CPSC 221
Basic Algorithms and Data Structures

June 1, 2015

Administrative stuff

Midterm exam is 2 days from today

Soon to be posted on Connect:

Solutions for Theory Assignment 1 (max marks is 18, not 26)

Solutions for in-class exercises that haven’t been posted yet

Administrative stuff

Midterm exam is 2 days from today

There will be no labs this week.

But the TAs will be in our lab room (X350) holding office hours during the Tuesday lab time and the Wednesday lab time that’s right before class.

Administrative stuff

Midterm exam is 2 days from today

- 90 minutes
- You may bring three (3) 8.5 x 11 inch sheets of paper with your notes, double-sided, any format
- You may not bring magnifying glasses, microscopes, weird goggles, Google glass, etc.
- Exam will start shortly after beginning of class
- Exam will take place in SWNG 121 and SWNG 122

Administrative stuff

Midterm exam is 2 days from today

Exam will take place in SWNG 121 and SWNG 122

Student numbers 11035136 to 38407136 write exam in SWNG 121

Student numbers 38439121 to 86867892 write exam in SWNG 122

Administrative stuff

Midterm exam is 2 days from today

Exam protocol:

No electronics, no books

Pen or pencil

Make sure you sit with an empty chair or aisle on either side of you.

Administrative stuff

Midterm exam is 2 days from today

Exam protocol:

You can’t leave the room during the first 30 minutes of the exam.

After that, washroom breaks are escorted by TA and only one at a time.

If you arrive more than 30 minutes after the start of the exam, you will not be permitted to write the exam.

Administrative stuff

Midterm exam is 2 days from today

Exam protocol:

If you are sick on exam day, don't write the exam. Bring us documentation from your physician. We'll count your final exam for both midterm and final exam marks.

If you write the exam, you own the mark, regardless of whether you were sick. You don't get to write the exam and then tell us it shouldn't count because you were ill.

Questions about exam logistics?

B	F	G	D	A	C	E
0	1	2	3	4	5	6

Given this array and a character, how do you determine if the character is in the array?

```
for (i = 0; i < array.length; i++)
{
   if array[i] == target_char
      return true;
}
return false;
```

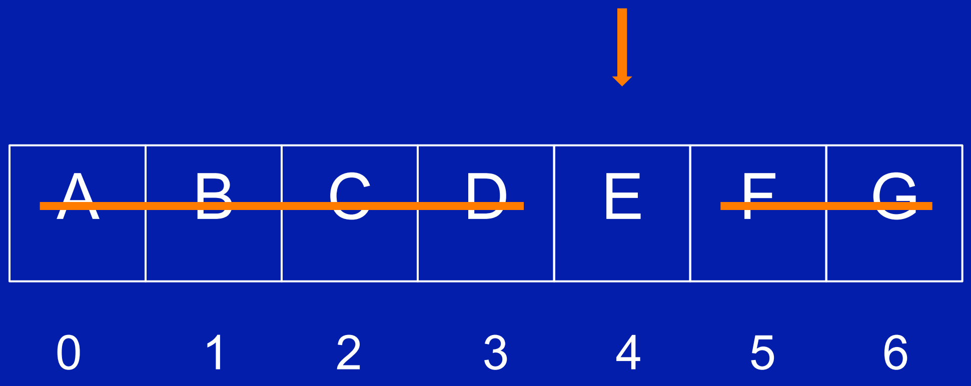
B	F	G	D	A	C	E
0	1	2	3	4	5	6

'(B F G D A C E)

Can we make the search substantially more efficient?

A	B	C	D	E	F	G
0	1	2	3	4	5	6

Describe a search algorithm that could take advantage of this sorting. Where would you start the search?



How many comparisons to find E?
Is it here? No.
Is it here? No.
Is it here? Yes.

Best case: 1 comparison (finding D on the first try)
Worst case: 3 comparisons

A	B	C	D	E	F	G	L	M	N	P	Q	R	W	X
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Can you figure out the relationship between the size of the array and the worst case search performance?

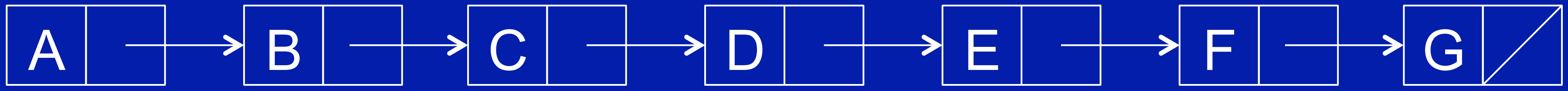
worst case comparisons = 3	array size = 7
worst case comparisons = 4	array size = 15
worst case comparisons = 5	array size = 31

$$T(n) = \text{some } f(n)$$

$$f(n) = \log_2(n+1)$$

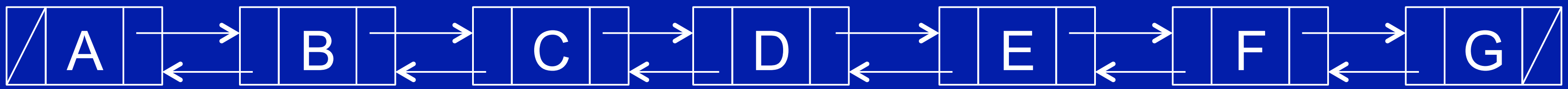
'(A B C D E F G)

Do we gain the same benefit by sorting the elements of this list?

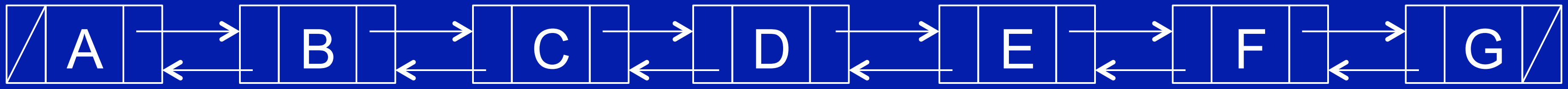


This is a singly linked list (sometimes called a one-way linked list). It's the kind of list that Racket provides.

Do you see a problem now with getting to the middle of the list in one simple step? With the sorted array, finding the midpoint of the unsearched remainder of the list was simple arithmetic. Now it requires traversing half the links, and if we get to the middle how do we go left? And wouldn't we have already searched to the left anyway? This isn't promising.



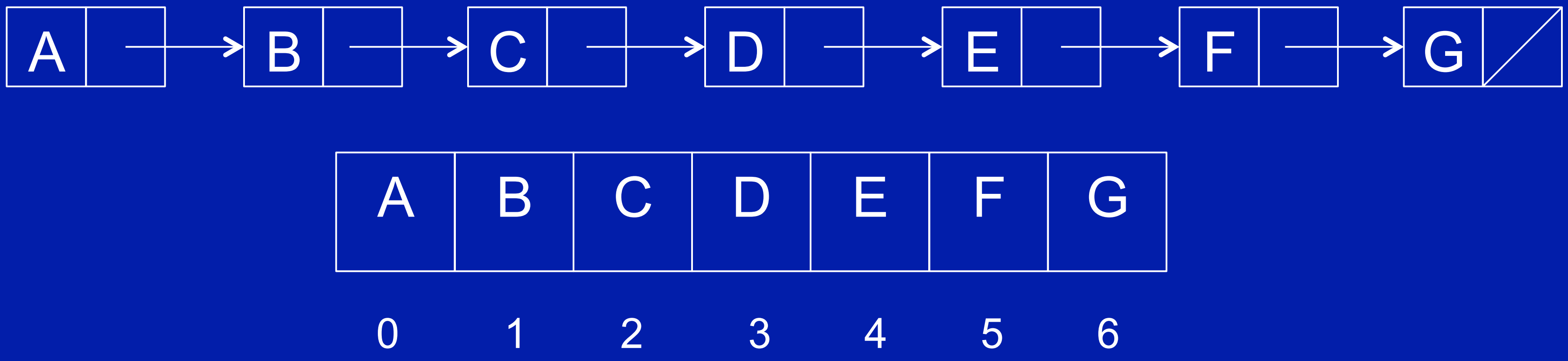
We could construct our own doubly linked list that would allow us to traverse the structure in both directions. How could you do that in Racket?



We could construct our own doubly linked list that would allow us to traverse the structure in both directions. How could you do that in Racket? Remember the `struct`?

But we still have the issue of no simple way to get to the middle of the remaining unsearched list, so this isn’t promising either.

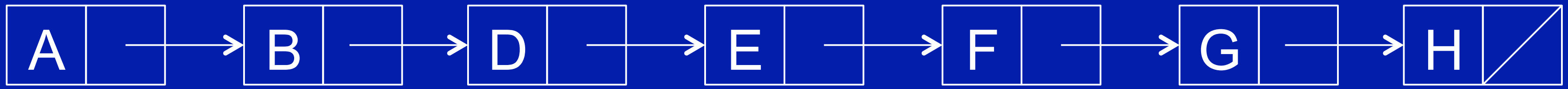
Is there some other linked structure that would facilitate the “telephone book” search? Yes. You’ve seen it in CPSC 110, and we’ll see it in CPSC 221 in the days ahead. Patience grasshopper.



Let's go back to comparing the sorted array to the sorted singly linked list. Now which one is better?

A	B	D	E	F	G	H
0	1	2	3	4	5	6

If you were thinking that arrays are the way to go here, think again. What if you want to insert a new element, C, in this sorted array? What's the algorithm for insertion? What if the array only has room for 7 elements?



What if you want to insert a new element, C, in this sorted list? How do you do the insertion in this case?

Questions like “which data structure is better?” or “which algorithm is better?” may not have absolute answers.

“It depends...” may be the beginning of many such analyses.

The point of this whole discussion is just to give you a sense for what this class is all about.

Procedures and algorithms

There's a general commonly-used definition of an algorithm. Here's one version of many:

An algorithm is

- a finite procedure
- written in a fixed symbolic vocabulary
- governed by precise instructions
- moving in discrete steps, 1, 2, 3, ...
- whose execution requires no insight, cleverness, intuition, intelligence, or perspicuity
- and that sooner or later comes to an end

David Berlinski in The Advent of the Algorithm

What’s a data structure again?

- Data structure (street definition): how to organize your data to get the results you want, along with the supporting algorithms
- Any storage structure for data objects, along with the operations that are specific to the data structure
- There’s a commonly used computing name for a combination of data structure and associated operations. You may have heard the term in CPSC 210 if not in CPSC 110...

Abstract data type

An abstract data type or ADT is a combination of

- a collection of data items
- a set of operations on those items

It’s abstract because it lets us ignore the implementation details of how the data is organized or how the operations really work.

Abstract data type

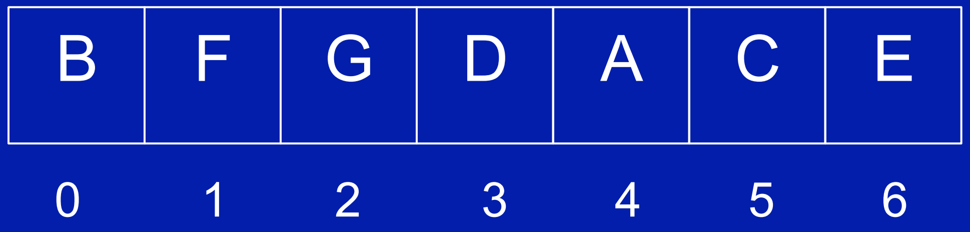
An abstract data type or ADT is a combination of

- a collection of data items
- a set of operations on those items

Formally, an ADT is a mathematical description of an object and the set of operations on the object

In practice, an ADT is the interface of a data structure, without any information about the implementation

Array and ArrayList



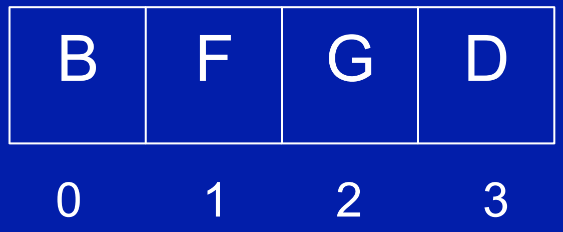
The array in Java is a very low-level ADT. Its logical model basically matches the underlying memory. And storing a value in an array requires that you know about array addressing. That's not much in the way of abstraction.

Arrays have fixed size. They can't grow or shrink.

You can't insert into or delete from the middle of an array.

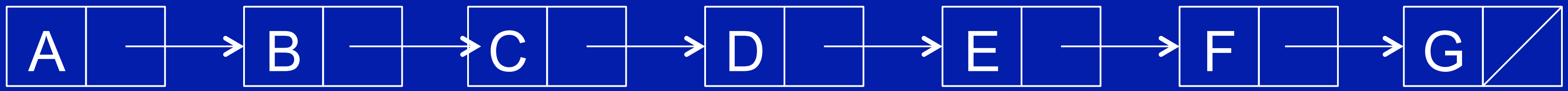
Within those limitations, they're very efficient.

Array and ArrayList



Java provides the ArrayList ADT for more flexibility and easier programming. You can add things and delete things anywhere. The ArrayList can grow in size. Great features for programmers, but there are hidden costs, as we talked about a little on Monday.

Linked Lists

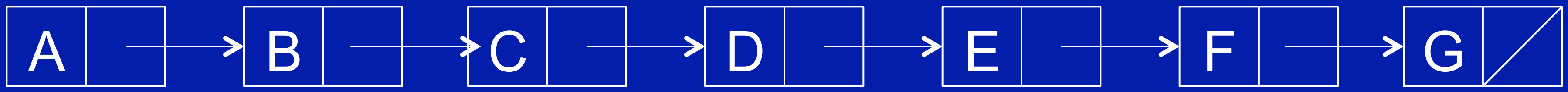


A linked list is an abstract data type for representing data as a collection of linked nodes. Each node contains some information and at least one pointer to in an adjacent node.

Instead of having a single monolithic structure like an array in which adjacent items are physically next to each other, the ordering in a list is represented locally: any node knows only about the next node (via the pointer).

List elements may not be represented contiguously in memory, and they're likely not represented in the order that the elements occur in the list.

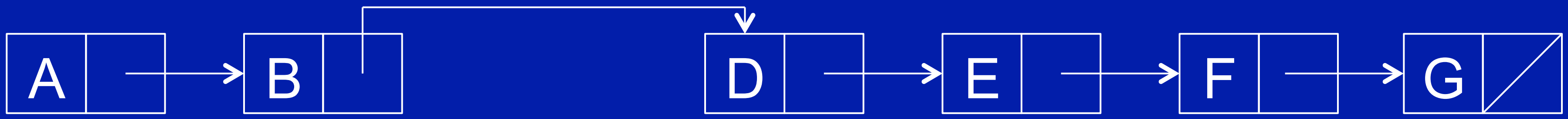
Linked Lists



With a singly linked list, addition to the front of the list (i.e., the cons) is simple and efficient. It takes the same amount of time every time, no matter how big the list is. (That's called constant time.)

