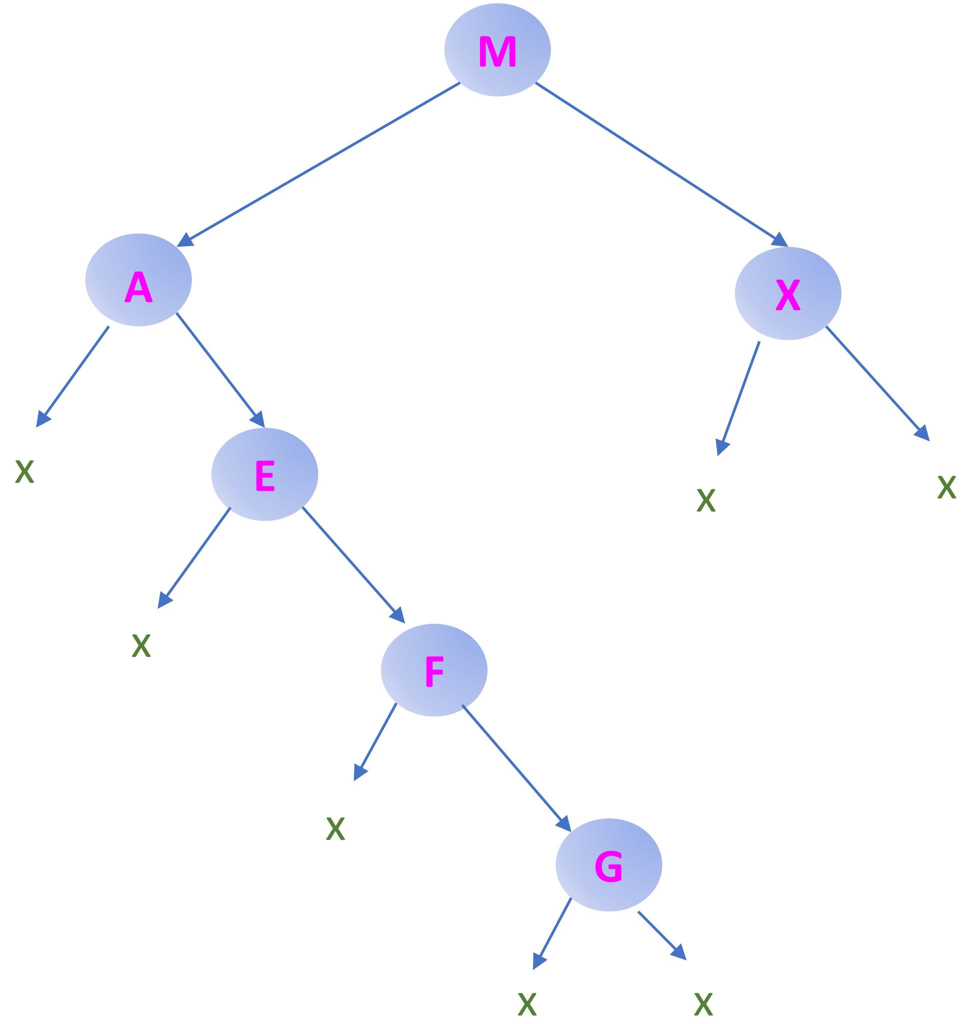


Here are three things **Jason** doesn't like.

(1) Very unbalanced, say goodbye to $\approx \log_2 n$ search…. ☹

(2) Wasting 8 bytes per null pointer ☹

Here’s a BST



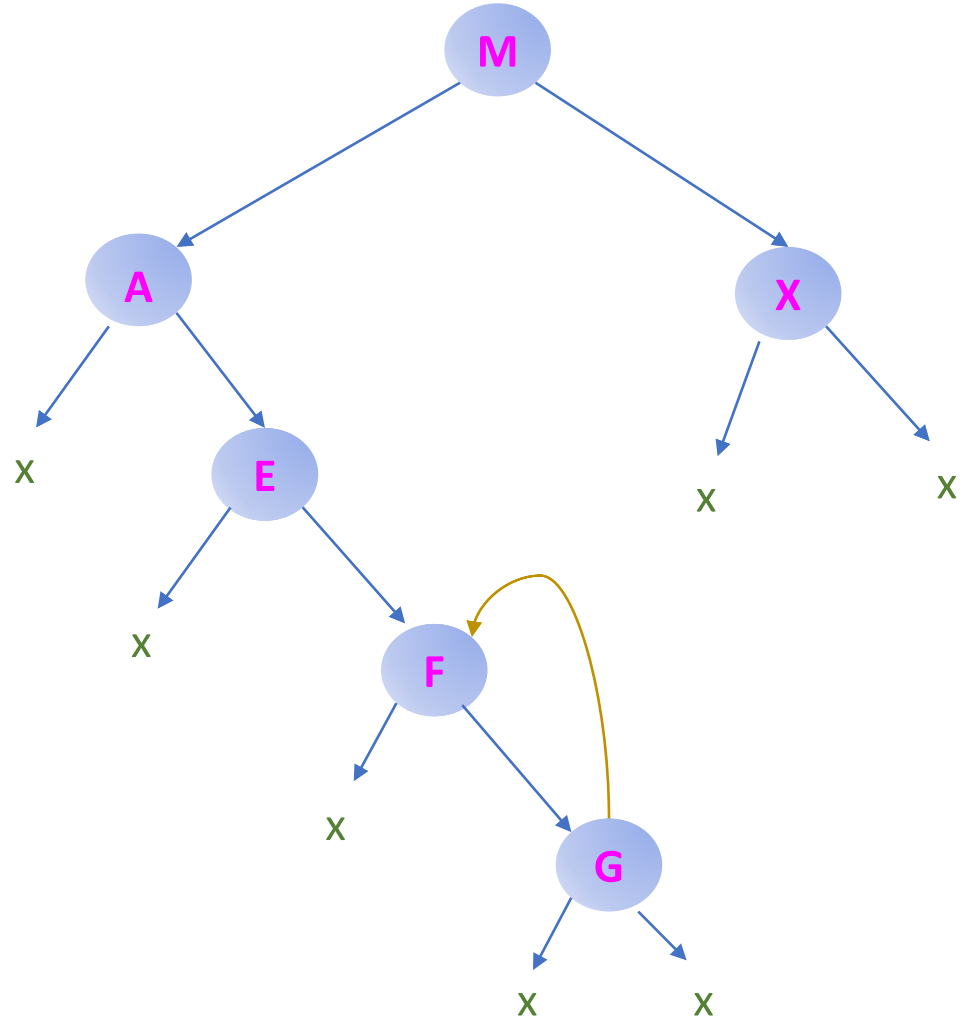
Here are three things Jason doesn’t like.

