Computer Science 32

Lecture #1

~~COVID~~ EDITION

Professor: Carey Nachenberg
E-mail: climberkip@gmail.com

Class details: Franz 1178 - M/W 10am-12pm
Office hours: Eng VI, 372 (and via Zoom initially)
Mondays 12pm-1pm
Wednesdays 9am-10am

My Office: Eng VI, 297

The Question on Everyone's Mind!

What is the airspeed velocity of an unladen swallow?

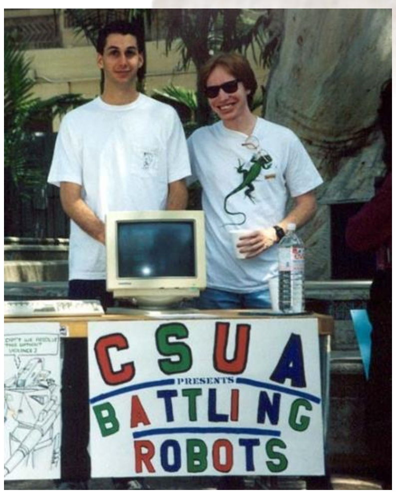
When will we be back in person?

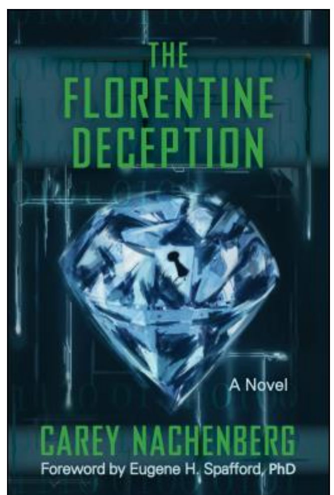
There's been no change - we're still supposed to be back in-person after two weeks.

Don't worry – I'll tell you the moment I hear any news!

Who Am I?

299934





Carey Nachenberg

Age: 50

School: BS+MS in CS, UCLA '95

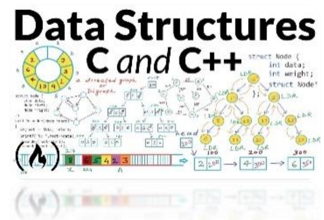
Work: Adjunct Prof at UCLA

Hobbies: Rock climbing, weight training, teaching CS!

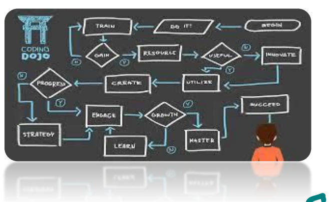
My goal: To make an impact on your life (through getting you excited about programming)!

What You'll Learn in CS32

Advanced C++ Topics
Object Oriented Programming and C++ language features



Data Structures
The most useful data structures (e.g., lists, trees, hash tables)



Algorithms
The most useful algorithms (e.g., sorting, searching)

Building Big Programs
How to write large programs (1000+ lines)

Basically, once you complete CS32, you'll know 95% of what you need to succeed in industry!

Official Class Websites

http://www.cs.ucla.edu/classes/winter22/cs32/

The class website has the syllabus, grading policy, academic integrity agreement and assignments/deadlines.

[Bruin Learn Website]()

While we're virtual, I will post recordings of all lectures on Bruin Learn.

Warning: I will not always announce homework/projects so you have to track this on your own and be on top of the trash*!

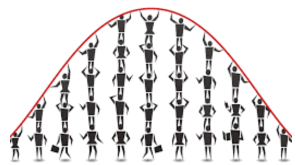
* Being on top of the trash means being responsible**.

** If you're not responsible, you could get *rekt*.

The Syllabus...

I know you won't read it... but...

Academic integrity agreement



Curving and grading policy

Late submission policy

TA Office hours

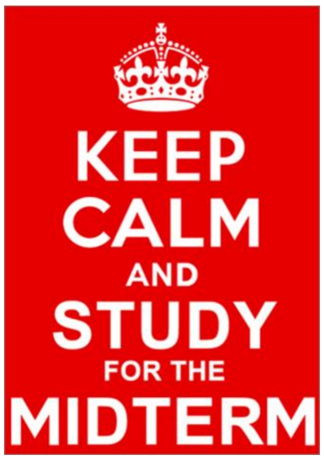
Visual Studio G++

Testing your code w/ 2 different compilers

Deadlines for all hw/projects

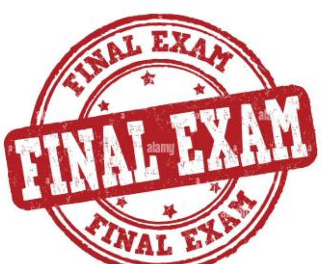
Important Dates

Project #1: Due Wed, Jan 12th (next Wed!)