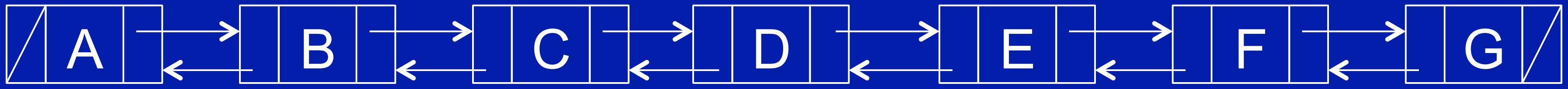
We showed you on Monday how to insert something into the middle of the list. We won't repeat that. But the time to insert something in the middle of the list is, on average, going to be proportional to the size of the list, no?

Linked Lists



Deletion from a singly linked list is similarly straightforward. Assuming you know what the information is you want to delete (C, in this case), but don't know where it is, traverse list until you find the info to be deleted, while retaining a pointer to the previous node along the way, and copy the link leading away from the current node (C) into the link leading away from the previous node (B). When the link node containing C is returned to free memory, the list looks like this.

Linked Lists



Your textbook says that singly linked lists have limitations:

- you can insert a node only after a node you have a pointer to
- you can remove a node only if you have a pointer to its predecessor node
- you can traverse only in one direction

The doubly linked list solves those problems by adding an additional link to each node, such that each node has a link to its predecessor and its successor in the sequence.

Stack

The stack is another container for a sequential collection of data.

The stack ADT permits data to be added or deleted only at one end of the sequence. It's easily implemented, and is great for storing "postponed obligations" or "postponed computations".

Real world analogies:

Spring-loaded dish dispenser in a cafeteria

Pez candy dispenser

Stack

Only the “top” of a stack is accessible, so the stack ADT requires very few operations:

push(item)	-	add to top of stack
pop()	-	remove from top of stack
top()	-	return item at top without removing it
empty()	-	return true if stack empty else false
size()	-	return number of items on stack

Stack

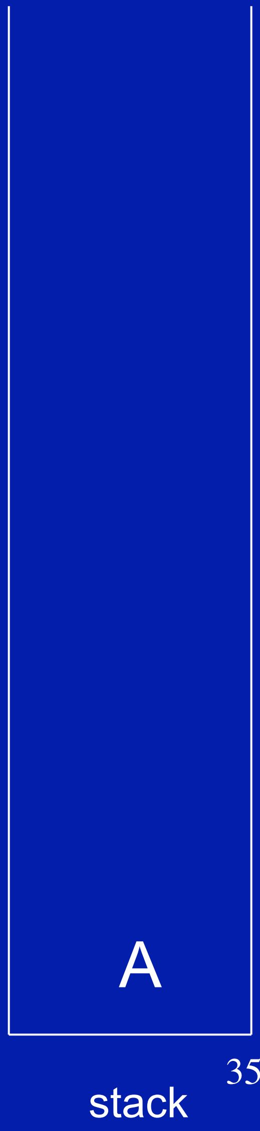
A stack reverses the order of arrival. This is called last-in-first-out or LIFO behaviour.

Stacks have been widely used in computing.

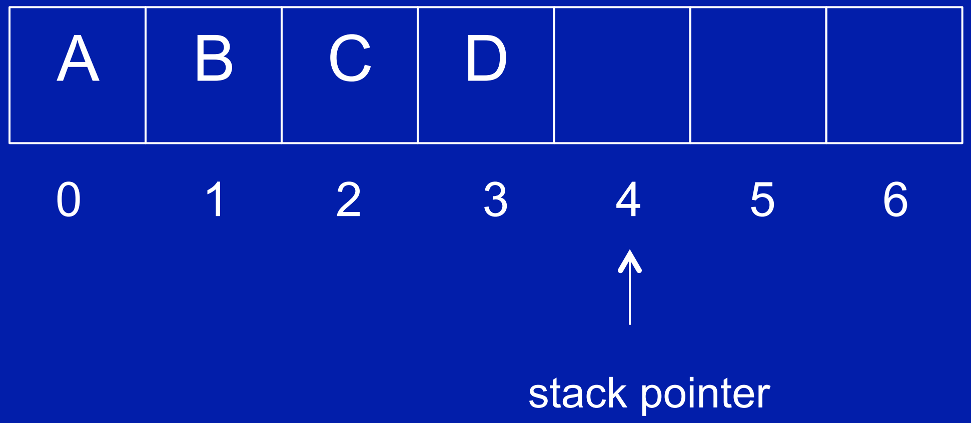
A compiler or interpreter may use a stack to parse expressions.

During program execution, a stack is used to keep track of procedure calls and parameters. It's how recursion is handled.

A web browser's 'back' button. Undo...



Stack implementation

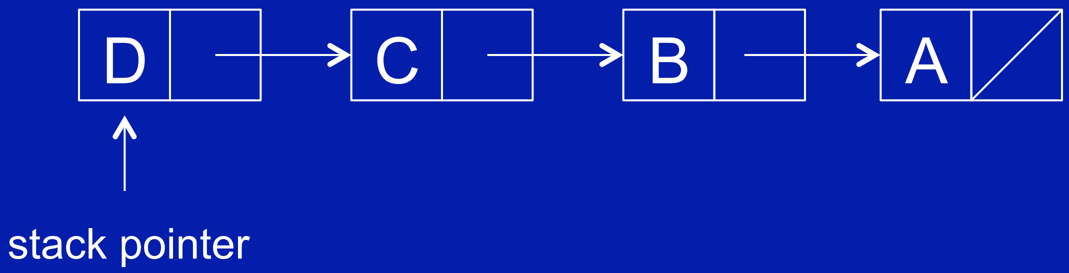


A stack can be implemented as a fixed array in memory.

What was the first thing pushed on the stack? What was the last thing? What will be the first thing popped off the stack?

What are the details involved in pushing something on the stack? In popping something off the stack?

Linked Lists



A stack can also be implemented as a linked list.

Now what are the details involved in pushing something on the stack? In popping something off the stack?

Queue

The queue is yet another container for a sequential collection of data. Like the stack, it's great for storing "postponed obligations" or "postponed computations".

Unlike the stack, the queue ADT permits data to be added only at one end of the sequence and removed from the other end.

Real world analogies:

The queue for a bus, or tickets at the movies, or Timmies, or ...

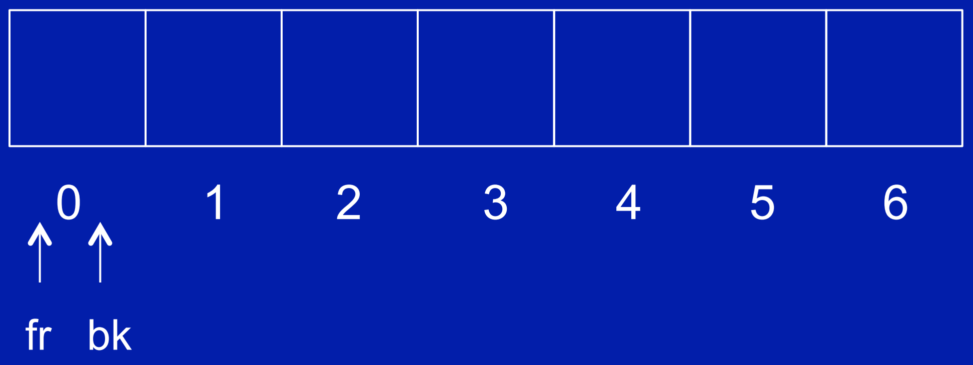
The waitlist to enrol in a course

Queue

Queue operations:

enqueue(item)	-	add to back of queue
dequeue()	-	remove from front of queue
front()	-	return item at front without removing it
empty()	-	return true if queue empty else false
size()	-	return number of items in queue

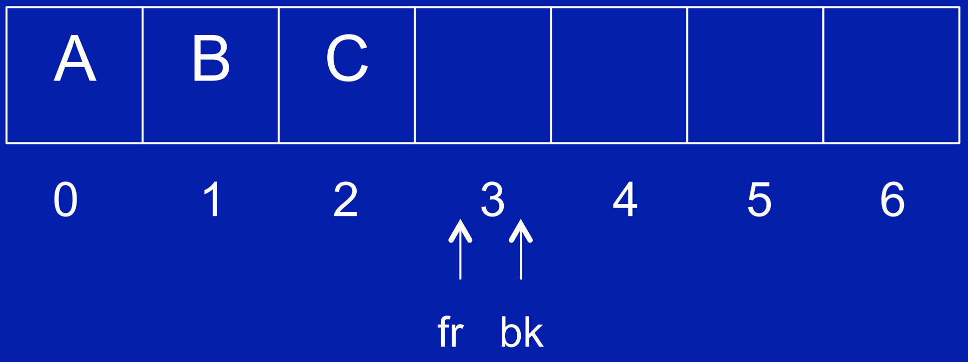
Queue implementation



You could implement a queue in a couple of different ways. What might they be?

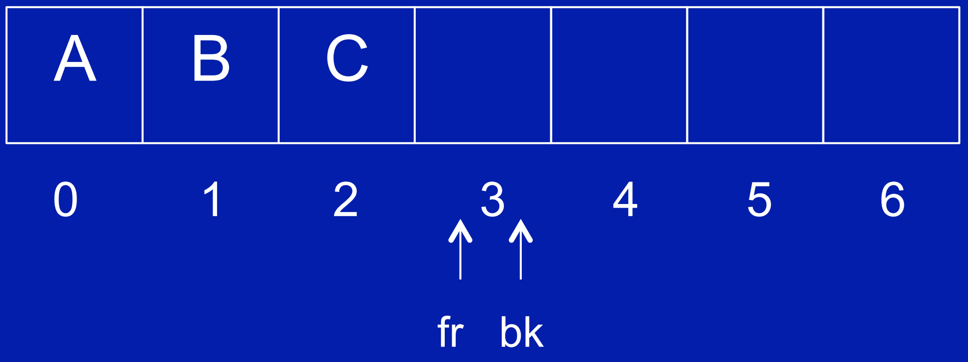
Just like the stack, we could use an array or a linked list for implementation. Here’s an example of some serious queue action.

Queue implementation



Now the queue is empty, but it only has about half the available space that it did when we started. How can we deal with that problem?

Queue implementation

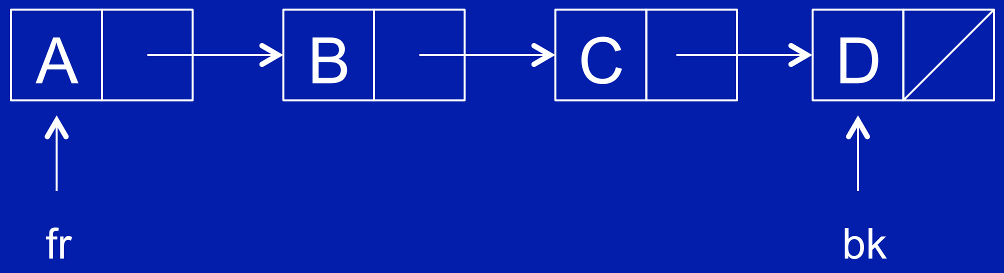


One approach would be to implement a circular array. When the back pointer is incremented to point beyond the array, reset the back pointer to 0. Same with the front pointer.

```
enqueue(x)
{
  array[bk] = x
  bk = (bk + 1) % size
}
```

```
dequeue()
{
  fr = (fr + 1) % size
}
```

Queue implementation



A queue can also be implemented as a linked list.

```
enqueue(x)
{
  if (empty())
     fr = bk = new Node(x)
  else
     bk->next = new Node(x)
     bk = bk->next
}
```

```
dequeue()
{
  if (not(empty()))
     temp = fr
     fr = fr->next
     delete temp
}
```

These bits of pseudocode should give you an idea of how to write similar algorithms for the stack ADT from Wednesday. A home exercise for you.

Analyzing runtime

Iterative Fibonacci:

```
old2 = 1
old1 = 1
for (i = 3; i <= n; i++)
{
   result = old2 + old1
   old1 = old2
   old2 = result
}
```

How many operations when this algorithm is run?

3 going into the loop

6 each time the loop is executed

the loop is executed n - 2 times

1 going out of the loop

3 + (6)(n-2) + 1

If we're ignoring details, does it make sense to be so precise? It's educational now, but later we'll see that we can do this more simply and ignore the details

Big-O notation

T(n) is O(n)

T(n) is our shorthand for the runtime of the function being analyzed.

The O in O(n) means “order of magnitude”, so Big-O notation is clearly not precise. It’s a formal notation for the approximation of the time (or space) requirements of running algorithms.

In the current context, we might say “the run time of iterative Fibonacci is Oh of n” or just “iterative Fibonacci is Oh of n”.

Big-O arithmetic

There's a formal mathematical definition for Big-O that we're obliged to discuss:

T(n) is O(f(n)) if there are two positive constants, n_0 and c, and a function f(n) such that cf(n) >= T(n) for all n > n_0

Does your head hurt yet?

Or, as your book puts it, as n gets sufficiently large (larger than n_0), there is some constant c for which the processing time will always be less than or equal to cf(n), so cf(n) is an upper bound on the performance. The performance will never be worse than cf(n) and may be better.

Big-O arithmetic

One more time. Here’s what this

T(n) is O(f(n)) if there are two positive constants, n_0 and c, and a function f(n) such that cf(n) >= T(n) for all n > n_0

really means in a practical sense:

If you want to show that T(n) is O(f(n)) then find two positive constants, n_0 and c, and a function f(n) that satisfy the constraints above. For example...

Big-O arithmetic

For our iterative Fibonacci example, $T(n) = 6n - 8$. We want to show that $T(n)$ is $O(n)$. So we set up our inequality like this:

$$\begin{aligned} 6n - 8 &<= cn \\ 6n &<= cn + 8 \\ n &<= cn/6 + 8/6 \end{aligned}$$

Now pick some value for c. Let’s use 6 to cancel the denominator.

$$\begin{aligned} n &<= 6n/6 + 8/6 \\ n &<= n + 4/3 \end{aligned}$$

Big-O arithmetic

$$n <= 6n/6 + 8/6$$
$$n <= n + 4/3$$

Now we pick some value for n_0, substitute it for n, and see if the inequality holds. Hmmm, let's try 1.

$$1 <= 1 + 4/3$$

1 is always less than or equal to 1 + 4/3, so in choosing c = 6 and n_0 = 1, we have shown that 6n - 8 is O(n) because 6n - 8 <= 6n for n >= 1. T(n) is O(n). Yippee!