(1) Very unbalanced, say goodbye to $\approx \log_2 n$ search…. ☹

(2) Wasting 8 bytes per null pointer ☹

(3) Inorder successor of ‘G’ will take 4 stack pops to reach ☹

Here's a BST



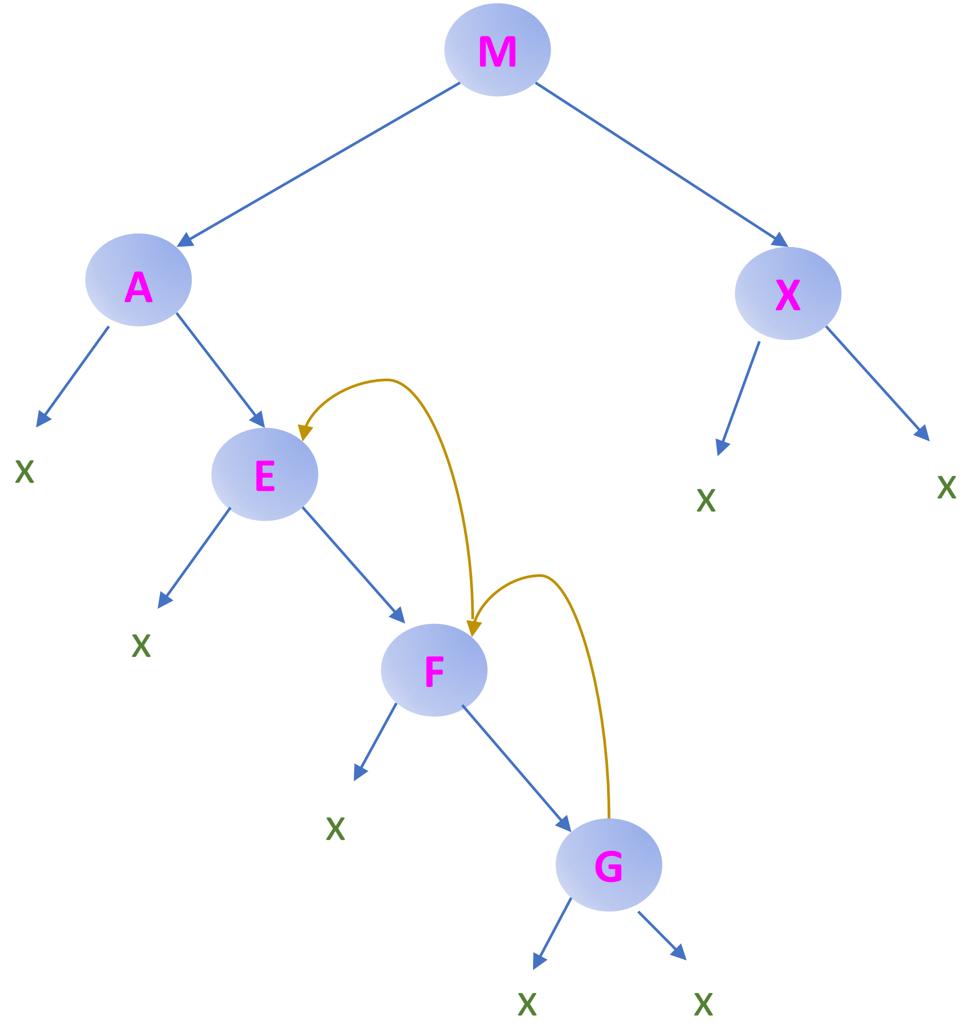
Here are three things Jason doesn't like.

(1) Very unbalanced, say goodbye to $\approx \log_2 n$ search.... ☹

(2) Wasting 8 bytes per null pointer ☹

(3) Inorder successor of 'G' will take 4 stack pops to reach ☹

Here's a BST



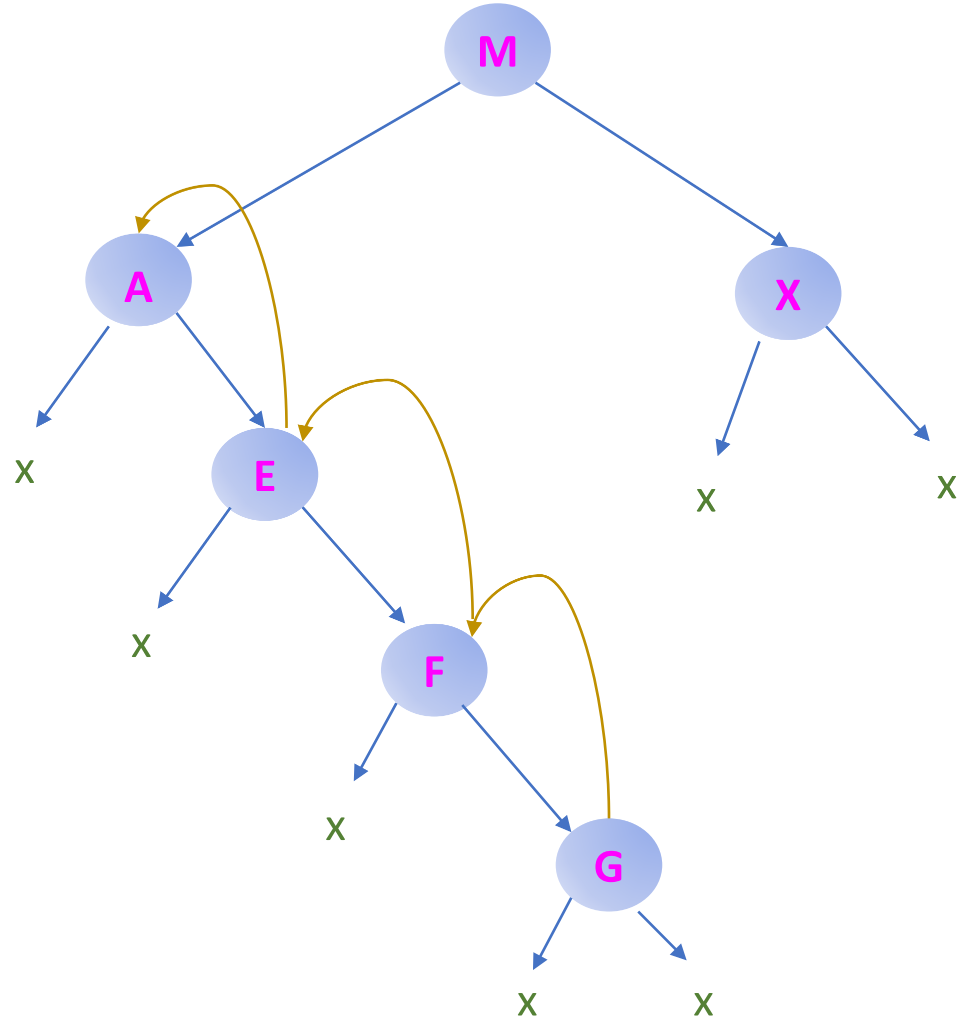
Here are three things Jason doesn't like.