Midterm #1: Tues, Feb 1st
6pm-7:30pm (not during class hours!!)

Midterm #2: Tues, Feb 22nd
6pm-7:30pm (not during class hours!!)



Final Exam: Sat, March 12th
11:30am-2:30pm
(This is the Saturday BEFORE Finals Week. Don't forget!)

Project #1: Due NEXT WED!

In P1, you have to add a few features to a C++ program that we provide.

It's worth exactly 

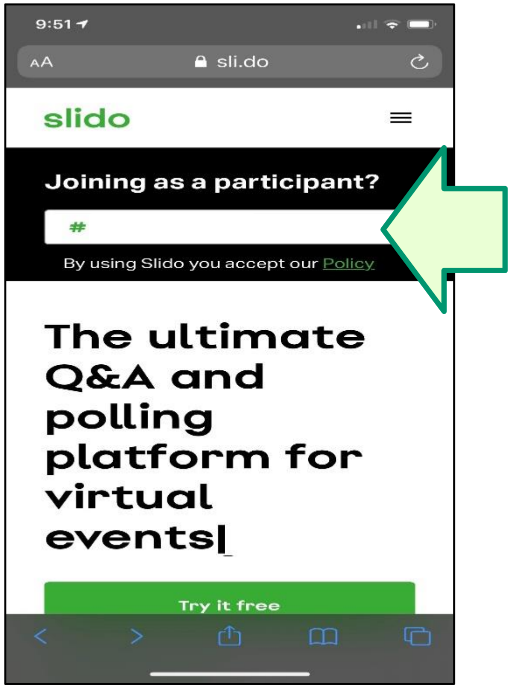
% of your total grade!

It's mostly for you to evaluate how ready you are for CS32.

If you find it difficult, I strongly encourage you to do a bit more prep and take the class in spring. ☺

Making Virtual Class Tolerable

(Until we're back in person!)



We'll be using Slido for questions!

On your laptop/phone, go to:
https://sli.do
Use event code: UCLA-CS32

You can ask and upvote questions, all anonymously.

I'll also be doing surveys to make sure everyone is following along.

To make things more interesting, we'll have 5 min "detox" activities from time to time...

Participate for Prizes!

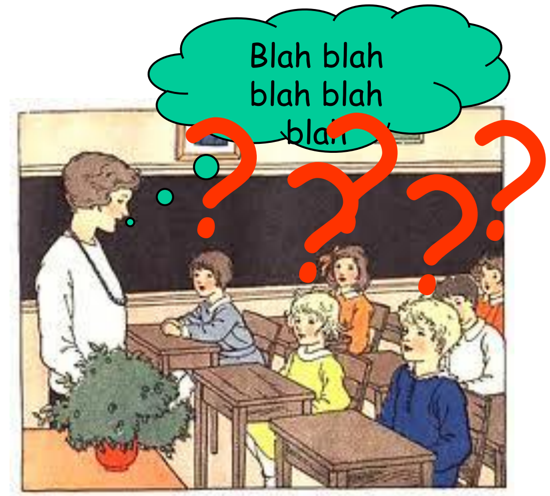
(Who says bribery doesn't work)

I'll be giving one $20 Amazon Gift Card out after each virtual lecture to reward participation.

If you answer one of my questions or ask a relevant question, you'll be entered in a raffle each day of class!

Carey's Thoughts on Teaching

It's more important that everyone understand a topic than I finish a lecture on time.



Don't be shy!!!
If something confuses you…
it probably confuses 5 other people too.
So always ask questions if you're confused!

Always save more advanced questions for office hours or break.

I reserve the right to wait until office hours to answer advanced questions.

Questions? Bring Em On!

Alright, bring on those questions!

Obfuscated C Programming Contest:

What Does it Do?

Answer: A maze solver! Given a file, it will find a path from the @ to the exclamation point!