Big-O arithmetic

Examining some code, we determine that $T(n) = 3n^2 + 6n$ and we think that $T(n)$ is $O(n^2)$. How do we prove it?

$$\begin{aligned} 3n^2 + 6n &<= cn^2 \\ 3n(n + 2) &<= cn^2 \\ n + 2 &<= cn/3 \\ n &<= cn/3 - 2 \end{aligned}$$

Now we let $c = 3$ to cancel the denominator. That gives us $n <= n - 2$. That won't work. Let's try $c = 6$, a multiple of 3, that will also simplify things by cancelling the denominator. That gives $n <= 2n - 2$. That's much more promising...

Big-O arithmetic

Examining some code, we determine that $T(n) = 3n^2 + 6n$ and we think that $T(n)$ is $O(n^2)$. How do we prove it?

$$3n^2 + 6n <= cn^2$$
$$3n(n + 2) <= cn^2$$
$$n + 2 <= cn/3$$
$$n <= cn/3 - 2$$
$$n <= 2n - 2 \text{ with } c = 6$$

If we let $n_0 = 1$ and substitute it for n, we get

$$1 <= 2 - 2$$
$$1 <= 0$$

That won’t do.

Big-O arithmetic

Examining some code, we determine that $T(n) = 3n^2 + 6n$ and we think that $T(n)$ is $O(n^2)$. How do we prove it?

$$3n^2 + 6n <= cn^2$$
$$3n(n + 2) <= cn^2$$
$$n + 2 <= cn/3$$
$$n <= cn/3 - 2$$
$$n <= 2n - 2 \text{ with } c = 6$$

If we let $n_0 = 2$ and substitute it for n, we get

$$2 <= 4 - 2$$
$$2 <= 2$$

That will do nicely.

Big-O arithmetic

Examining some code, we determine that $T(n) = 3n^2 + 6n$ and we think that $T(n)$ is $O(n^2)$. How do we prove it?

$$
\begin{aligned}
3n^2 + 6n &<= cn^2 \\
3n\,(n + 2) &<= cn^2 \\
n + 2 &<= cn/3 \\
n &<= cn/3 - 2 \\
n &<= 2n - 2 \text{ with } c = 6
\end{aligned}
$$

Choosing $c = 6$ and $n_0 = 2$, we have shown that $3n^2 + 6n$ is $O(n^2)$ because $3n^2 + 6n <= 6n^2$ for $n >= 2$.

Big-O arithmetic

The Big-O definition simply(?) says that there is a point n_0 such that for all values of n that are past this point, T(n) is bounded by some multiple of f(n). Thus, if the running time T(n) of an algorithm is $O(n^2)$, we are guaranteeing that at some point we can bound the running time by a quadratic function (a function whose high-order term involves n^2).

Big-O says there's a function that is an upper bound to the worst case performance for the algorithm. Big-O is sort of like less than or equal to when growth rates are being considered.

What exactly are we trying to do?

Sometimes it’s called *complexity analysis*.

Complexity = the rate at which the storage or time grows as a function of the problem size.

The absolute or real growth depends on the machine used to execute the program, the compiler used to construct the program, the cost of individual operations, and many other factors. We would like to have a way of describing the inherent complexity of a program (or piece of a program), independent of machine/compiler considerations. This means that we must not try to describe the absolute time or storage needed. We must instead concentrate on a "proportionality" approach, expressing the complexity in terms of its relationship to some known function. This type of analysis is also known as *asymptotic analysis*.

What exactly are we trying to do?

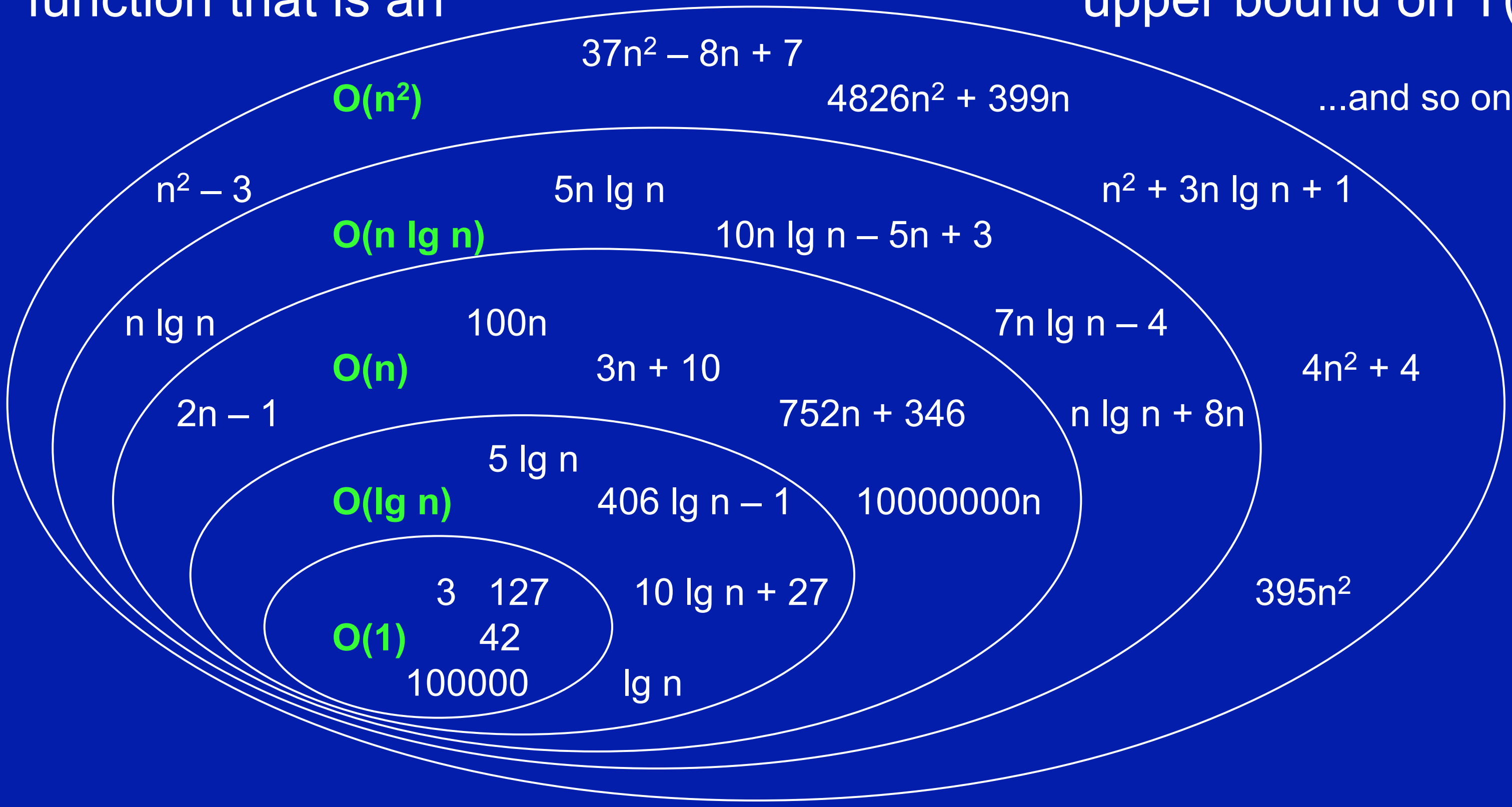
Asymptotic analysis means that we're studying the behaviour of functions f(n) as n gets very large, approaching ∞.

Asymptotic analysis is based on the idea that as the problem size grows, the complexity can be described as a simple proportionality to some known function. This idea is incorporated in the Big-O notation for asymptotic performance. When we say T(n) is $O(n^2)$, we mean that the growth of the run time is proportional to n^2, and that there is at least one specific function, cn^2, that is an upper bound to the growth of the run time.

We're just trying to classify algorithms into these standard complexity categories. We're not trying to say which algorithm among many within the category is better...that's where the constants that we ignore while doing the analysis come back into play again.

What exactly are we trying to do?

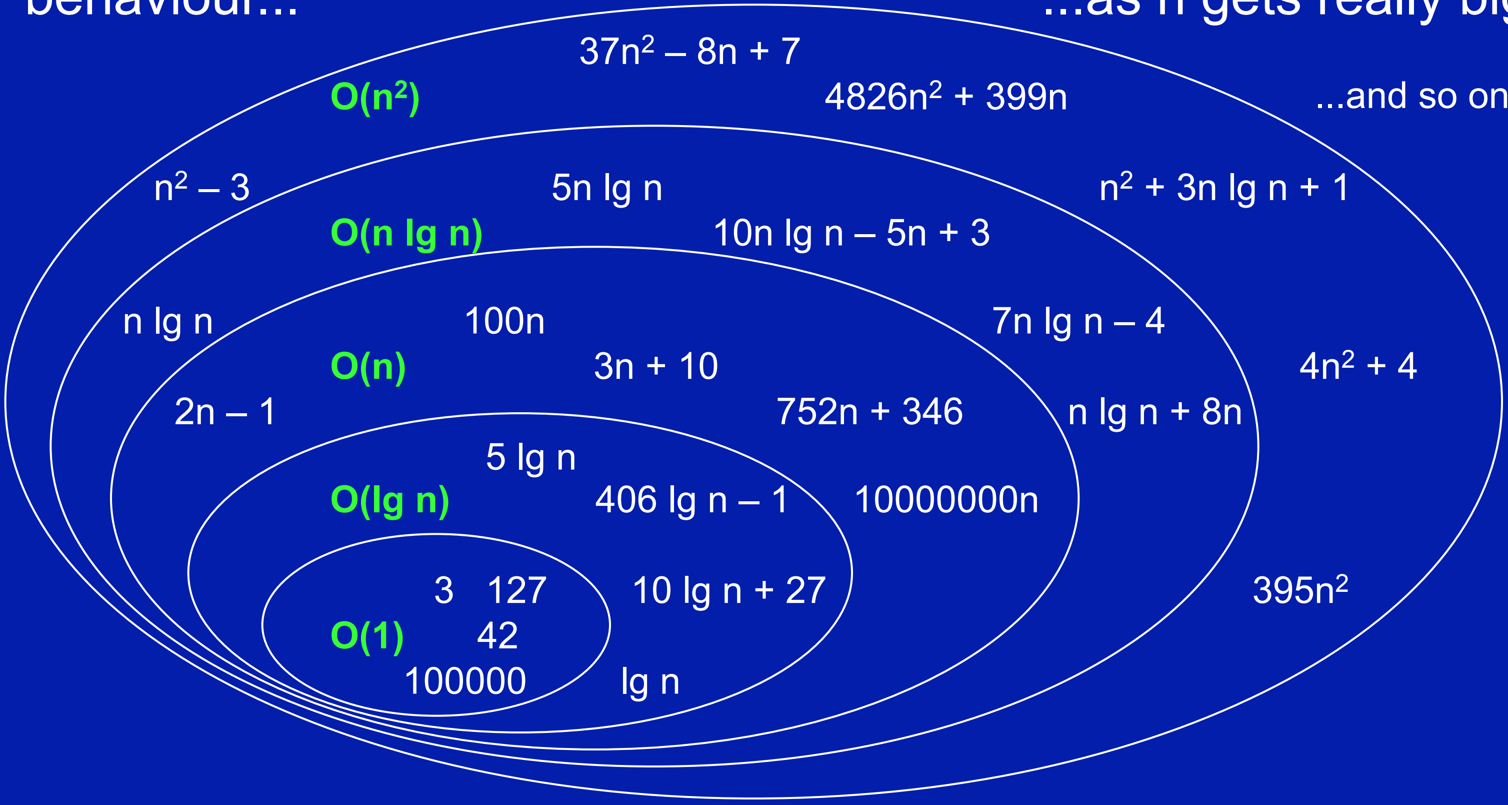
For any f(n) with run time T(n), we're trying to categorize the run time behaviour by placing it in the smallest possible group that contains a function that is an upper bound on T(n)



O(g(n)) is the set of functions with smaller or same order of growth as g(n)

What exactly are we trying to do?

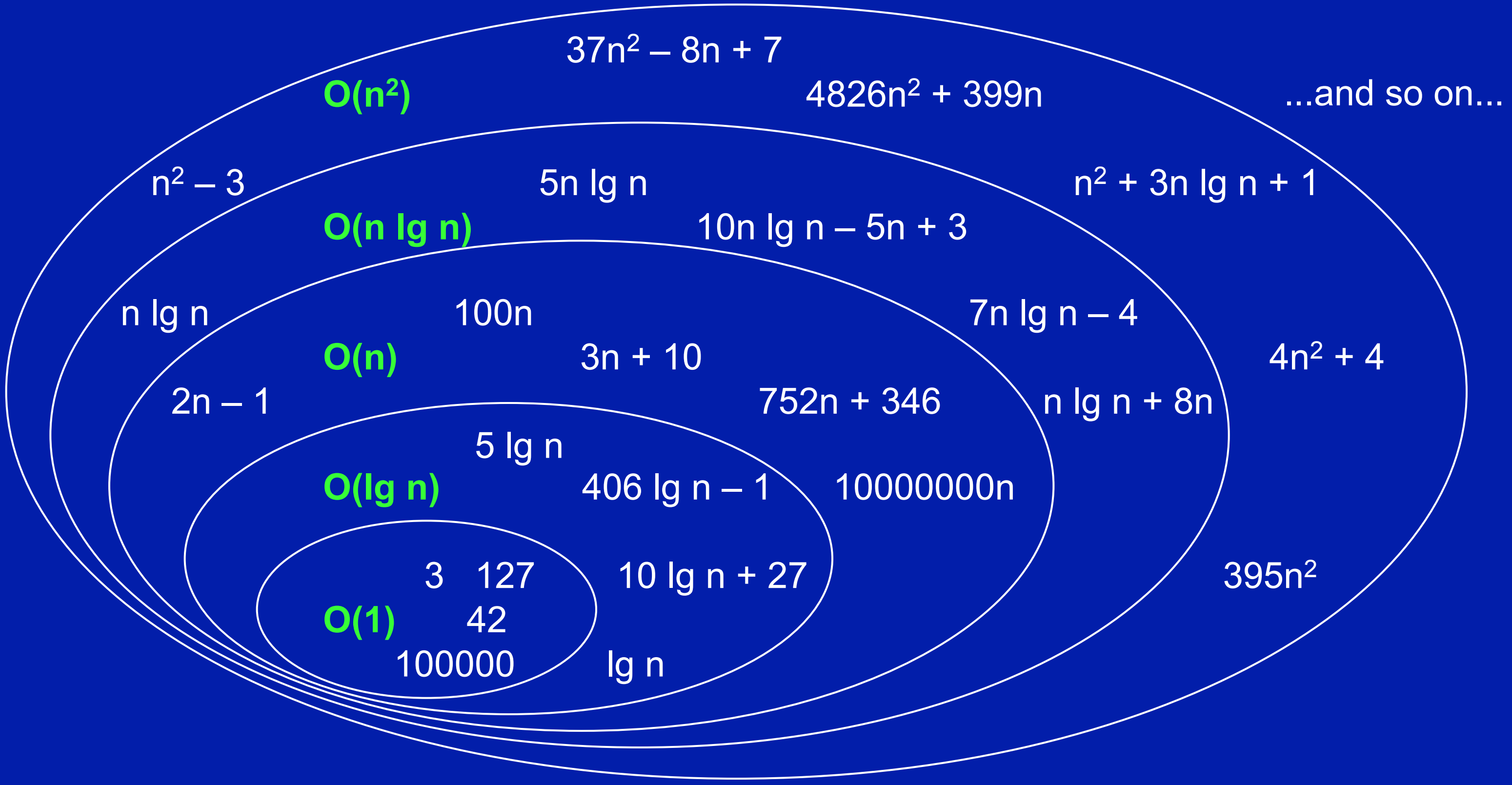
If you start with an algorithm that inherently has $O(n^2)$ complexity, there's no combination of constants that will give it $O(n \lg n)$ behaviour... ...as n gets really big.