(1) Very unbalanced, say goodbye to $\approx \log_2 n$ search…. ☹

(2) Wasting 8 bytes per null pointer ☹

(3) Inorder successor of 'G' will take 4 stack pops to reach ☹

Here’s a BST



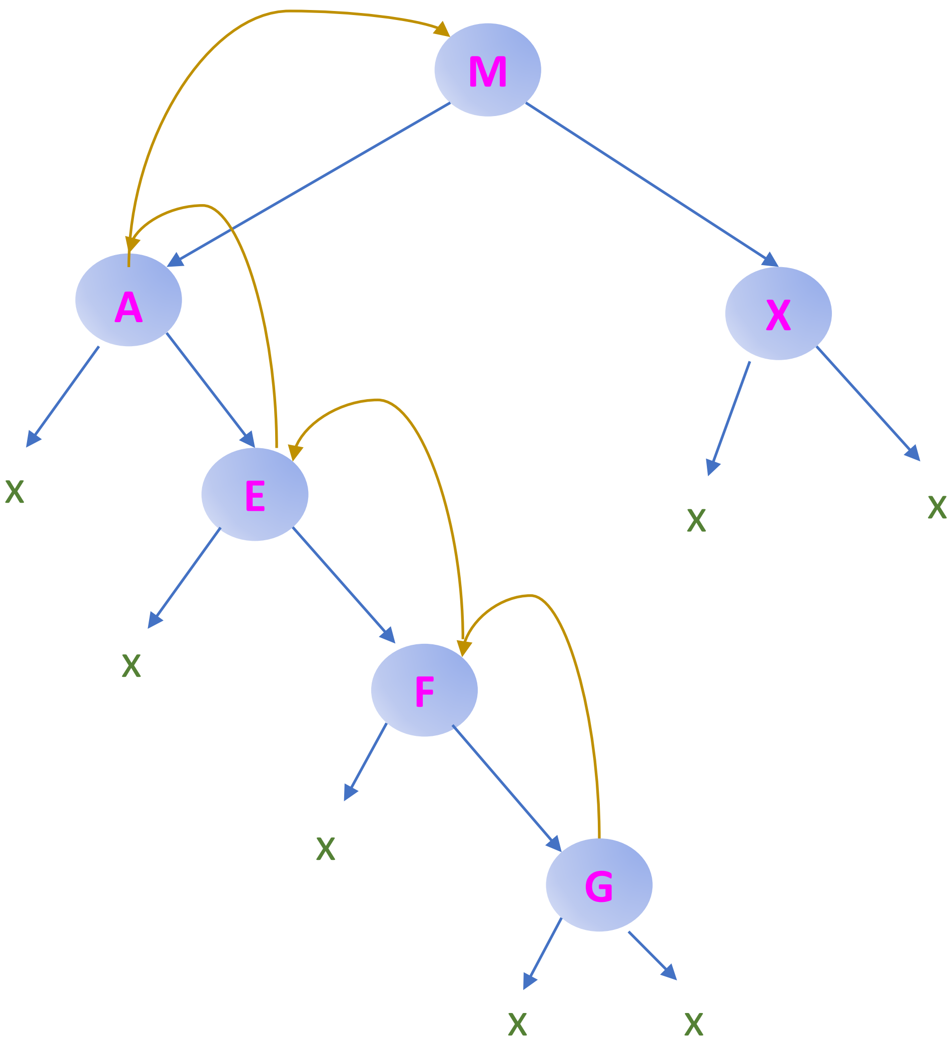
Here are three things Jason doesn’t like.

(1) Very unbalanced, say goodbye to $\approx \log_2 n$ search…. ☹

(2) Wasting 8 bytes per null pointer ☹

(3) Inorder successor of ‘G’ will take 4 stack pops to reach ☹

Here’s a BST



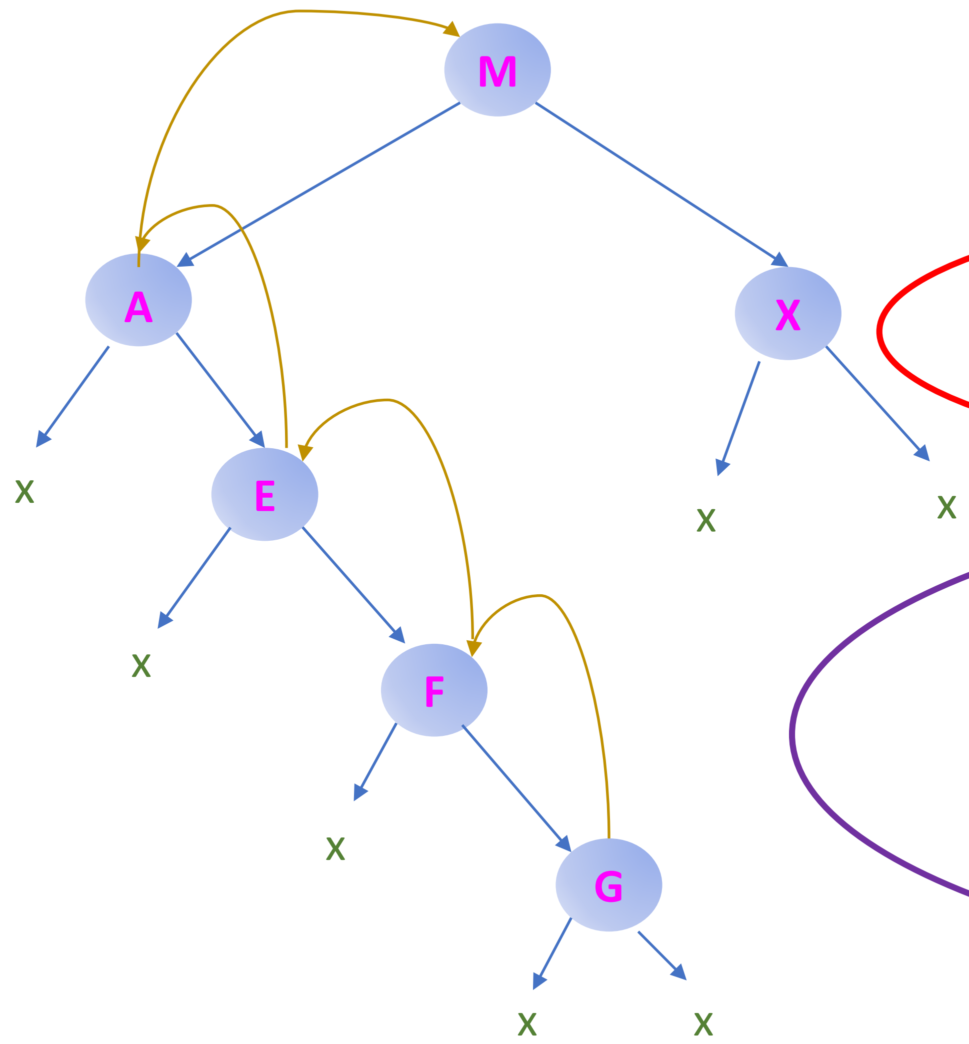
Here are three things Jason doesn’t like.