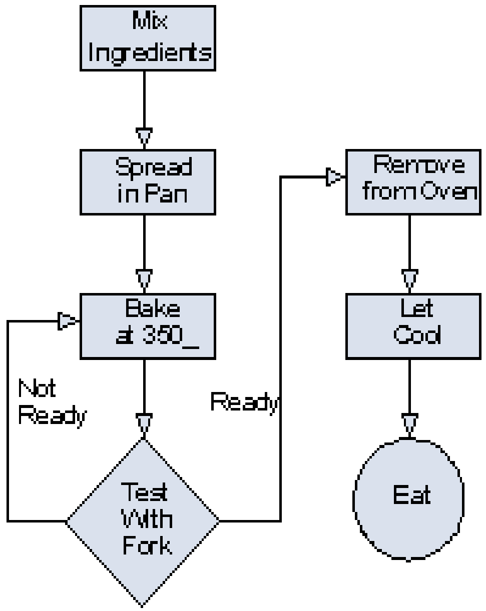
```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//@                                                                                  //
char _int[]={37,99,0,10,0,71,111,  111,100,98,121,101,32,58,45,41,10,0,27,91, 72,
27,  91,74,0,27,91,37,100,59,37,   100,72,0,27,91,37,100,109,0,0,0,-30,-108,  -125,0,-
30,  -108,-127,0,-30,-108,-109,0, -30,-108,-113,0,-30,-108,-101,0,-30,-108,  -105,0,
33,  0x20,0,0,0,0,0,0,0,0,0,0,0,  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0   ,0,0,0,0
,0,0      ,0,0,0};main(){void*   y3=stdout,*s0=stdin,*(*mv)(size_t)=&      malloc;int qq_,*hh,
qf,(*g4)  (void*)=&fflush,(*   z1)(const char*,...)=&printf,(*p0)(int)=   &putchar,c=0,
i,dd,sz,h ,s,e,pr,*trundle,* vp5,*nqd,oo[10];char*p5;goto cow;pic:g4(y3      );    sz=dd*h;
hh=mv(     sizeof(int)*sz);  z1(_int+18);goto china;tokyo:;if(qq_==     qf)   goto ten ;c=hh [
qq_++];  oo[0]=dd;oo[1]=h;oo [2]=c%dd;oo[3]=c/dd;oo[4]=0;{int x=    oo[2],         y=oo[3],dd=*
oo,h=   oo[1],i=x+y*dd;if(x+  1<dd&&(p5[i+1]==32||p5[i+1]==33))  oo[5+oo[4]++  ]=i+1;if(
x-1>  -1&&(p5[i-1]==_int[66]  ||p5[i-1]==_int[67]))chew: oo      [5+oo[4]++]=i    -1;if(y+1<h
&&p5   [i+dd]==_int[67]||p5[i +dd]==_int[66])oo[5+oo[4]++]=  i+    dd;if(y-1>-1&&   p5[i-dd]
==_int   [67]||p5[i-dd]==_int       [66])chun:oo[5+oo[4]    ++]=i  -dd;}if(c&1?nqd  [c]:vp5[c] )
goto   tokyo;wow:;if(oo[4]--  <=0)   goto chin;i=oo[5+oo  [4]];   if((i&1?vp5[i]:  nqd[i])==0
){   trundle[i]=c;usleep(    25000)  ;g4(y3);z1(_int+34   ,_int[  67]);z1(_int+25,  1+i/dd,1+i%
dd); p0(88);if(i==e)goto z; i&1?(vp5    [i]=1):(nqd[i]   =1);hh[  qf++]=i;}goto     wow;chin:c
&1?( nqd[c]=1):(vp5[c]=1);  goto tokyo; z:;z1(_int+34, 36);z1(   _int+25,1+e/   dd, 1+e%dd)
;p0(  33);tww:;if(trundle   [i]==s)goto  ten;i=trundle  [i];    usleep(50000) ;z1(     _int+34,35
);z1(  _int+25,1+i/dd,    1+i%dd);p0(47)  ;g4(y3);z1(  _int+34 ,33);z1(_int   +25,1 +    s/dd,1+s%
dd);p0    (95);goto    tww;ten:;z1(_int  +25,h+3,1);  g4(y3);  goto pio;    cow:;   {int c=0;s=e=-1
;char**      strs=mv  (1000),d;h=0;for(;  h<1000;){   char*s=  mv(512);if  (fgets  (s,   512, s0)  ||
feof(s0)) {    dd=   strlen(s)>dd?strlen(  s):dd;    strs[h]=  s;h+=1;if  (feof(s0  )    &&(h--|1))goto
tau;}else goto ten; }tau:p5=mv(dd*h);for(   int i=0;  i<dd*h;i  +=1)p5[i]  =040;for   (int i
=0;i<h;i  ++){      memcpy(p5+i*dd,strs[i],          strlen(   strs[i]))  ;for(int   q=0; ((d=strs[i][q])
||1)&&q<dd&&((d==           64?s=i*dd+q:d==33?e=i   *dd+q:d==   10?p5      [i*dd+q]= 040:13)||1);q++); }
if(s==-1||e==-1)goto ten;   goto pic;}china:;for(int r=0;r<sz;        r++){z1(_int+ 25,1+(r
/dd),1+r%dd);p0(p5[r]);}   trundle=mv(sizeof(int)*sz);vp5=mv(sizeof(int)*sz);nqd =   //
mv(sizeof(int)*sz);trundle [s]=0;hh[0]=s;qf=1;qq_=0;goto tokyo;pio:;          //!     //
(_int[0]=27)&&(_int[1]=91)&&(_int[2]=48)&&(_int[3]=109)&&(_int[4]=10)&&(_int[5]=0)&&z1(_int);}
```