O(g(n)) is the set of functions with smaller or same order of growth as g(n)

What exactly are we trying to do?

However, there may be a limited range of values of n for which the $O(n^2)$ algorithm gives better performance than the $O(n \lg n)$ algorithm.



O(g(n)) is the set of functions with smaller or same order of growth as g(n)

Common growth rates

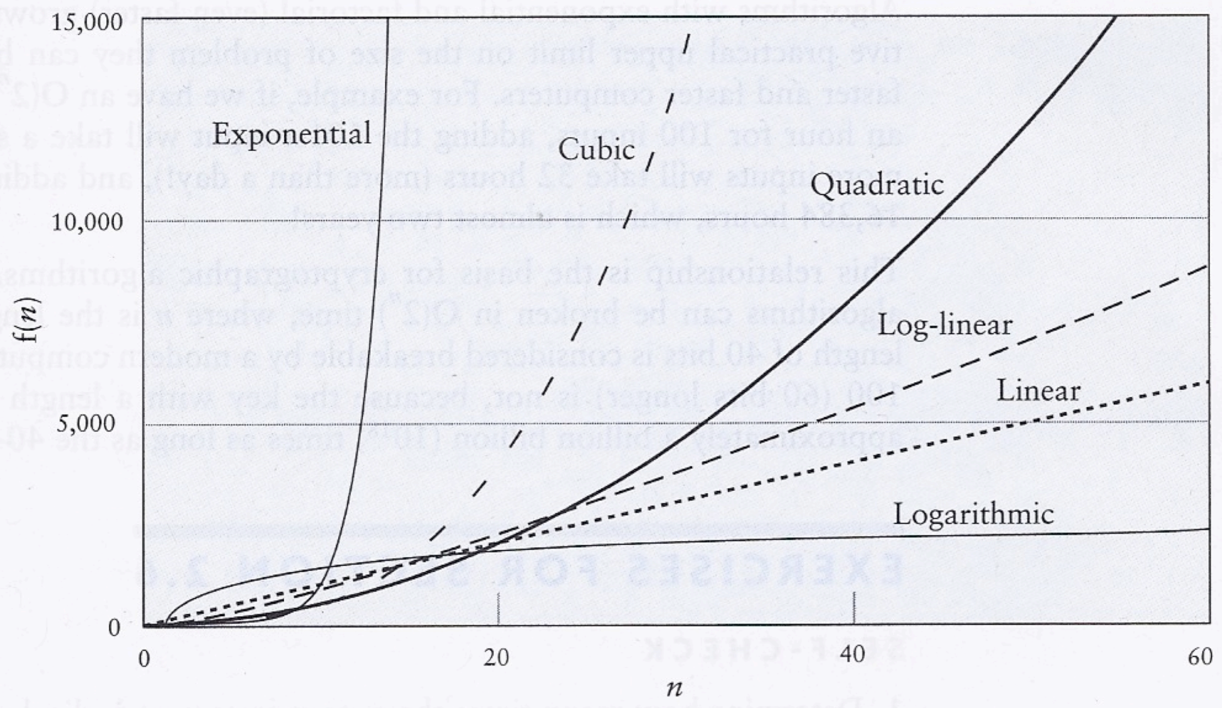
Big-O is for expressing how run time or memory requirements grow as a function of the problem size. Rate of growth gets bigger as you go down the table.

Big-O	*name*
O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n log n)	Log-linear
O(n^2)	Quadratic
O(n^3)	Cubic
O(n^k)	Polynomial – k is constant
O(2^n)	Exponential
O(n!)	Factorial

Common growth rates

FIGURE 2.8

Different Growth Rates



Low-order terms

Let's say we've calculated the run time for some bit of code as

$$T(n) = 42n^3 + 3n^2 - 17n - 8$$

We know we can disregard the constants:

$$T(n) = n^3 + n^2 - n$$

But we don't say that T(n) is $O(n^3 + n^2 - n)$. Why? Because as n gets very big, the growth of n^3 dominates the growth of the other terms – the low-order terms – and those terms have no noticeable impact as n gets very big.

Low-order terms

The “order of dominance” is in this same table. Read from the bottom up. $O(n!)$ dominates $O(2^n)$. $O(2^n)$ dominates $O(n^k)$. And so on...

Big-O	*name*
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(n^k)$	Polynomial – k is constant
$O(2^n)$	Exponential
$O(n!)$	Factorial

Big-Ω

What if you want to find a lower bound to the performance of an algorithm? That's where Big-Ω (Big-Omega) comes in.

Big-Ω says there's a function that is a lower bound to the worst/best/average/whatever case performance for the algorithm. In other words, the algorithm's worst/best/average/whatever case performance will never get better than that lower bound. Big-Ω is sort of like greater than or equal to when growth rates are being considered.

Big-Ω

The formal mathematical definition for Big-Ω:

T(n) is Ω(f(n)) if there are two constants, n_0 and c, both > 0, and a function f(n) such that cf(n) **<=** T(n) for all n > n_0

If T(n) is Ω(f(n)), then we say “T(n) is at least order of f(n)” or “T(n) is bound from below by f(n)” or “T(n) is Big-Omega of f(n)”.

For example, searching for an item in an unsorted array of n elements must be Ω(n) in time complexity because we must look at all n elements.

Big-Θ

If we know that T(n) is both O(f(n)) and Ω(f(n)), then we may say that T(n) is Θ(f(n)) (i.e., Big-Theta of f(n)).

The formal mathematical definition for Big-Θ:

T(n) is Θ(f(n)) if and only if T(n) is O(f(n)) and T(n) is Ω(f(n)).

This says that the growth rate of T(n) equals the growth rate of f(n). If we say that T(n) is Θ(n^2), we're saying that T(n) is bounded above and below by a quadratic function. It won't get worse than the upper bound, and it can't get better than the lower bound. Big-Θ assures the tightest possible bound on the algorithm's performance.

Big-Θ

Show that T(n) = 48n + 8 is Θ(n):

for O(n):

```
  48n + 8 <= cn
8(6n + 1) <= cn
   6n + 1 <= cn/8
       6n <= cn/8 – 1
        n <= cn/48 – 1/6
        n <= n – 1/6? (if c = 48) no
        n <= 2n – 1/6? (if c = 96) maybe
        1 <= 2 – 1/6? (if n0 = 1) yes

T(n) is O(n)
```

Big-Θ

Show that T(n) = 48n + 8 is Θ(n):

for Ω(n):

```
  48n + 8 >= cn
8(6n + 1) >= cn
   6n + 1 >= cn/8
       6n >= cn/8 – 1
        n >= cn/48 – 1/6
        n >= n – 1/6? (if c = 48) maybe
        1 >= 1 – 1/6? (if n₀ = 1) yes

T(n) is Ω(n) and T(n) is O(n) so T(n) is Θ(n).
```

Recursion

A recursive procedure consists of three parts:

1 The base case or termination condition. Usually the first thing done upon entering a recursive procedure

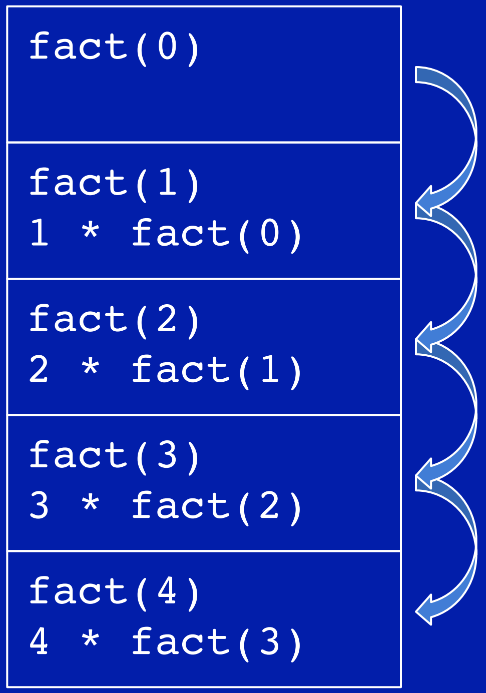
2 The reduction step -- the operation that moves the computation closer to the termination condition

3 The recursive procedure call itself

Visualizing recursion

How does it work? The activation stack model:

```
int factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return n * factorial(n – 1);
}


cout << fact(4) << endl;
```



Visualizing recursion

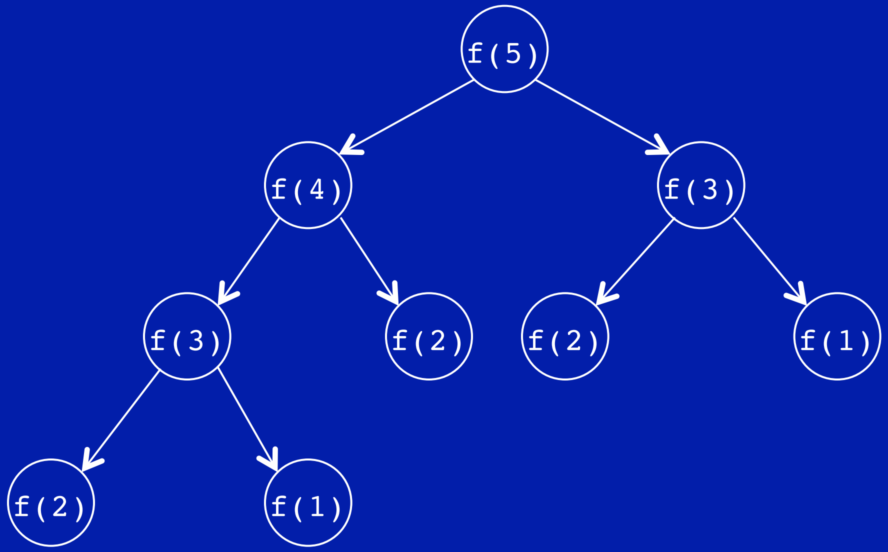
How does it work? The substitution model:

```
int factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return n * factorial(n – 1);
}


cout << fact(4) << endl;
  4 * fact(3)
  4 * 3 * fact(2)
  4 * 3 * 2 * fact(1)
  4 * 3 * 2 * 1 * fact(0)
  4 * 3 * 2 * 1 * 1
  4 * 3 * 2 * 1
  4 * 3 * 2
  4 * 6
  24
```

Visualizing recursion

How does it work? The recursion tree model:

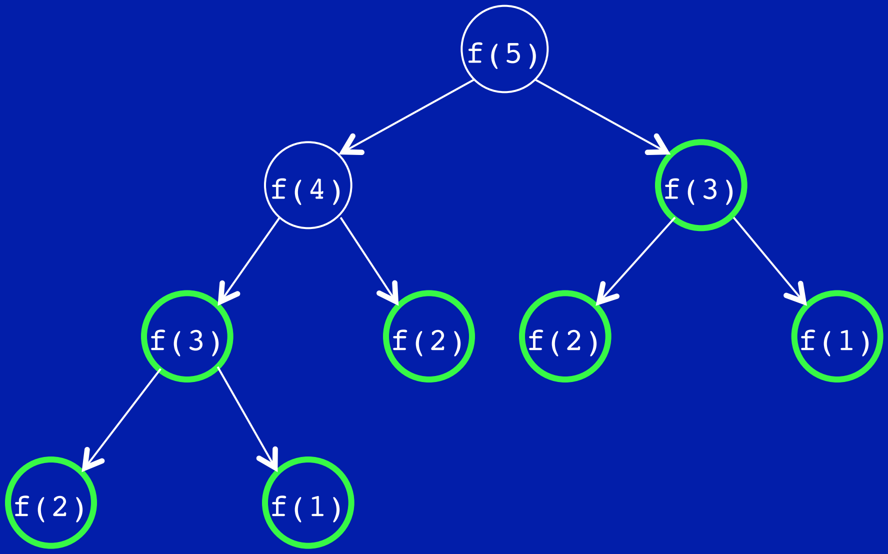
```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```



Visualizing recursion

How does it work? The recursion tree model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```



fib(3) is evaluated 2 times
fib(2) is evaluated 3 times
fib(1) is evaluated 2 times

How efficient is that?
Your book says that fib(100) requires about 2^{100} activation frames. If your computer can process 1,000,000 activation frames per second, it'll take 3×10^{16} years.

Visualizing recursion

How does it work? The activation stack model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
                    fib2      fib1
               fib3 fib3 fib3 fib3 fib3      fib2                fib2      fib1
          fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4 fib4      fib3 fib3 fib3 fib3 fib3
     fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5 fib5
main main main main main main main main main main main main main main main main main main main
```

Visualizing recursion

How does it work? The substitution model:

```
int fibonacci(int n)
{
  if (n <= 2)
    return 1;
  else
    return fibonacci(n - 1)
           + fibonacci(n - 2);
}

cout << fib(5) << endl;
```

```
fib(5)
fib(4) + fib(3)
fib(3) + fib(2) + fib(3)
fib(2) + fib(1) + fib(2) + fib(3)
1 + fib(1) + fib(2) + fib(3)
1 + 1 + fib(2) + fib(3)
2 + fib(2) + fib(3)
2 + 1 + fib(3)
3 + fib(3)
3 + fib(2) + fib(1)
3 + 1 + fib(1)
4 + fib(1)
4 + 1
5
```

Recursion mythology

A common complaint about recursion, as we've just seen, is the resource consumption in terms of memory (activation stack space) as well as time (handling the function calls and putting frames on/taking frames off the stack).