(1) Very unbalanced, say goodbye to $\approx \log_2 n$ search…. ☹

(2) Wasting 8 bytes per null pointer ☹

(3) Inorder successor of ‘G’ will take 4 stack pops to reach ☹

Here's a BST



Here are three things Jason doesn't like.

(1) Very unbalanced, say goodbye to $\approx \log_2 n$ search.... ☹

AVL / Splay / Red-Black Trees

(2) Wasting 8 bytes per null pointer ☹

(3) Inorder successor of 'G' will take 4 stack pops to reach ☹

Threaded binary search trees

Take-home message #4: ***improvements of basic structures***

- Even “boring” data structures like binary trees can be majorly improved.
- The linked representation of a binary tree wastes space (by default)
  - Complete binary trees can be represented as an array without NULL references in it…
  - Prefer this over 2^{h+1} null pointers, for a tree of height $h = 0,1,2, \ldots$

We talked storage; what about **operations**?

- Which among the following Java sum calculations will run **faster**?

(1)

```
long sum = 0;
for(int i = 0;  i < 1000; i++)
     for(int j = 0;  j < 1000; j++)
            sum += A[i][j];
```

(2)

```
long sum = 0;
for(int i = 0;  i < 1000; i++)
     for(int j = 0;  j < 1000; j++)
            sum += A[j][i];
```

(3)

They will both need about the same amount of CPU cycles to run.

We talked storage; what about **operations**?

- Which among the following Java sum calculations will run **faster**?

(1)

```
long sum = 0;
for(int i = 0;  i < 1000; i++)
     for(int j = 0;  j < 1000; j++)
            sum += A[i][j];
```

(2)

```
long sum = 0;
for(int i = 0;  i < 1000; i++)
     for(int j = 0;  j < 1000; j++)
            sum += A[j][i];
```

(3)

They will both need about the same amount of CPU cycles to run.

- This “classic” loop leverages row-major order!

We talked storage; what about **operations**?

- Which among the following Java sum calculations will run **faster**?

(1)

```
long sum = 0;
for(int i = 0;  i < 1000; i++)
     for(int j = 0;  j < 1000; j++)
            sum += A[i][j];
```

(2)

```
long sum = 0;
for(int i = 0;  i < 1000; i++)
     for(int j = 0;  j < 1000; j++)
            sum += A[j][i];
```

(3)

They will both need about the same amount of CPU cycles to run.

- This “classic” loop leverages row-major order!
- Time for another Java demo!

Any languages that do **column-major** order?

Yes
(which?)

No
(why?)

Any languages that do **column-major** order?

Yes (which?)	No (why?)

- MATLAB, GNU Octave, LISP…
- Semantics of M[i][j] the same
  - Only storage different, with whatever that implies about operations…

Any languages that do **column-major** order?

Yes (which?)

No (why?)

- MATLAB, GNU Octave, LISP…
- Semantics of M[i][j] the same
  - Only storage different, with whatever that implies about operations…
- Time for a MATLAB demo!

How row-major works…

- Here's a matrix M of 8 byte doubles:

M

2.33	8.145	67.9823	17.02	8.1	96.480	-25.05
5.67012	4.2	-100.02	-5600.1	32	25.789	89.01
521.23	-0.16	45.89	49	3.21	-1	-34.2
0.01	-32	34.578	-0.07	2.899	390245	87.1
567.98	89.56923	-897.6	90	0.45923	-67.45	25.6
-40.23	17.32	0	-6.43	100.56	0.01	5.23
702.97	14.56	56.79	100	12.6	-0.0001	-4.1

How row-major works…

double **M

	double M[0][1]	double M[0][2]		. . .			double M[0][6]
double* M[0]	2.33	8.145	67.9823	17.02	8.1	96.480	-25.05
double* M[1]	5.6701	4.2	-100.02	-5600.1	32	25.789	89.01
	521.23	-0.16	45.89	49	3.21	-1	-34.2
	0.01	-32	34.578	-0.07	2.899	390245	87.1
	567.98	89.56923	-897.6	90	0.45923	-67.45	25.6
	-40.23	17.32	0	-6.43	100.56	0.01	5.23
double* M[6]	702.97	14.56	56.79	100	12.6	-0.0001	-4.1

How row-major works…

double **M

double M[0][1] double M[0][2] …. double M[0][6]

double* M[0]	2.33	8.145	67.9823	17.02	8.1	96.480	-25.05
double* M[1]	5.6701	4.2	-100.02	-5600.1	32	25.789	89.01
	521.23	-0.16	45.89	49	3.21	-1	-34.2
	0.01	-32	34.578	-0.07	2.899	390245	87.1
	567.98	89.56923	-897.6	90	0.45923	-67.45	25.6
	-40.23	17.32	0	-6.43	100.56	0.01	5.23
double* M[6]	702.97	14.56	56.79	100	12.6	-0.0001	-4.1

double M[6][1] double M[6][2] double M[6][6]

So you need another column for pointers, and another cell for the initial pointer.

How row-major works…

double **M

double M[0][1] double M[0][2] . . . . double M[0][6]

double* M[0]	2.33	8.145	67.9823	17.02	8.1	96.480	-25.05
double* M[1]	5.6701	4.2	-100.02	-5600.1	32	25.789	89.01
	521.23	-0.16	45.89	49	3.21	-1	-34.2
	0.01	-32	34.578	-0.07	2.899	390245	87.1
	567.98	89.56923	-897.6	90	0.45923	-67.45	25.6
	-40.23	17.32	0	-6.43	100.56	0.01	5.23
double* M[6]	702.97	14.56	56.79	100	12.6	-0.0001	-4.1

double M[6][1] double M[6][2] double M[6][6]

So you need another column for pointers, and another cell for the initial pointer.

Assumed array size: 49 * 8 = 392 bytes . **Real size**: **49* 8 for data** + 7*8 for row references + 1*8 for initial reference = 456 bytes!

How row-major works…

double **M

double M[0][1] double M[0][2] . . . . double M[0][6]

double* M[0]	2.33	8.145	67.9823	17.02	8.1	96.480	-25.05
double* M[1]	5.6701	4.2	-100.02	-5600.1	32	25.789	89.01
	521.23	-0.16	45.89	49	3.21	-1	-34.2
	0.01	-32	34.578	-0.07	2.899	390245	87.1
	567.98	89.56923	-897.6	90	0.45923	-67.45	25.6
	-40.23	17.32	0	-6.43	100.56	0.01	5.23
double* M[6]	702.97	14.56	56.79	100	12.6	-0.0001	-4.1

double M[6][1] double M[6][2] double M[6][6]

That's ≈ 16.33% on top!

So you need another column for pointers, and another cell for the initial pointer.