Credit: Joshua Karns

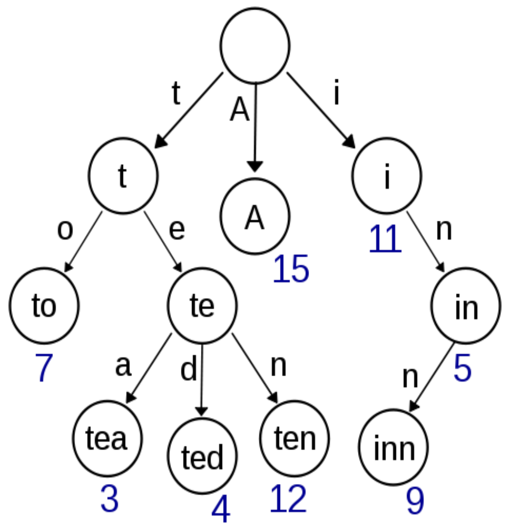
Alright... Enough administration!

Let's learn about...

Algorithms



Data Structures



What is an Algorithm

An algorithm is a set of instructions/steps that solves a particular problem.

Each algorithm operates on input data.

Each algorithm produces an output result.

Each algorithm can be classified by how long it takes to run on a particular input.

Each algorithm can be classified by the quality of its results.

Algorithm Comparison
"Guess My Word"

Let's say you're building a word guessing game.

The user secretly picks a random word from the dictionary.

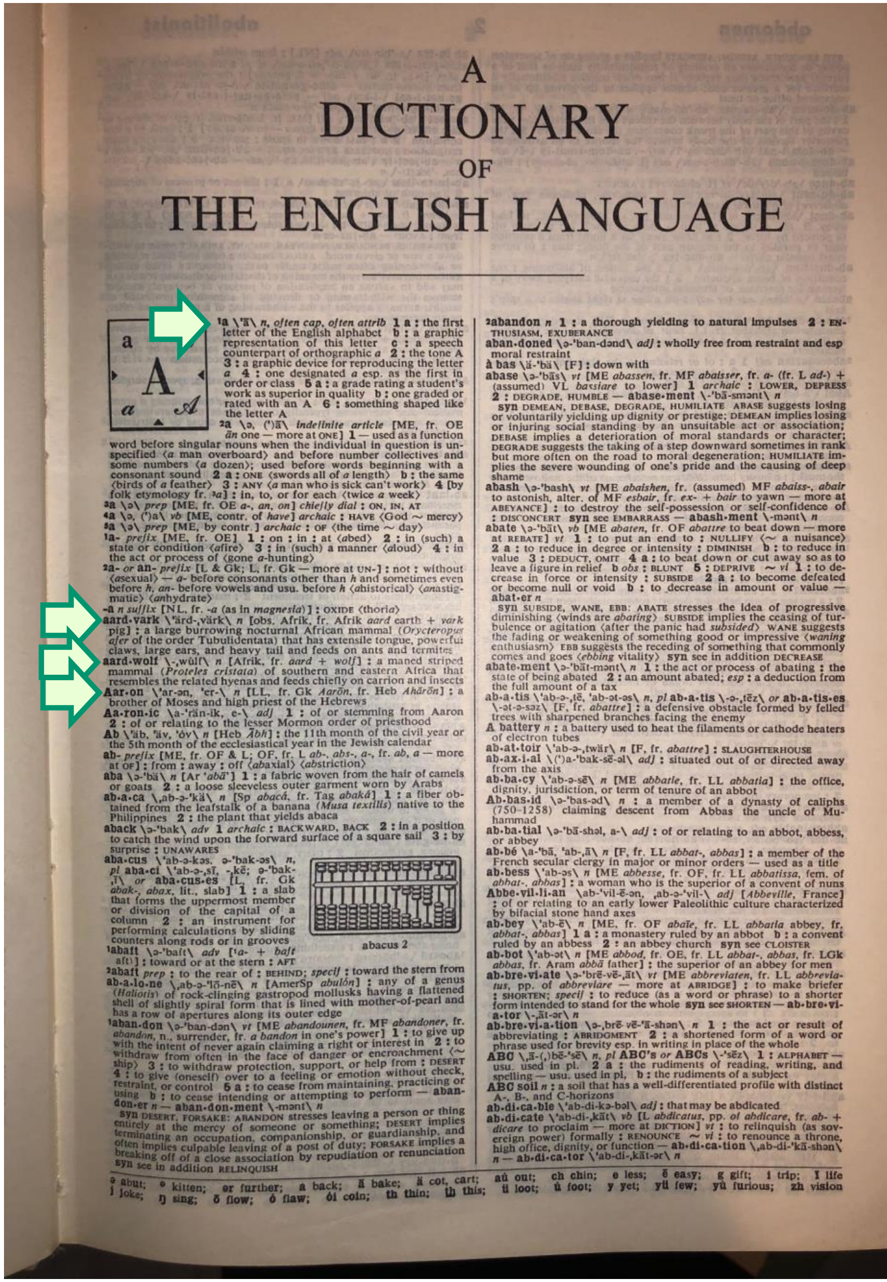
Our program then must discover the word the user picked as quickly as possible.