The culprit here is the repeated postponement of computations by pushing those computations on the stack.

Consider factorial. What’s the space complexity? How many frames go on the stack for factorial(10)? factorial(100)? factorial(1000)?

Recursion mythology

But what if there were a type of recursion that worked without postponing computations? If this were so, we could have factorial using O(1) stack space instead of O(n) stack space.

This type of recursion exists, and it's inspired by the fact that if we can pass a partially-completed computation through the recursive function calls, we don’t have to postpone any computations...we just do them as we go.

If we don’t postpone any computations, we don’t really need to push anything on the stack.

Tail recursion

This type of recursion is called tail recursion, and it often involves the introduction of an additional parameter used as a "variable" to hold the partially-computed result instead of storing postponed computations on the stack.

Tail recursion

```
int fact_aux(int n, int result)
{
  if (n <= 1)
    return result;
  else
    return fact_aux(n – 1, n * result);
}

int fact(n)
{
  return fact_aux(n, 1);
}
```

fact_aux is an auxiliary or helper function to keep the syntax of fact(n) the same as before. The recursive function is really fact_aux, not fact. Note that fact_aux has no postponed or pending computations on return from recursive calls. All the work is done “inside” the recursive call in the tail call position. No work is done “outside” the recursive call in that tail call position.

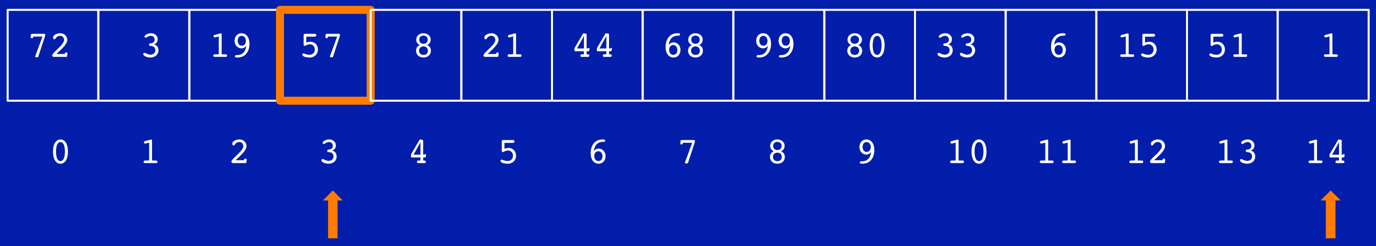
Tail recursion

```
int fact_aux(int n, int result)
{
  if (n <= 1)
    return result;
  else
    return fact_aux(n – 1, n * result);
}

int fact(n)
{
  return fact_aux(n, 1);
}
```

```
fact(4)
fact_aux(4, 1)
fact_aux(3, 4)
fact_aux(2, 12)
fact_aux(1, 24)
24
```

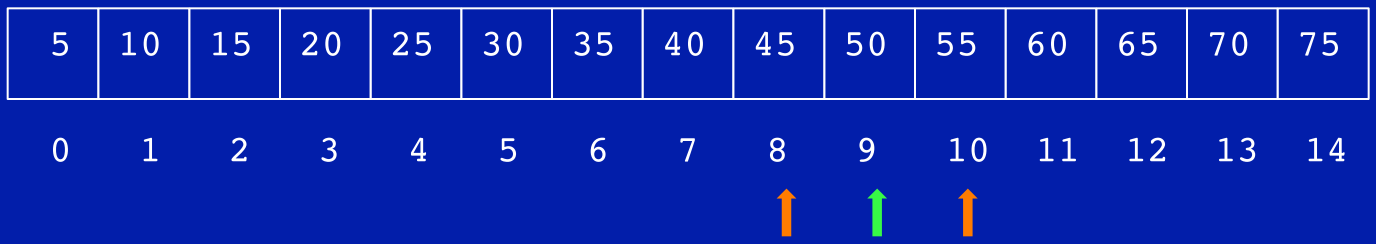
Using the substitution model, we don’t see any growth to the right, indicating there’s no postponed computations. But does that mean there’s no stack growth? We’re still making function calls and pushing stack frames, aren’t we?

Linear search with recursion



```
int linSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    if (array[left] == target)
        return left;
    else
        return linSearch(array, target, left + 1, right);
}

cout << linSearch(array, 57, 0, 14) << endl;

linSearch(array, 57, 0, 14)
  linSearch(array, 57, 1, 14)
    linSearch(array, 57, 2, 14)
      linSearch(array, 57, 3, 14) // yippee!!
```

Binary search with recursion



```
int binSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    int mid = (left + right) / 2;
    if (array[mid] == target)
        return mid;
    else if (target < array[mid])
        return binSearch(array, target, left, mid - 1);
    else
        return binSearch(array, target, mid + 1, right);
}

cout << binSearch(array, 55, 0, 14) << endl;

binSearch(array, 55, 0, 14)
  binSearch(array, 55, 8, 14)
    binSearch(array, 55, 8, 10)
```

Recursion vs. Iteration

Both involve the repetition of a sequence of statements.

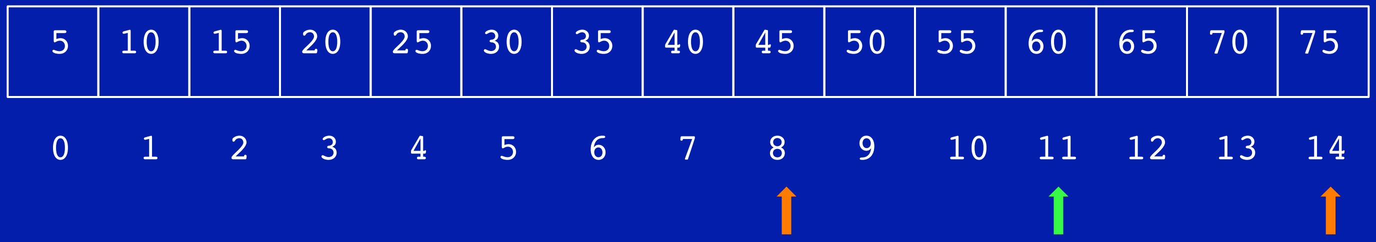
An iterative solution exists for any problem solvable by recursion.

An iterative solution may be more efficient.

A recursive solution is often easier to understand.

Here’s the iterative version. You decide...

Binary search with iteration



```
int binSearchIt(int array[], int target, int left, int right)
{
    int result = -1;
    while (! (right < left))
    {
        int mid = (left + right) / 2;
        if (array[mid] == target)
        {
            result = mid;
            right = left - 1;  // kill the loop
        }
        else if (target < array[mid])
            right = mid - 1;
        else
            left = mid + 1;
    }
    return result;
}
cout << binSearch(array, 55, 0, 14) << endl;
```

Recursion vs. Iteration

Although the iterative and recursive solutions to binary search in a sorted array achieve the same result with roughly the same number of steps, there is technically more overhead for function calls and returns than for simple loop repetition.

This difference is small, however. To repeat what your textbook says, generally if it is easier to conceptualize a problem as recursive, it should be coded as such. With problems like Fibonacci numbers being exceptions.

Induction for proving correctness

We can use induction to establish the truth of a given statement over an infinite range (usually natural numbers):

First, prove the base case (usually n = 0 or n = 1)

Then, prove that

> if any one statement in the sequence of statements is true (the inductive hypothesis)
>
> the very next one (n + 1) must be true (the inductive step)

Induction for proving correctness

So we have:

1. **Base Case (BC):** Prove the theorem for the simplest case
2. **Inductive Hypothesis (IH):** Assume the theorem holds for some arbitrary element, e
3. **Inductive Step (IS):** Show that the theorem holds for the successor of e
4. **Conclusion:** Together, 1 – 3 imply the theorem holds for all possible cases (the minimal case and all successors)

A valid induction proof clearly shows all these steps.

Induction for proving correctness

If the “statement” you want to prove correct is represented as a recursive function, the inductive proof is relatively easy...

Just follow the code’s lead and apply induction:

Your base case(s)? Your code’s base cases.

How do you break down the inductive step? However your code breaks the problem down into smaller cases.

What do you assume? That the recursive calls just work (for smaller input sizes as parameters, which better be how your recursive code works, no?).

Induction for proving correctness

```
// Precondition: n >= 0
// Postcondition: returns n!

int factorial(int n)
{
  if (n == 0)
    return 1;









  else
    return n * factorial(n – 1);
}
```

Always connect what the code does with what you want to prove.

```
Prove: factorial(n) = n!

Base case: n = 0
Our code returns 1 when
n = 0, and 0! = 1 by
definition. ✓

Inductive hypothesis: assume
that factorial(k - 1) =
(k - 1)!

Inductive step:  Show that
factorial(k) = k!

factorial(k - 1) = (k - 1)!
  (by IH)
k! = k * (k - 1)! (by defn)
For any k > 0, our code
returns k * factorial(k - 1)
  (look at the last line of
   the code) QED!!!
```

Induction for proving correctness

Can we use the same techniques for proving correctness of loops that we used for proving correctness of recursion?

Yes, we do this by stating and proving "invariants" – properties that are always true (i.e., they don't vary) at particular points in the program.

Induction for proving correctness

A loop invariant is:

A statement that is true at the beginning of a loop or at a given point) and remains true throughout the execution of the loop

It allows us to relate the state of the program (i.e. the data values) to the current iteration *i*

We can then use mathematical induction...

Induction for proving correctness

Proving a loop invariant:

Induction variable: number of times through the loop

Base case: prove the invariant true before the first loop guard (i.e., conditional) test

Induction hypothesis: Assume the invariant holds just before some (unspecified) iteration’s loop guard test

Inductive step: Prove the invariant holds at the end of that iteration (just before the next loop guard test)

Extra bit: Make sure the loop will eventually end

Induction for proving correctness

```
// Precondition: n >= 0
// Postcondition: returns n!

int factIter(int n)
{
    int f = 1;



    for (int i = 1; i <= n; i++)
        f = f * i;
    return f;
}
```

Loop invariant: at the end of the ith iteration, f = i!
i is the induction variable

Base case: 0! and 1! are both 1 by defn
f = 1 before first “i <= n”

Induction hypothesis: assume that just before “k <= n”, f = (k – 1)! (OR assume for 0, 1, ..., k – 1 that f = (k – 1)!)

Inductive step: during the i = kth iteration, the statement “f = f * i” executes, giving f_{new} = f_{old} * k

Induction for proving correctness

```
// Precondition: n >= 0
// Postcondition: returns n!

int factIter(int n)
{
    int f = 1;




    for (int i = 1; i <= n; i++)
        f = f * i;
    return f;
}
```

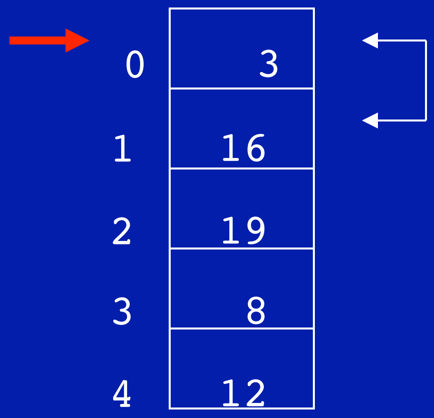
Induction hypothesis: assume that just before “k <= n”, f = (k - 1)! (OR assume for 0, 1, ..., k - 1 that f = (k - 1)!)

Inductive step: during the i = kth iteration, the statement “f = f * i” executes, giving f_{new} = f_{old} * k

By the induction hypothesis, f_{old} = (k - 1)!, so the value of f_{new} is k * (k - 1)!

By definition, k * (k - 1)! is k!, so f_{new} = k! QED again!

Selection sort



The smallest value so far is 3

Its index is 1

Let's say we want to sort the values in the array at the left in increasing order. One way to approach the problem would be to use an algorithm called selection sort. We start by setting a pointer to the first element in the array; this is where the smallest value in the array will be placed. Then we'll look at every value in this unsorted array and find the minimum value. Once we've found the minimum value, we swap that value with the one we selected at the beginning.

Estimating time required to sort

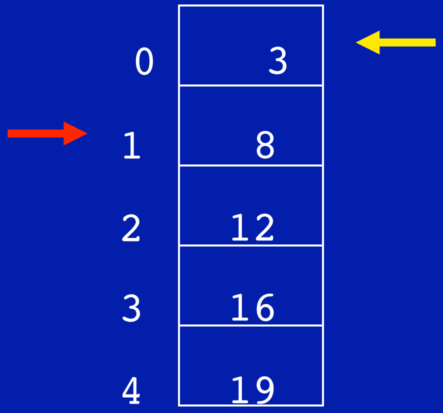
Let's assume that your computer could make 1 billion (1,000,000,000) comparisons per second. That's a lot of comparisons in a second. And let's say your computer was using selection sort to sort the names of the people in the following hypothetical telephone books. Here's some mathematical food for thought.

phone book	number of people (N)	N^2	number of seconds needed to sort	
Vancouver	544,320	296,284,262,400	296	or 5 minutes
Canada	30,000,000	900,000,000,000,000	900,000	or 10.4 days
People's Republic of China	1,000,000,000	1,000,000,000,000,000,000	1,000,000,000	or 31.7 years
World	7,000,000,000	49,000,000,000,000,000,000	49,000,000,000	or 1554 years

Estimating time required to sort

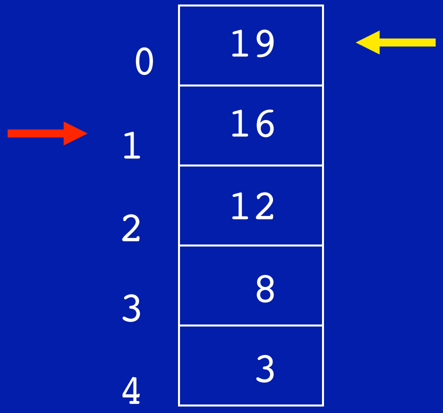
But note that the best sorting algorithms can run in N $\log_2$ N time instead of N^2 time. If N is 7,000,000,000, then $\log_2$ N is just a little less than 33. So if we round up to 33, N log N would be 231,000,000,000 comparisons. If our computer can perform 1,000,000,000 comparisons per second, then sorting all the names in the whole world phone book now takes 231 seconds instead of 1554 years. And that's why it's important to think about the efficiency of algorithms, especially as the size of your data set gets really really big.