Assumed array size: 49 * 8 = 392 bytes . **Real size**: **49* 8 for data** + 7*8 for row references + 1*8 for initial reference = 456 bytes!

double **M

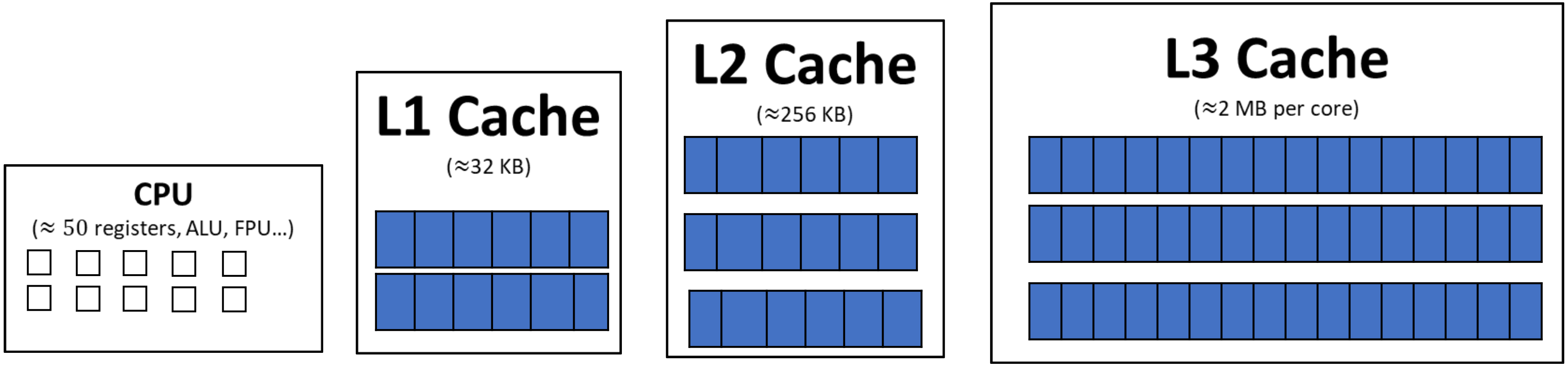
The bigger problem...

double* M[0]	2.33	8.145	67.9823	17.02	8.1	96.480	-25.05
double* M[1]	5.6701	4.2	-100.02	-5600.1	32	25.789	89.01
	521.23	-0.16	45.89	49	3.21	-1	-34.2
	0.01	-32	34.578	-0.07	2.899	390245	87.1
	567.98	89.56923	-897.6	90	0.45923	-67.45	25.6
	-40.23	17.32	0	-6.43	100.56	0.01	5.23
double* M[6]	702.97	14.56	56.79	100	12.6	-0.0001	-4.1

double **M

The bigger problem...

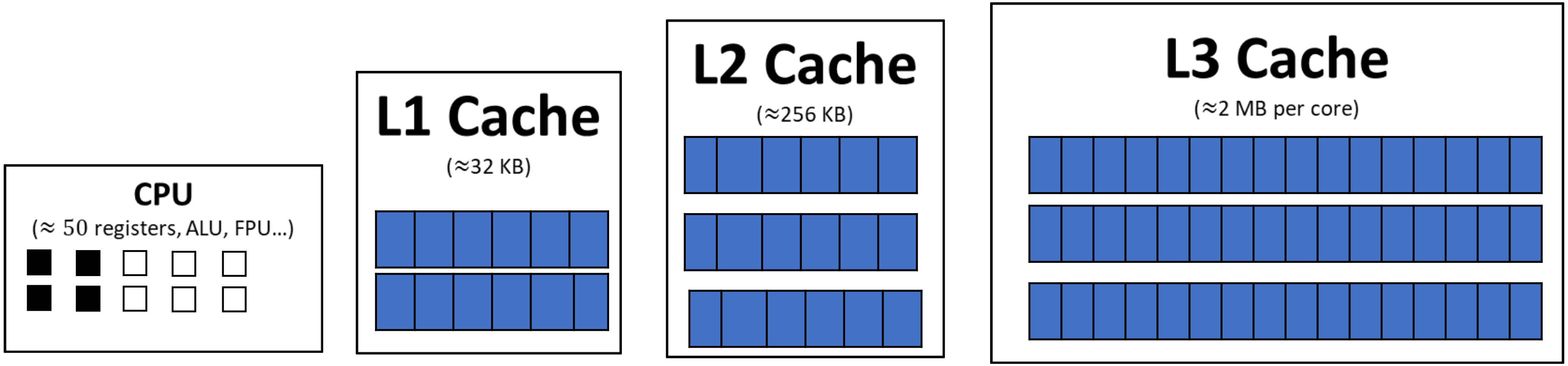
double* M[0]	2.33	8.145	67.9823	17.02	8.1	96.480	-25.05
double* M[1]	5.6701	4.2	-100.02	-5600.1	32	25.789	89.01
	521.23	-0.16	45.89	49	3.21	-1	-34.2
	0.01	-32	34.578	-0.07	2.899	390245	87.1
	567.98	89.56923	-897.6	90	0.45923	-67.45	25.6
	-40.23	17.32	0	-6.43	100.56	0.01	5.23
double* M[6]	702.97	14.56	56.79	100	12.6	-0.0001	-4.1



double **M

The bigger problem...

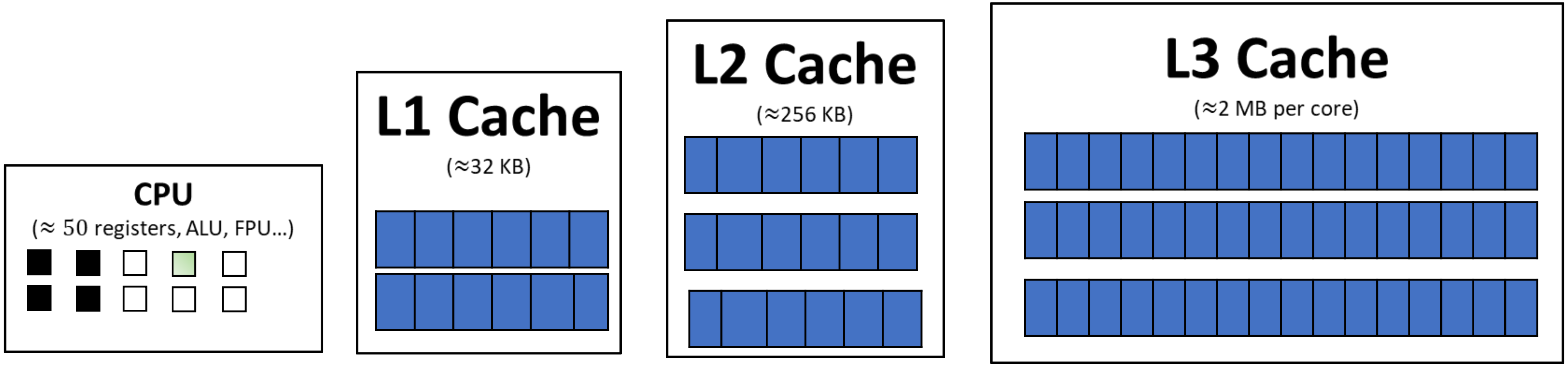
double* M[0]	2.33	8.145	67.9823	17.02	8.1	96.480	-25.05
double* M[1]	5.6701	4.2	-100.02	-5600.1	32	25.789	89.01
	521.23	-0.16	45.89	49	3.21	-1	-34.2
	0.01	-32	34.578	-0.07	2.899	390245	87.1
	567.98	89.56923	-897.6	90	0.45923	-67.45	25.6
	-40.23	17.32	0	-6.43	100.56	0.01	5.23
double* M[6]	702.97	14.56	56.79	100	12.6	-0.0001	-4.1



double **M

The bigger problem...