Let's consider two different algorithms…

Algorithm #1: Linear Search

Let's try a simple algorithm…

1. Ask the user to think of a random word
2. Start at the first word in the dictionary
3. While we haven't found the user's word:
    a. Ask: "Is that your word?"
    b. If not, advance to the next word



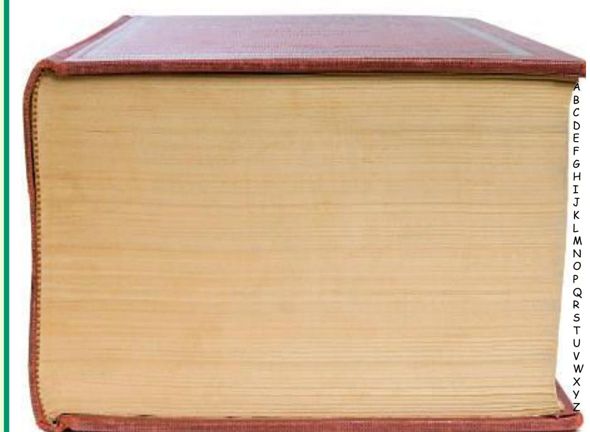
Question: If there are 100,000 total words in our dictionary, on average, how many guesses will our algorithm require?

Ok, let's try it.

Algorithm #2: Binary Search

Alright, for our second strategy let's try a more intelligent approach called binary search:

1. Print: "Think of a random word"
2. While we haven't found the word:
   a. Pick the middle word in the dictionary
   b. Ask: "Does your word come before or after that word?"
   c. If their word comes before, throw away the 2nd half of the dictionary
   d. If their word comes after, throw away the 1st half of the dictionary



See how it works? If the user says their word comes before the middle word in the dictionary, we can immediately rule out the second half of the dictionary! We know the user's word must in the first half of the dictionary. Then we can repeat this process for the remaining first half of the dictionary, and so on. Eventually we'll arrive at a single word, which MUST be the user's word!

Question: If there are 100,000 total words in our dictionary, on average, how many guesses will our Binary Search algorithm require? [see next slide]

Binary Search: How Many Guesses?

We keep on dividing our search area in half
until we finally arrive at our word.

In the worst case, we must keep halving our search area until it contains just a single word – our word!

If our dictionary had 16 words, how many times would we need to halve it until we have just one word left?

16 8 4 2 1

It would take 4 steps

Ok, what if our dictionary had 131,072 words?

131072 65536 32768 16384 8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1

It would take just 17 steps!!! WOW!