Insertion sort



What input would let insertion sort show off its best possible performance? An array that's already sorted. How many comparisons would be made? How many elements would be moved?

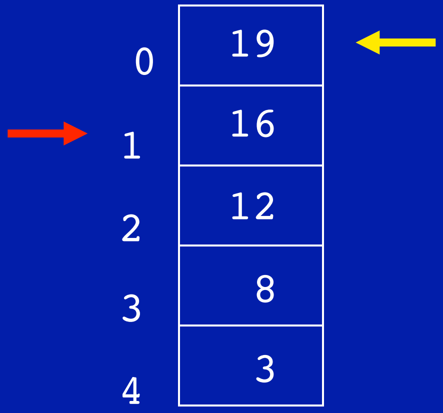
Insertion sort



What input would be the worst possible case for insertion sort? Sorted, but reversed. How many comparisons would be made? How many elements would be moved?

So what's the worst case Big-O for insertion sort? $O(n^2)$, or pretty much the same as for selection sort.

Insertion sort



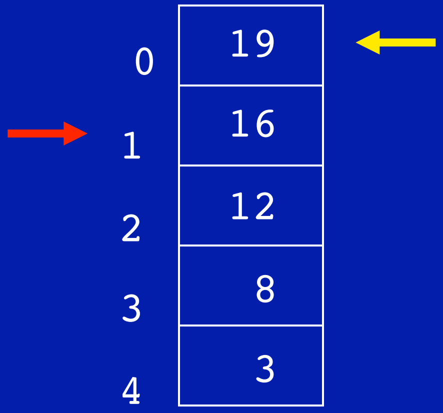
Pseudocode algorithm:

```
1.  for each array element from the second
    element to the last element
2.      Insert the selected element where it
        where it belongs in the array by
        shifting all values larger than the
        selected element back by one location
```

Your textbook has a more detailed pseudocode algorithm for insertion sort that behaves sort of like what we just did.

Here's some C++ code that might do what we we're talking about too:

Insertion sort



```
#include <iostream>
using namespace std;

void insertSort(int array[], int n)
{
    int i, j;
    int key;

    for(i = 1; i < n; i++)
    {
        key = array[i];
        j = i - 1;
        while(j >= 0 && array[j] > key)
        {
            array[j + 1] = array[j];
            j--;
        }
    array[j + 1] = key;
    }
}
```

Mergesort

Mergesort takes a different approach to the problem. It falls in the class of algorithms called “divide and conquer”.

In mergesort, the problem space is continually split in half by applying the algorithm recursively to each half, until the base case is reached.

A simple algorithm for mergesort is:

```
mergesort(unsorted_list)
    Divide the unsorted_list into two sublists of half the size of the unsorted_list
    Apply mergesort to each of the unsorted sublists
    Merge those two now-sorted sublists back into one sorted list
```

where ‘list’ could be any sequential data structure

Mergesort

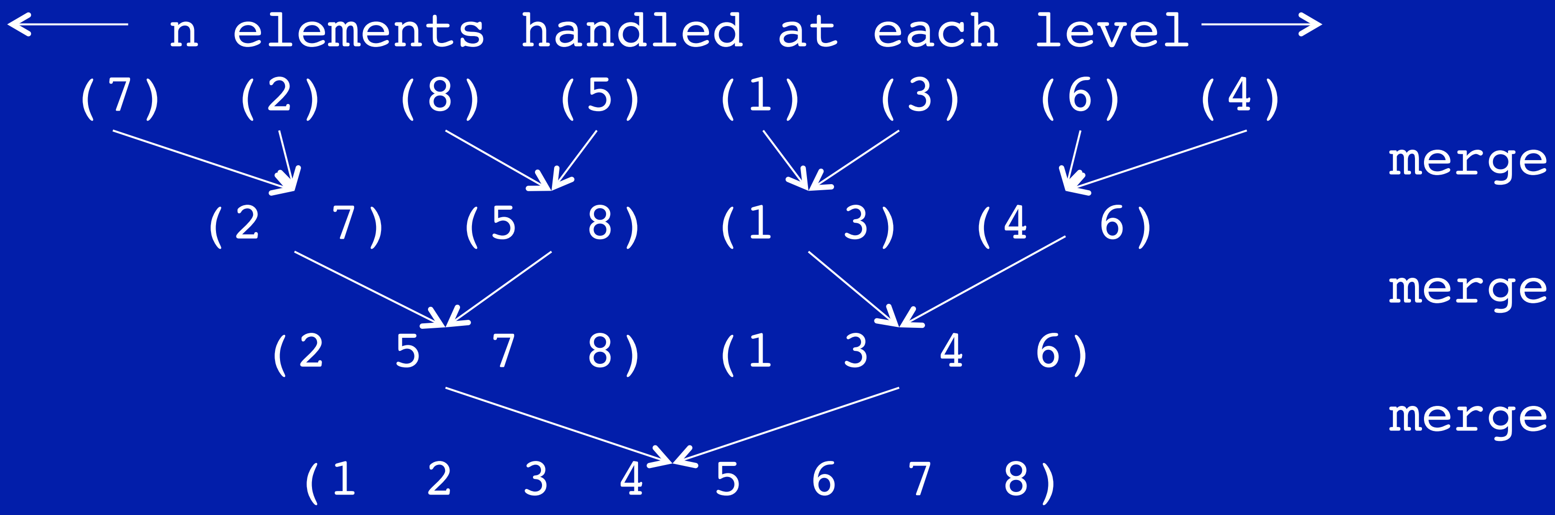
What are we imperfectly tracing? Code like this:

```
void merge(int array[], int lo, int mid, int hi, int tmp[])
{
    int a = lo, b = mid + 1;
    for(int k = lo; k <= hi; k++)
    {
        if(a <= mid && (b > hi || array[a] < array[b]))
            tmp[k] = array[a++];
        else
            tmp[k] = array[b++];
    }
    for(int k = lo; k <= hi; k++)
        array[k] = tmp[k];
}
```

Mergesort

What kind of time complexity are we dealing with here?

If were working with arrays, not lists, those last three lines represent the movement of n elements from the original array to the temporary array and back again, along with the required comparisons to maintain sorted order while merging. So each round of merging is O(n).

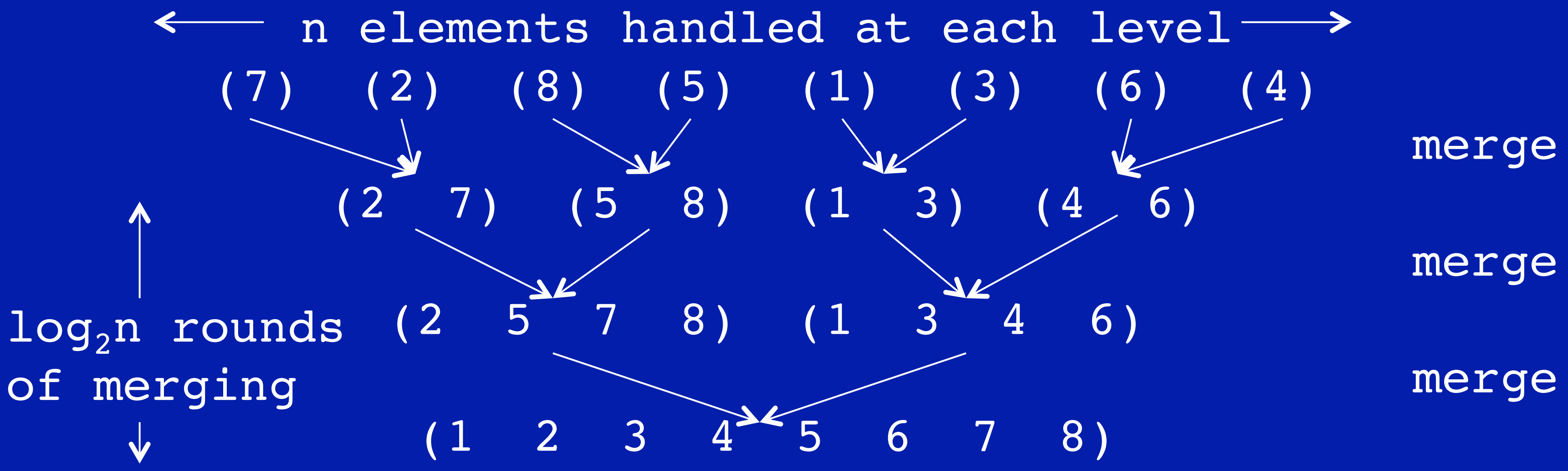


Mergesort

What kind of time complexity are we dealing with here?

There are 3 rounds of merging when n = 8. How many rounds of merging would be required if n = 16? n = 32? What does that tell you?

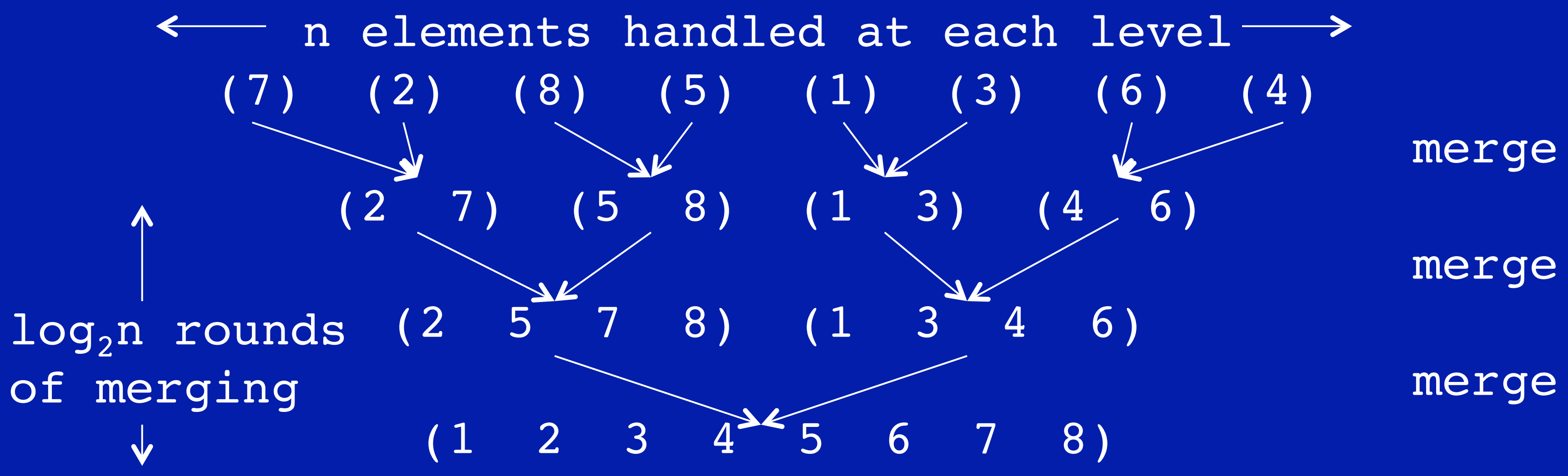
How does O(lg n) sound?



Mergesort

What kind of time complexity are we dealing with here?

Worst case time complexity for mergesort is O(n lg n).
What about space complexity for the array-based approach?



Quicksort

If we want to sort big sequences quickly, without the extra memory and copying back and forth of mergesort, the answer is quicksort. In practice, it is the fastest sorting algorithm known. While its worst-case run time is O(n^2), its average run time is O(n lg n).

A simple algorithm for mergesort is:

```
quicksort(unsorted_list)
    Choose one element to be the *pivot*
    Partition (i.e. reorder) the list so that all elements < the pivot are to the left
      of the pivot, and all elements > the pivot are to the right of the pivot
    Apply quicksort recursively to the sublist to the left and the sublist to the right
```

where ‘list’ could be any sequential data structure

Quicksort

Here’s a C++ implementation of quicksort (from Jon Bentley). Its behaviour may not match up exactly with what you just saw.

```
void qsort(int x[], int lo, int hi)
{
    int i, p;
    if (lo >= hi) return;
    p = lo;
    for( i=lo+1; i <= hi; i++ )
        if( x[i] < x[lo] )
            swap(x[++p], x[i]);
    swap(x[lo], x[p]);
    qsort(x, lo, p-1);
    qsort(x, p+1, hi);
}

void quicksort(int x[], int n)
{
    qsort(x, 0, n-1);
}
```

Quicksort complexity

Quicksort is comparison-based, so the operations are comparisons.

In the partitioning task, each element was compared to the pivot. So there are roughly n comparisons in each partitioning (really n – 1?).

In the first step, we have n comparisons.
In the second step, we have floor(n/2) * 2 comparisons (for each half)
In the third step, we have floor(n/4) * 4 comparisons (for each quarter)

... or O(n lg n)

Quicksort complexity

What input will give quicksort its worst performance?
An array that's already sorted.

$1\ \boxed{2\ 3\ 4\ 5\ 6\ 7\ 8}$ $n - 1$ comparisons

$1\ 2\ \boxed{3\ 4\ 5\ 6\ 7\ 8}$ $n - 2$ comparisons

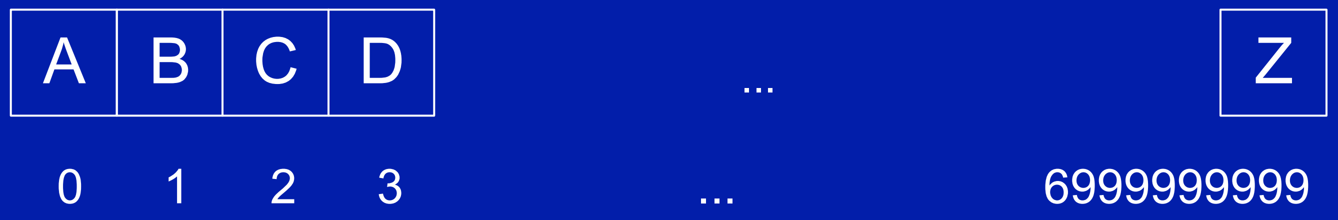
$1\ 2\ 3\ \boxed{4\ 5\ 6\ 7\ 8}$ $n - 3$ comparisons and so on...