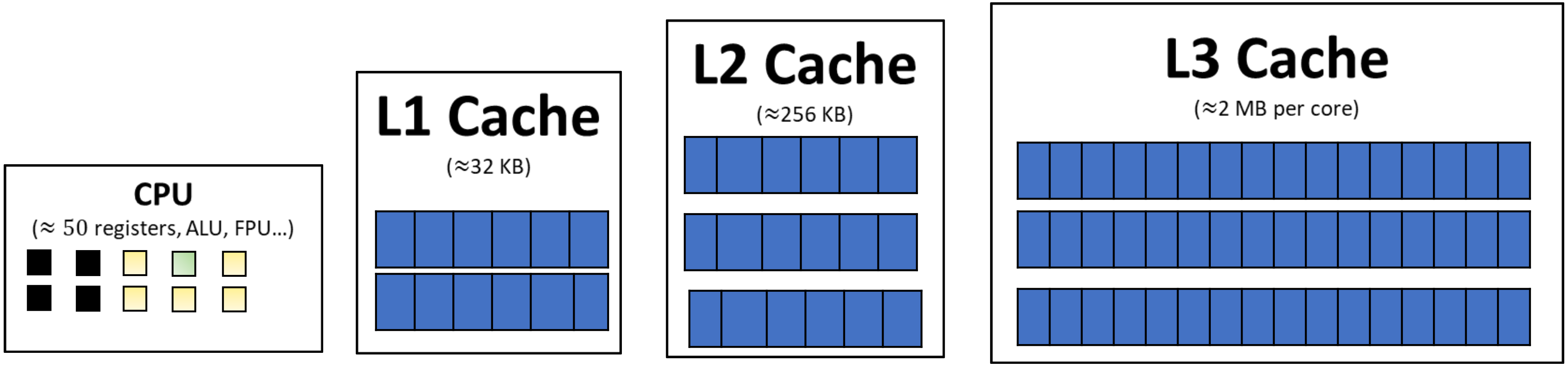
double* M[0]	2.33	8.145	67.9823	17.02	8.1	96.480	-25.05
double* M[1]	5.6701	4.2	-100.02	-5600.1	32	25.789	89.01
	521.23	-0.16	45.89	49	3.21	-1	-34.2
	0.01	-32	34.578	-0.07	2.899	390245	87.1
	567.98	89.56923	-897.6	90	0.45923	-67.45	25.6
	-40.23	17.32	0	-6.43	100.56	0.01	5.23
double* M[6]	702.97	14.56	56.79	100	12.6	-0.0001	-4.1



double **M

The bigger problem...

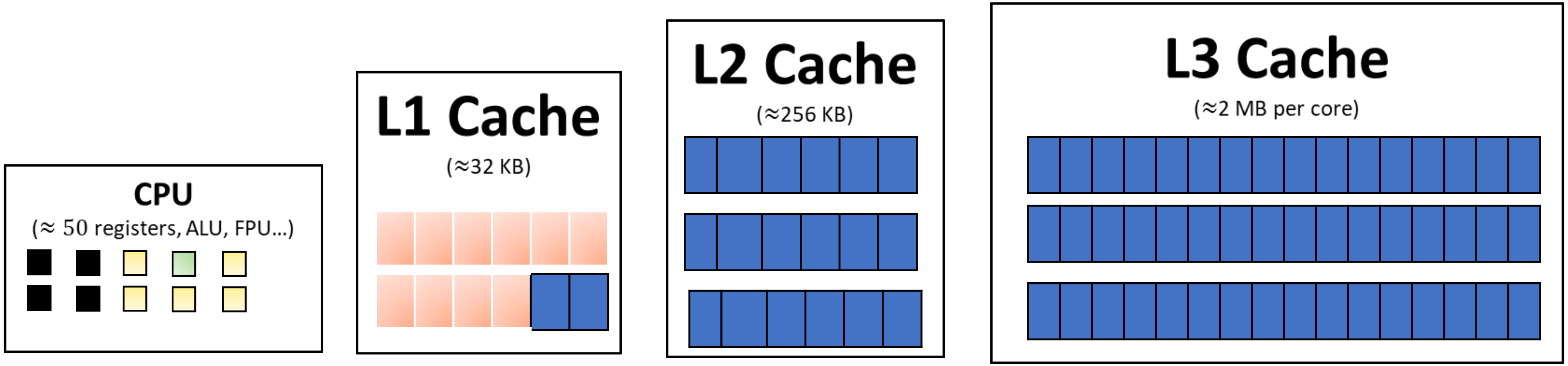
double* M[0]	2.33	8.145	67.9823	17.02	8.1	96.480	-25.05
double* M[1]	5.6701	4.2	-100.02	-5600.1	32	25.789	89.01
	521.23	-0.16	45.89	49	3.21	-1	-34.2
	0.01	-32	34.578	-0.07	2.899	390245	87.1
	567.98	89.56923	-897.6	90	0.45923	-67.45	25.6
	-40.23	17.32	0	-6.43	100.56	0.01	5.23
double* M[6]	702.97	14.56	56.79	100	12.6	-0.0001	-4.1



double **M

The bigger problem...

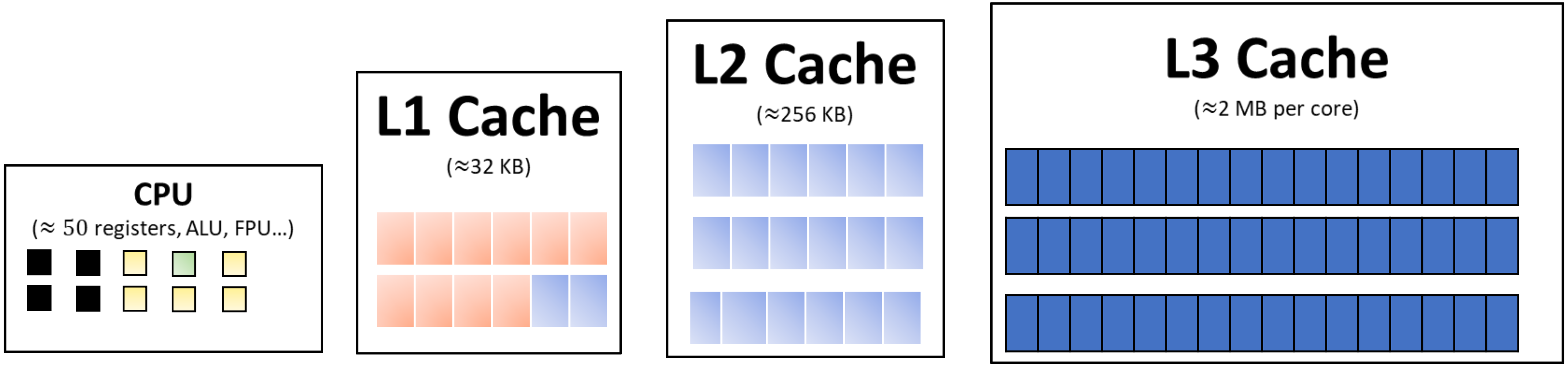
double* M[0]	2.33	8.145	67.9823	17.02	8.1	96.480	-25.05
double* M[1]	5.6701	4.2	-100.02	-5600.1	32	25.789	89.01
	521.23	-0.16	45.89	49	3.21	-1	-34.2
	0.01	-32	34.578	-0.07	2.899	390245	87.1
	567.98	89.56923	-897.6	90	0.45923	-67.45	25.6
	-40.23	17.32	0	-6.43	100.56	0.01	5.23
double* M[6]	702.97	14.56	56.79	100	12.6	-0.0001	-4.1