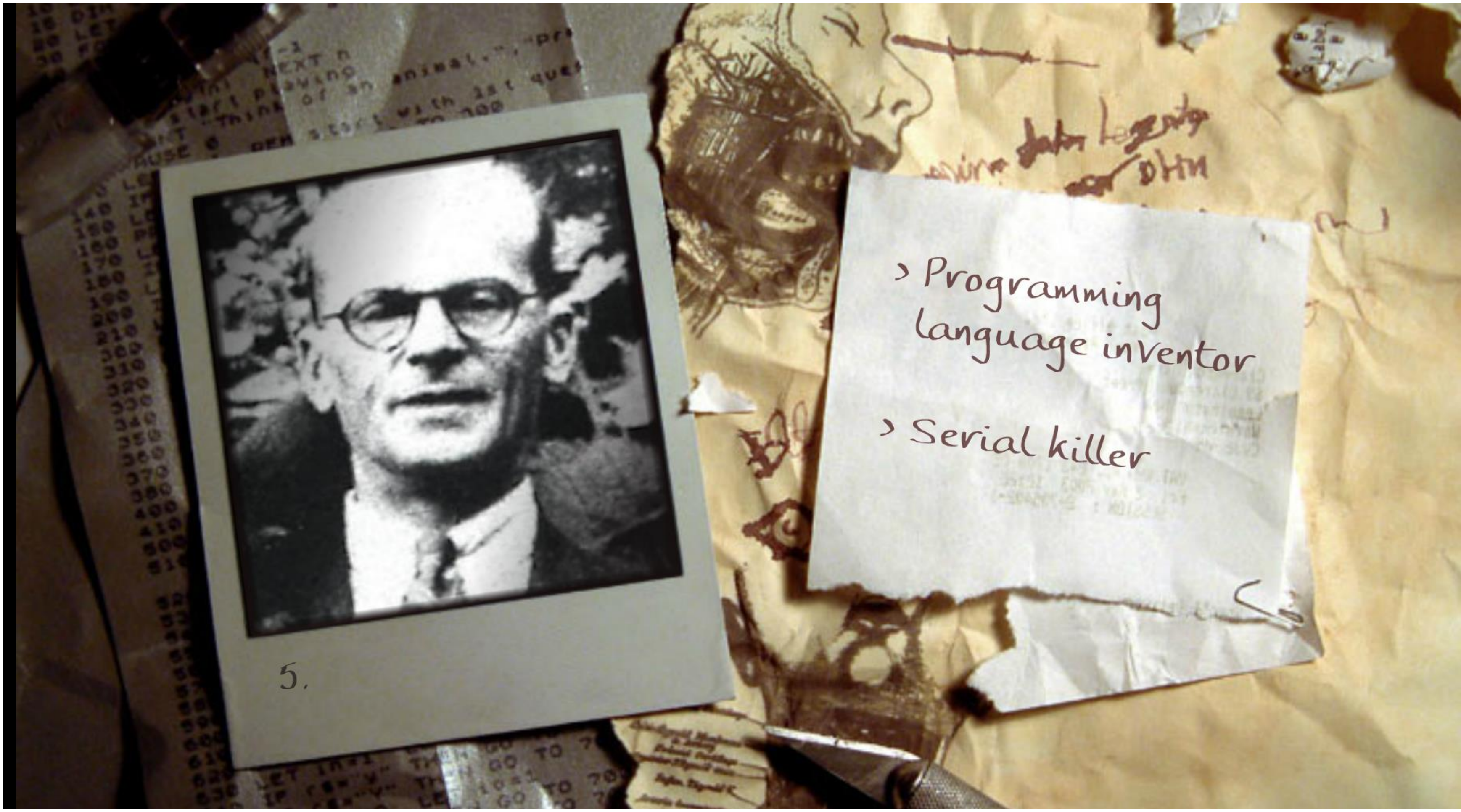
Wow! That's Significant!

Linear search requires ~50,000 steps to guess a word from a 100k word dictionary.

But our binary search algorithm requires just 17 steps, on average, to guess the user's secret word!

In CS32, you'll learn:
all the major algorithms,
how to analyze them, and
how to pick the best one

And now for a fun game!



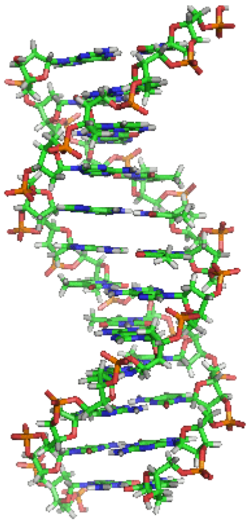
Is this guy a programming language inventor… or a SERIAL KILLER?!?!

Killed 8 women in London, was arrested after new tenants tracing an unpleasant odor peeled off the kitchen wallpaper to reveal a corpse.