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

or $O(n^2)$

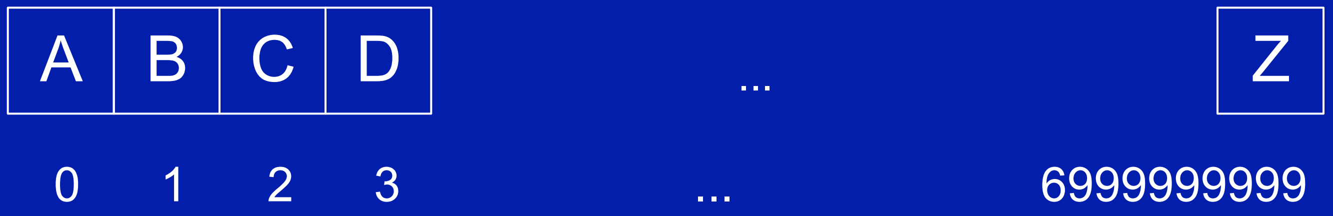
Performance depends on choosing good pivots. With good pivot choices, average case behaviour for quicksort is $O(n \lg n)$

Issue #1



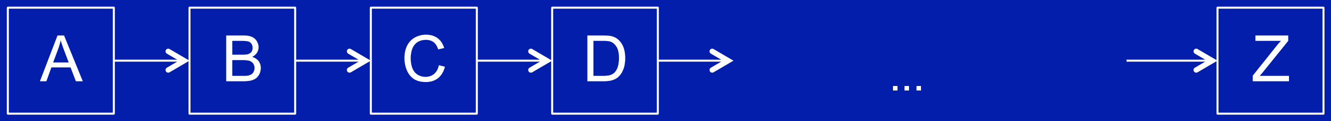
Where are we gonna put that 7 billion element array in memory? Finding that much contiguous available memory might be problematic. A data structure that was more flexible, more dynamic, and much less dependent on contiguous memory could be helpful. What kind of structure might that be? Some sort of linked list structure seems like a possibility.

Issue #2



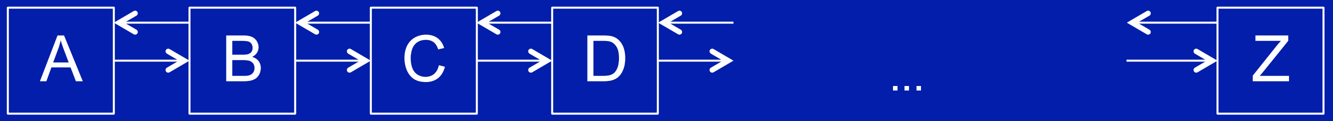
Now that we’ve sorted our 7 billion element array in O(n lg n) time, what do we do if we want to add one more element in the right place?

There’s got to be a better way



Maybe we could speed up the insertion time even more.
Before we hinted at a linked list structure for flexibility.
Can we do binary search on a singly-linked list to get that O(lg n) time complexity?

There’s got to be a better way



A doubly-linked list lets us move in both directions. So binary search should be possible, but we can’t move the pointers (left, right, mid) with simple index arithmetic. We would have to do link traversals, and as long as we’re traversing the links in the list, we might as well look at the values at the nodes as we pass by, and this just turns into linear search. What is the linked structure that makes it all come together?

Trees

For example, we like our trees upside down. Non-computing people tend to think this is weird. Really, we're just trendsetters...

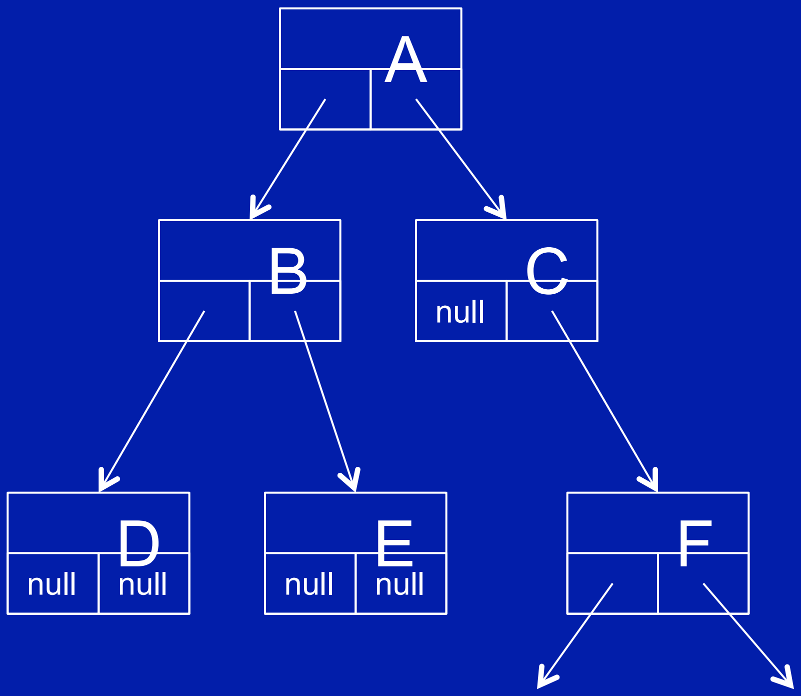
Binary trees

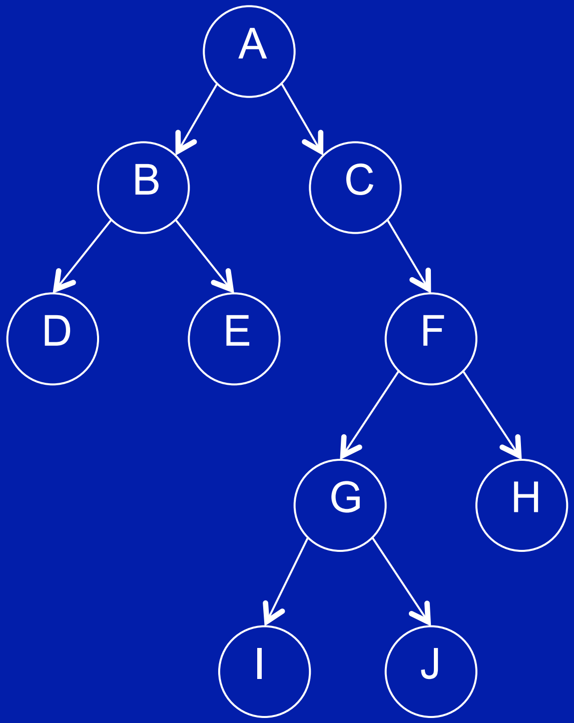
For reasons which will become obvious, we’re primarily interested in binary trees.

A binary tree is either

- empty (null for us), or
- a node called the root node and two binary trees called the left subtree and a right subtree

Binary trees



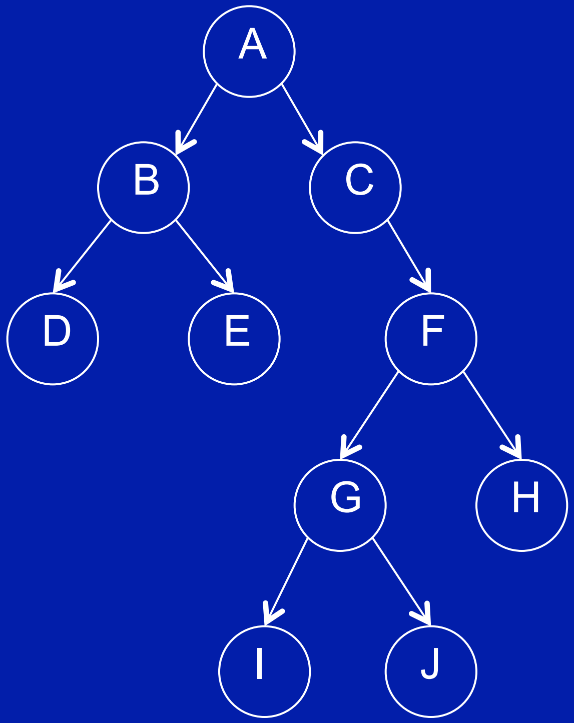


Same tree as in the previous slide but the pointer from C to F is now clearly a right pointer.

Each binary tree node holds some data, a pointer to its left subtree and a pointer to its right subtree.

Preorder traversal

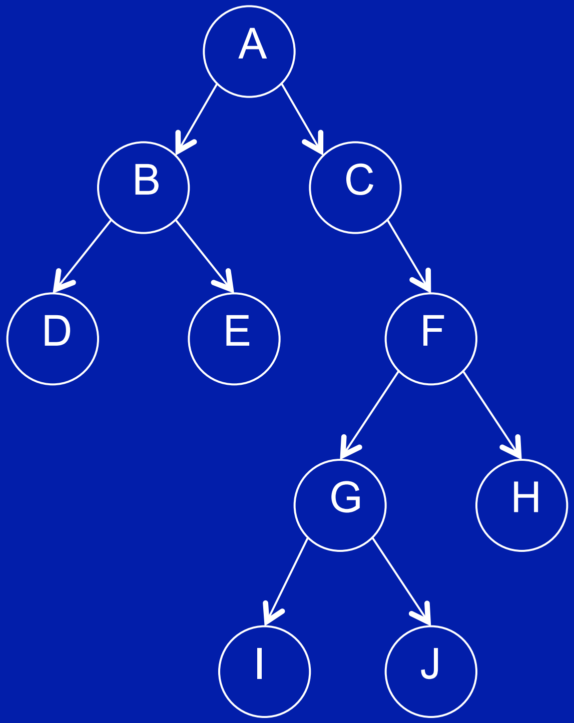
```
if the tree is empty
  return;
else
  visit (process) the root;
  apply preorder to the
    left subtree;
  apply preorder to the
    right subtree;
```



Let’s say that visit or process in this case means “print the value”. In what order will the values be printed?
A B D E C F G I J H

Inorder traversal

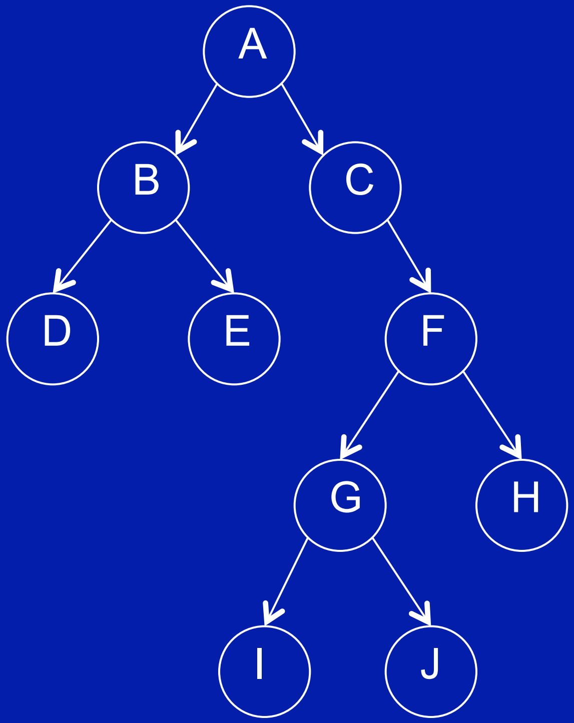
```
if the tree is empty
  return;
else
  apply inorder to the
    left subtree;
  visit (process) the root;
  apply inorder to the
    right subtree;
```



Let’s say that visit or process in this case means “print the value”. In what order will the values be printed?
D B E A C I G J F H

Postorder traversal

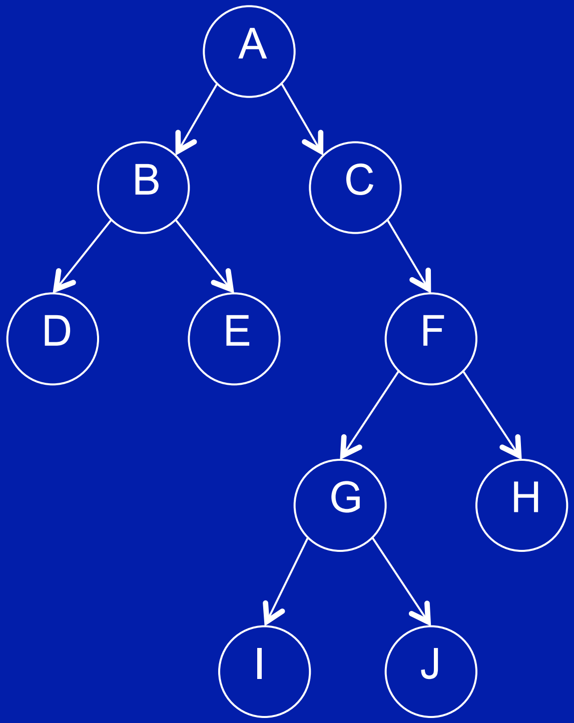
```
if the tree is empty
  return;
else
  apply postorder to the
    left subtree;
  apply postorder to the
    right subtree;
  visit (process) the root;
```


$$\begin{tikzcd}[column sep=small]
 & & A \arrow[dl] \arrow[dr] & & & \\
 & B \arrow[dl] \arrow[dr] & & C \arrow[dr] & & \\
D & & E & & F \arrow[dl] \arrow[dr] & \\
 & & & G \arrow[dl] \arrow[dr] & & H \\
 & & I & & J &
\end{tikzcd}$$

Let's say that visit or process in this case means "print the value". In what order will the values be printed?
D E B I J G H F C A

Level order traversal

```
add root to queue
while queue not emtpy:
  take node from queue
  process the node
  if node->left isn’t null
    then put node->left on
    queue
  if node->right isn’t null
    then put node->right on
    queue
```



How could I get the traversal (printing) to happen in this order?: A B C D E F G H I J

This is called level order traversal.

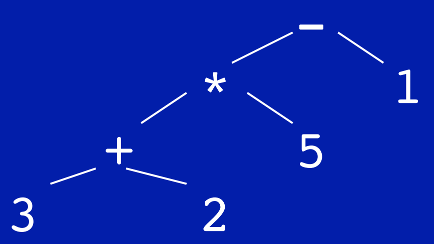
Binary expression trees

Here’s one way in which tree traversal can be useful. Arithmetic expressions can be represented as binary trees. Consider the expression (3 + 2) * 5 – 1

Going left to right through the expression, we can start to build the expression tree.

If we print this tree using postorder traversal, we get

```
3 2 + 5 * 1 -
```



Binary search trees

Binary trees are cool, but what we are really interested in is a class of binary trees called binary search trees.

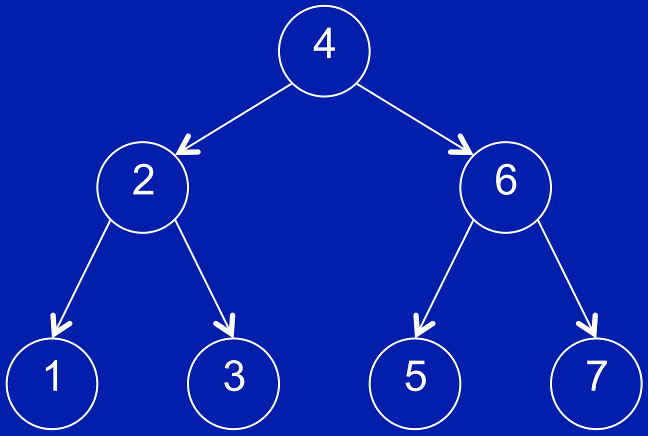
A binary search tree is a binary tree in which every node is

- empty or
- the root of a binary tree in which all the values in the left subtree are less than the value at the root, and all the values in the right subtree are greater than the value at the root.