double **M

The bigger problem...

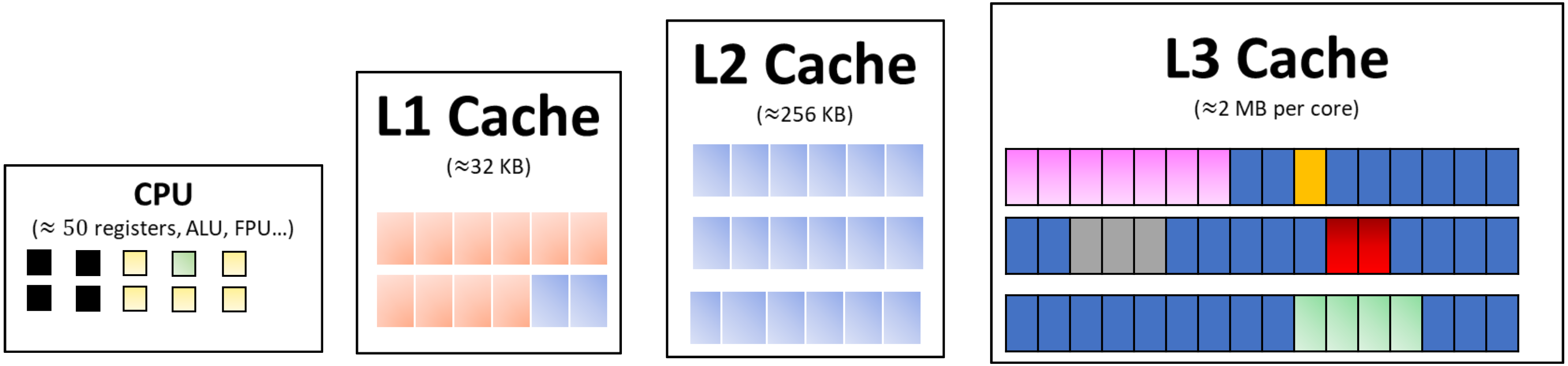
double* M[0]	2.33	8.145	67.9823	17.02	8.1	96.480	-25.05
double* M[1]	5.6701	4.2	-100.02	-5600.1	32	25.789	89.01
	521.23	-0.16	45.89	49	3.21	-1	-34.2
	0.01	-32	34.578	-0.07	2.899	390245	87.1
	567.98	89.56923	-897.6	90	0.45923	-67.45	25.6
	-40.23	17.32	0	-6.43	100.56	0.01	5.23
double* M[6]	702.97	14.56	56.79	100	12.6	-0.0001	-4.1



double **M

The bigger problem…

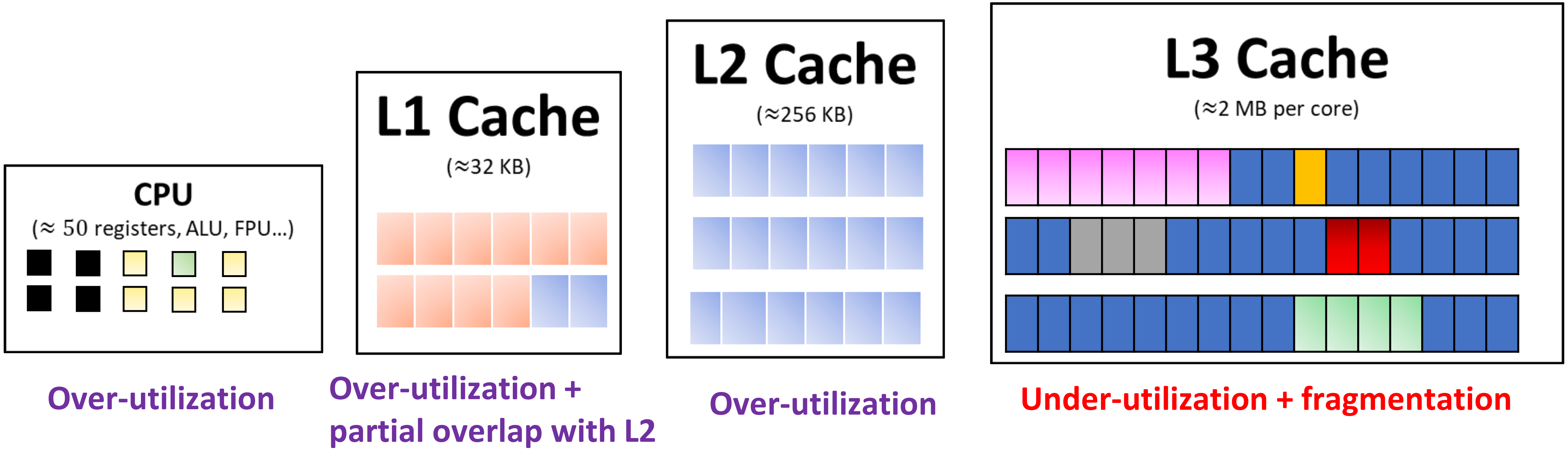
double* M[0]	2.33	8.145	67.9823	17.02	8.1	96.480	-25.05
double* M[1]	5.6701	4.2	-100.02	-5600.1	32	25.789	89.01
	521.23	-0.16	45.89	49	3.21	-1	-34.2
	0.01	-32	34.578	-0.07	2.899	390245	87.1
	567.98	89.56923	-897.6	90	0.45923	-67.45	25.6
	-40.23	17.32	0	-6.43	100.56	0.01	5.23
double* M[6]	702.97	14.56	56.79	100	12.6	-0.0001	-4.1



The bigger problem...

double **M

double* M[0]	2.33	8.145	67.9823	17.02	8.1	96.480	-25.05
double* M[1]	5.6701	4.2	-100.02	-5600.1	32	25.789	89.01
	521.23	-0.16	45.89	49	3.21	-1	-34.2
	0.01	-32	34.578	-0.07	2.899	390245	87.1
	567.98	89.56923	-897.6	90	0.45923	-67.45	25.6
	-40.23	17.32	0	-6.43	100.56	0.01	5.23
double* M[6]	702.97	14.56	56.79	100	12.6	-0.0001	-4.1



Take-home message #5:
programming language intricacies affect design

- Modern languages have HUGE and very complex compilers / interpreters, and teams of developers working at the lowest levels (as close to the hardware as possible).
  - If you want to be such a developer, the teaching staff applauds your choice.