Data Structures

A data structure is the set of variable(s) that an algorithm uses to solve a problem.

Let's consider a data structure to efficiently store millions of DNA sequences so they can be searched.



DNA is made up of four bases:
Adenine, Cytosine, Guanine and Thymine

A Data Structure For DNA

If I wanted to store millions of bacterial DNA sequences, each with 10 bases:

atggacatct
acattaacga
caccctcacc
accgtagaat
caccccagct
caccgaaatt
...

What simple data structure might I use?

A Data Structure For DNA

Right! You could use a sorted array of strings!

```
string dna_seq[10000000];
```

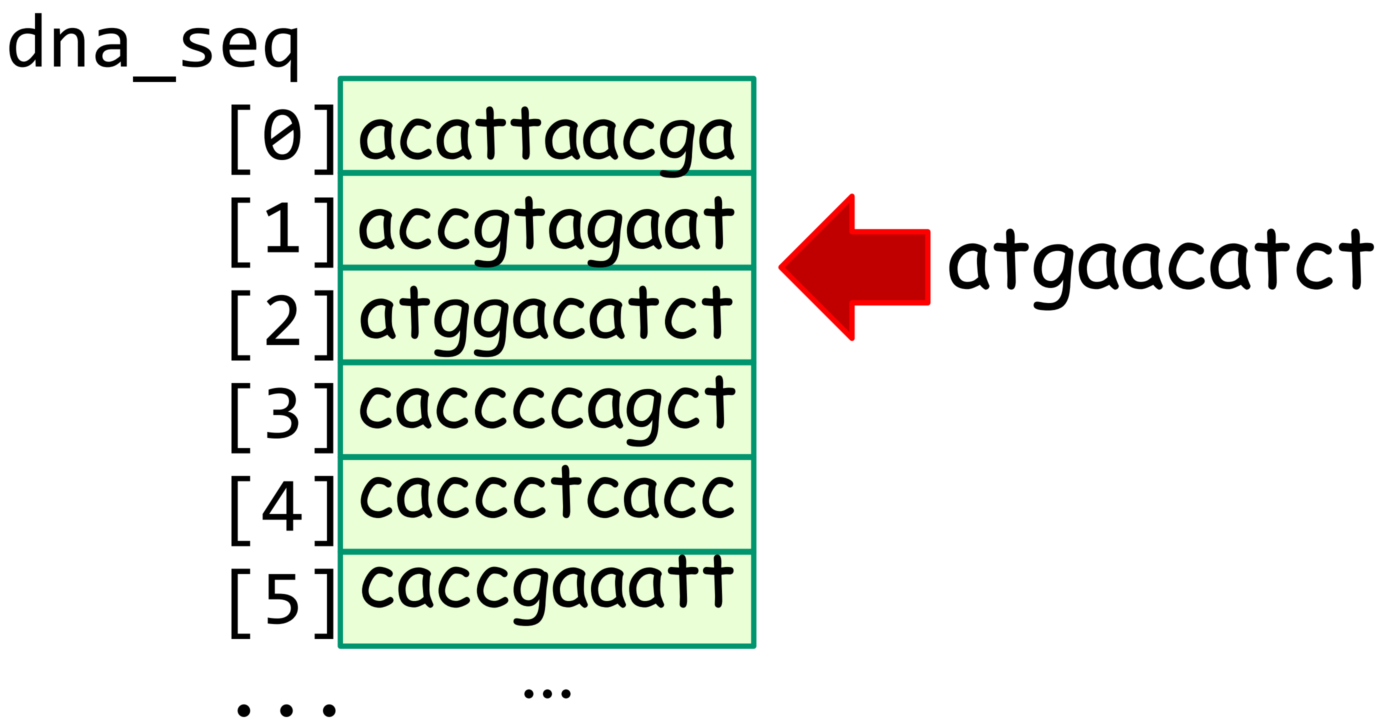
dna_seq

[0]	acattaacga
[1]	accgtagaat
[2]	atggacatct
[3]	caccccagct
[4]	caccctcacc
[5]	caccgaaatt
...	...

Not bad. Given a new sequence, I could use binary search to determine if its is held in our data structure.

A Data Structure For DNA

Are there any drawbacks of the array data structure?



Right! If we wanted to add even one new sequence, we'd have to move millions of items, and that'd be slow.

A Data Structure For DNA

Are there any other drawbacks of the array data structure?

dna_seq

[0]	acattaacga
[1]	accgtagaat
[2]	atggacatct
[3]	caccccagct
[4]	caccctcacc
[5]	caccgaaatt
...	...