Binary search trees

Finding a target value in a binary search tree (BST) is like applying binary search to a sorted array.

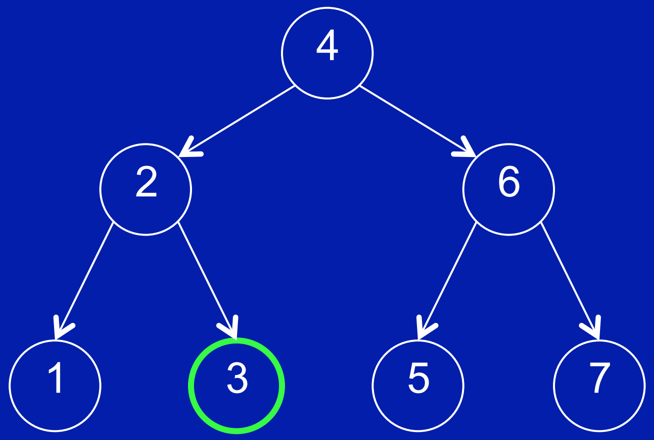
```
if the tree is empty
  (i.e. the root is null)
  then the value is not
  found so return failure;
else if
  the target value = the
  value at the root node
  then return success;
else if
  the target value < the
  value at the root node
  then return the result
  of searching the left
  subtree;
else
  return the result of
  search the right subtree;
```



Binary search trees

Finding a target value in a binary search tree (BST) is like applying binary search to a sorted array.

```
if the tree is empty
  (i.e. the root is null)
  then the value is not
  found so return failure;
else if
  the target value = the
  value at the root node
  then return success;
else if
  the target value < the
  value at the root node
  then return the result
  of searching the left
  subtree;
else
  return the result of
  search the right subtree;
```

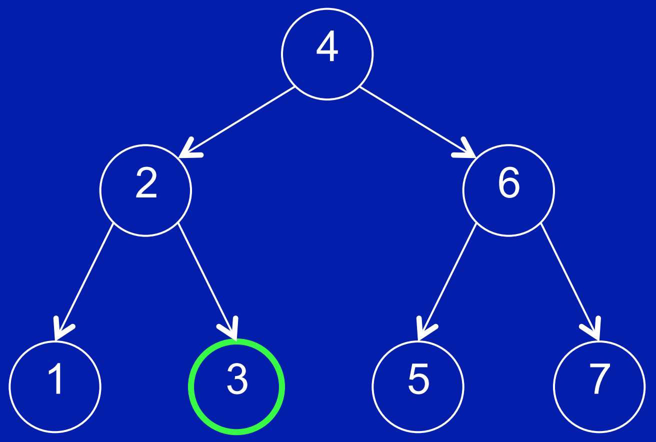


Search for 3.
Is it here? No.
Is it here? No.
Is it here? Yes.
What's your guess as to time complexity of finding a target in this BST?

Binary search trees

Finding a target value in a binary search tree (BST) is like applying binary search to a sorted array.

```
if the tree is empty
  (i.e. the root is null)
  then the value is not
  found so return failure;
else if
  the target value = the
  value at the root node
  then return success;
else if
  the target value < the
  value at the root node
  then return the result
  of searching the left
  subtree;
else
  return the result of
  search the right subtree;
```

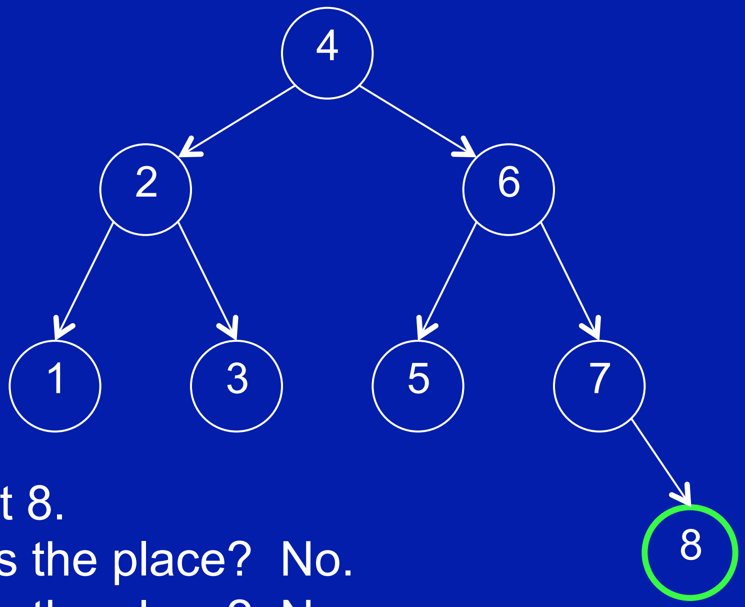


O(lg n) is a pretty good guess. Why?

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success;
else if
  the target value = the
  value at the root then
  the target is already
  in the BST, return failure;
else if
  the target < the value at
  the root then call insert on
  the left subtree;
else
  call insert on the right
  subtree;
```

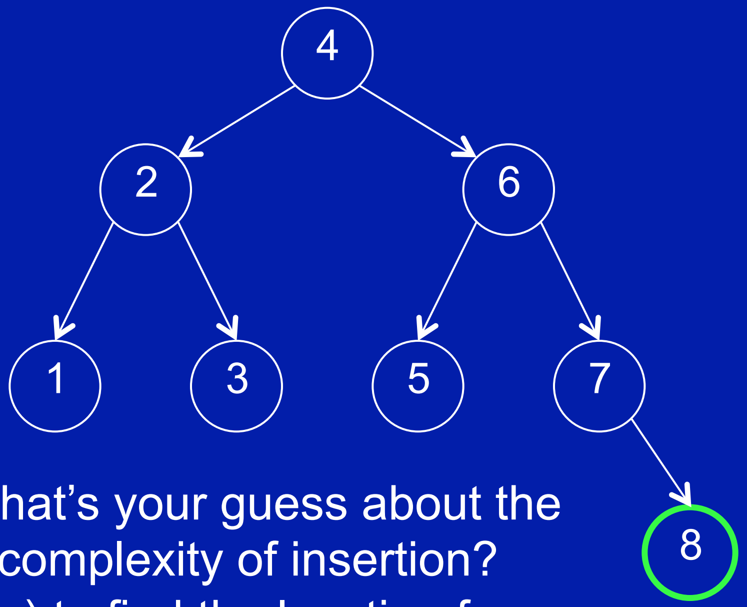


Insert 8.
Is this the place? No.
Is this the place? No.
Is this the place? No.
But the right subtree of 7 is empty, so we create a new node which is now the root of right subtree of 7.

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success;
else if
  the target value = the
  value at the root then
  the target is already
  in the BST, return failure;
else if
  the target < the value at
  the root then call insert on
  the left subtree;
else
  call insert on the right
  subtree;
```

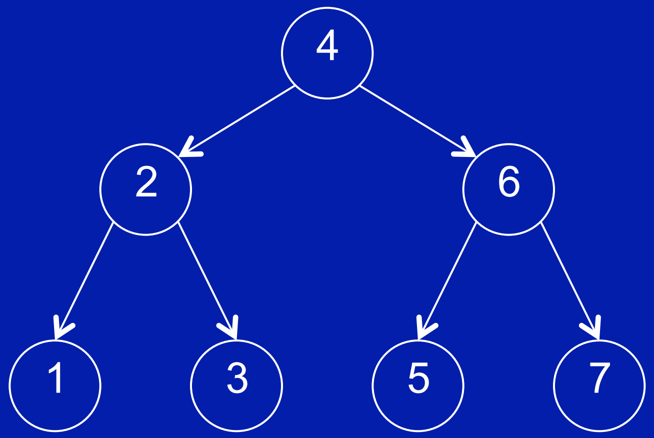


So what's your guess about the time complexity of insertion? O(lg n) to find the location for insertion plus some fixed time to create the new node and add the link.

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success;
else if
  the target value = the
  value at the root then
  the target is already
  in the BST, return failure;
else if
  the target < the value at
  the root then call insert on
  the left subtree;
else
  call insert on the right
  subtree;
```



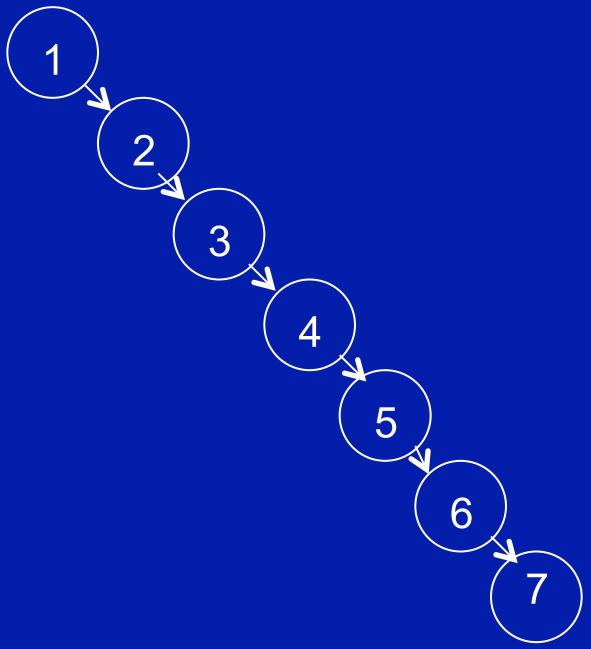
Building a binary search tree from an unsorted sequence of values is just the repeated application of insert.

What do you get if you insert 4 6 2 5 3 1 7 in that order?

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success;
else if
  the target value = the
  value at the root then
  the target is already
  in the BST, return failure;
else if
  the target < the value at
  the root then call insert on
  the left subtree;
else
  call insert on the right
  subtree;
```



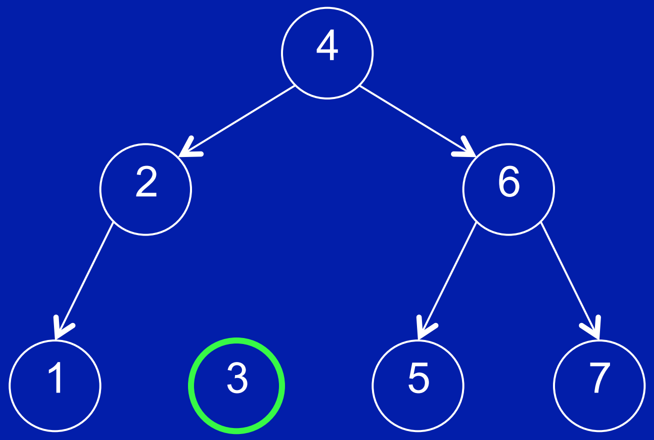
Building a binary search tree from an unsorted sequence of values is just the repeated application of insert.

What do you get if you insert
1 2 3 4 5 6 7 in that order?

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  is a leaf node then its
  parent's pointer to that
  leaf node is set to null;
```

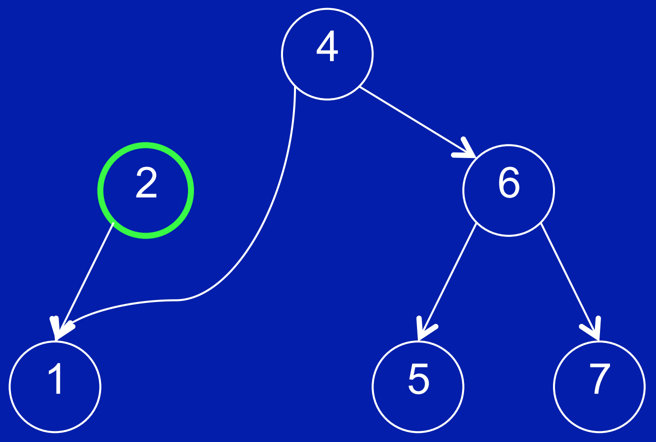


Delete 3.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  has only a left or a right
  child, then replace the
  target with the child;
```

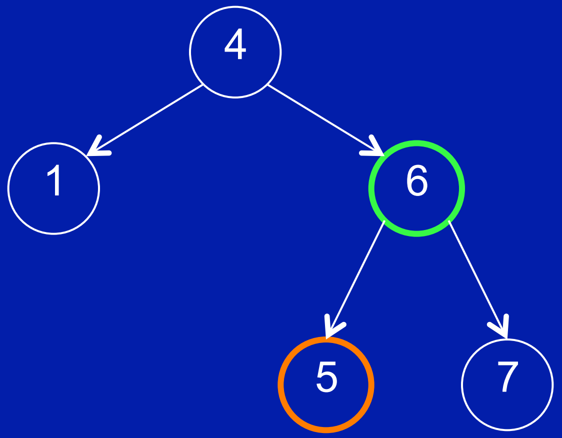


Delete 2.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one;
```

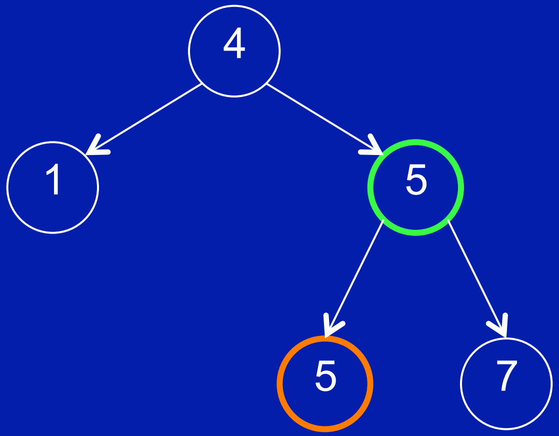


Delete 6.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one;
```



Delete 6.

What kind of programming on exam?

Not a lot.
Not overly complicated.
No OO programming.

What kind of programming on exam?

You should be able to read and write code on the order of, but not necessarily limited to:

push
pop
enqueue
dequeue
linear search
binary search
sorting (both iterative and recursive)

What else on exam?

Expect to

- write pseudocode
- trace algorithms
- figure out T(n) for (pseudo)code
- show Big-O, Big-Omega, Big-Theta
- perform simple induction proof on series
- draw pictures
- write short answers in English
- answer multiple choice questions
- do some simple math
- whatever else I can think of...