Take-home message #5: ***programming language intricacies affect design***

- Modern languages have HUGE and very complex compilers / interpreters, and teams of developers working at the lowest levels (as close to the hardware as possible).
  - If you want to be such a developer, the teaching staff applauds your choice.
- Unless you know what you're doing, never attempt to do the compiler's job for it!
  - Modern compilers can do their job better than you can.
  - Optimization flags -01, -02, -03, -Wall, for gcc
  - JIT ("Just-in-time") compiling in the JVM does stuff like loop unrolling, optimized tail recursion…
  - `register` keyword in C!

What we'll cover

1. Building **dictionaries**: Abstract data structures that store <Key, Value> pairs
   - Balanced binary trees
   - Skiplist
   - B-Trees
   - Hashing (peering ***under*** under the `java.util.Hashmap` hood)

2. Data structures to support efficient operations on **string** data
   - Huffman coding
   - String compression (LZW)
   - Tries
   - Suffix Tries / Trees / Arrays

3. Multi-dimensional data structures to model 2D, 3D, …, ND and perform similarity tasks.
   - KD-Trees
   - Quadtrees: Fink's, P-R (classic, bucketed, loose), MX-CIF, Region

4. Elective topics:
   - Range Trees
   - Priority Search Trees
   - Binomial and Fibonacci heaps
   - MinHash for document similarity
   - Locality Sensitive Hashing (LSH)
   - Consistent Hashing

Notable things we **won't** cover (a lot)

- We won't talk a ***ton*** about B+ (B-plus) trees (extension of B-Trees to disk)
  - According to Nick Roussopoulos, the most successful data structure ever!

- Treaps (random balanced binary trees)
- Brent's and Gunnot-Monroe algorithms for self-organizing collision chain hash tables.

Assessment

- 5-6 projects on Java *(40%)*
  - Score from submit.cs unit tests
  - Manual inspection of code to verify grading for some methods.
  - Moss!
  - Please install OpenJDK for licensing reasons
- About 5 homeworks, every 2 /3 weeks *(15%)*
- Lecture worksheets, which will start Wednesday! *(5%)*
  - Short and sweet(?) questions on stuff we talk about on lecture
  - TA assistance / supervision
- 1 Midterm, see ELMS for date/time/place *(20%)*
- 1 Final, see ELMS for details *(20%)*

Webpages

- ELMS: Static information (syllabus), files and your gradebook.
  - Recordings, resources, programming project handouts, breakdown of lectures,…
- Piazza: We talk. Teaching staff announces future materials, resources, you post your questions.
- Submit server: You submit your projects. Refer to the programming project handouts for specifics of how you can do this.
- Jason’s GitHub: Skeleton code for your projects on my GitHub so that you can download it and fill in the class implementations.
  - Code demos will also be shared on the GitHub repo for you.
- Gradescope: You will be uploading your homeworks there, and we will be scanning in and grading worksheets and exams there as well. This means that you can see all of your answers ***and*** mistakes until the end of the semester!

Textbooks

- None *required,* all *recommended*

1. Shaffer, *Data Structures & Algorithm Analysis*

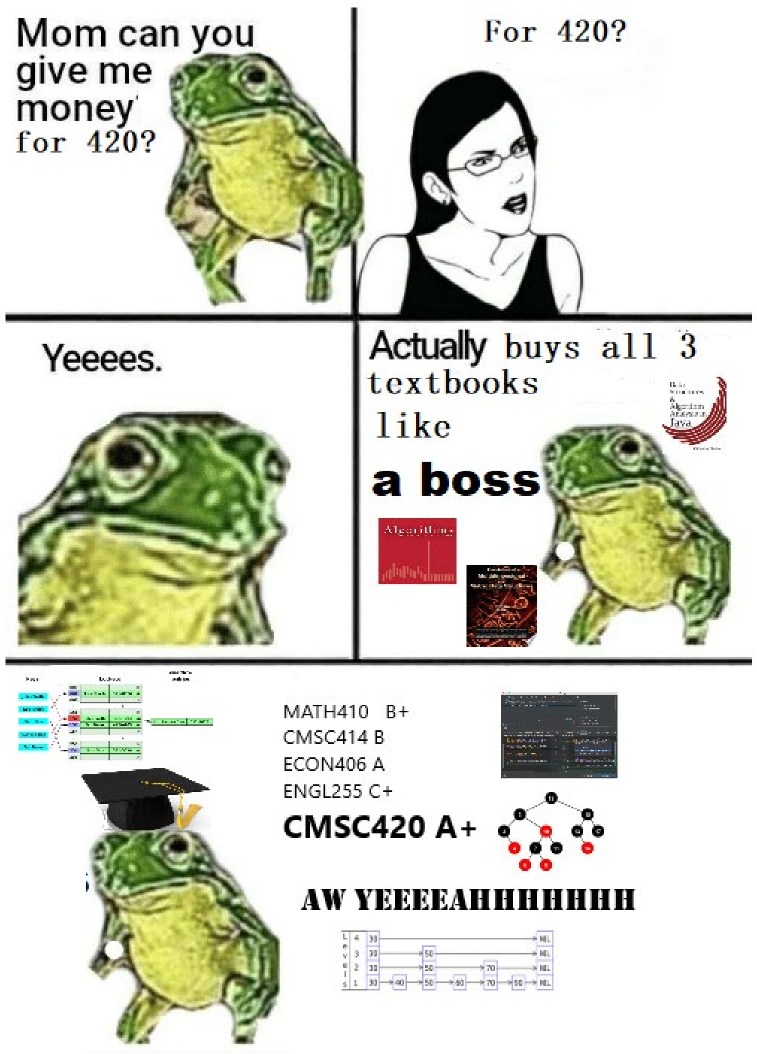
Freely available, Link on ELMS. Very broad, with both C++ and Java pseudocode versions printed. Has great detail on Hashing and Skiplists. Does ***not*** assume key rotations in B-Trees, which ***we*** will later on (don't worry, we will explain in time).

2. Sedgewick & Wayne, *Algorithms*, 5th ed. (4th is good too), Pearson

This is probably in the top 5 of books you want to read in the entire major. Spatial and temporal analysis using Java examples, very mature treatment of hashing, red-black trees, lots of paragraphs about APIs and programming practices.

3. Samet, *Multidimensional and Metric Data Structures,* Morgan Kaufman

If you like this course and you want to do some work (personal project, MSc, PhD, job,…) on Spatial Data Structures, you want this book. It is the holy book of its field.



AMA