Right! Arrays have a fixed capacity, so we're limited in how many DNA sequences we can add!

A Better Data Structure for DNA

Instead of an array, let's use a Trie data structure.

A Trie is just like a balanced mobile.

Let's see!

A Trie for DNA

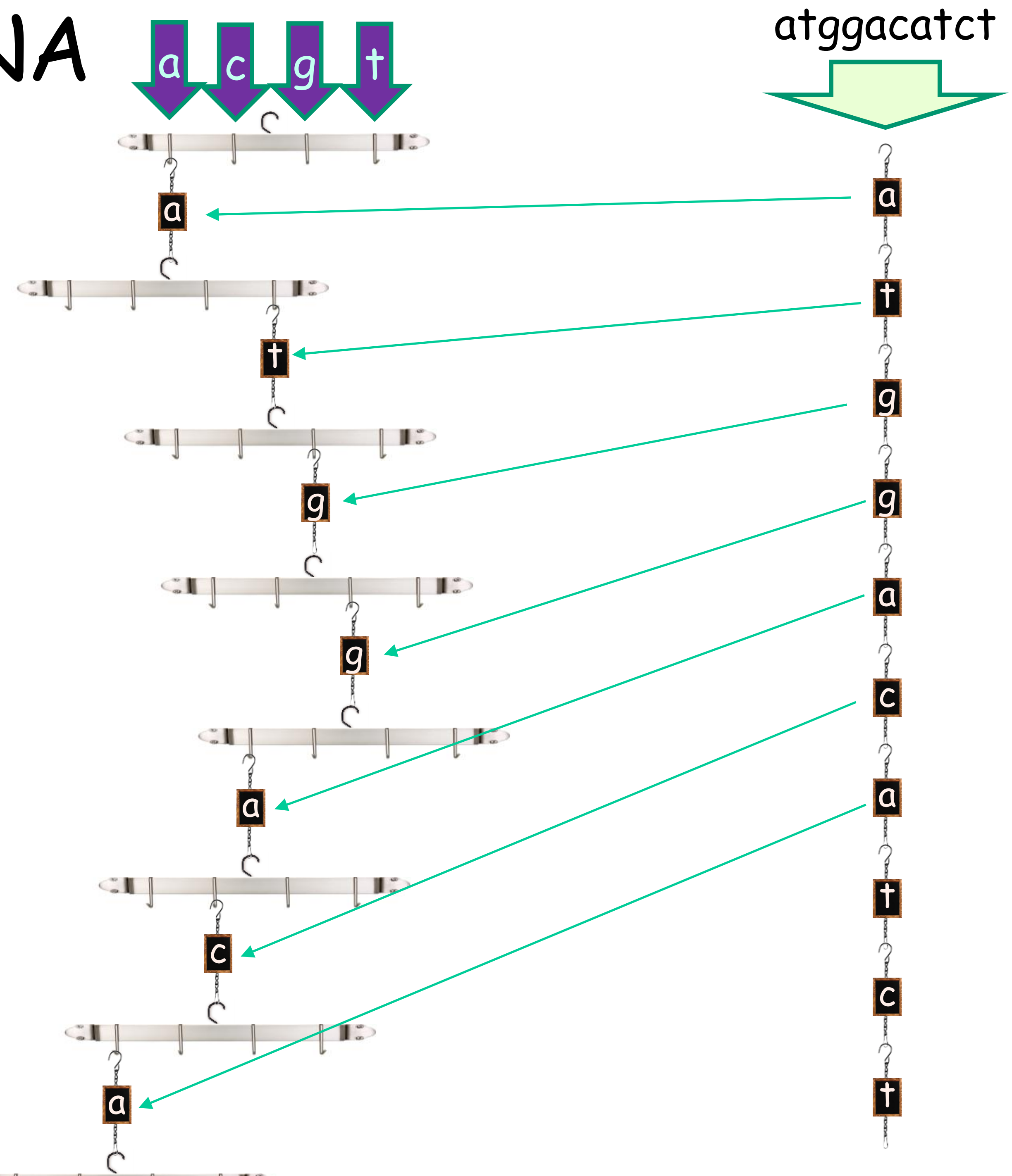
Imagine each DNA sequence dangling from ten linked chains.

Now imagine a hanging bar with four hooks, one hook for A, one for C, one for G and one for T.

Starting with the first base of the sequence, hang its chain from the proper hook...

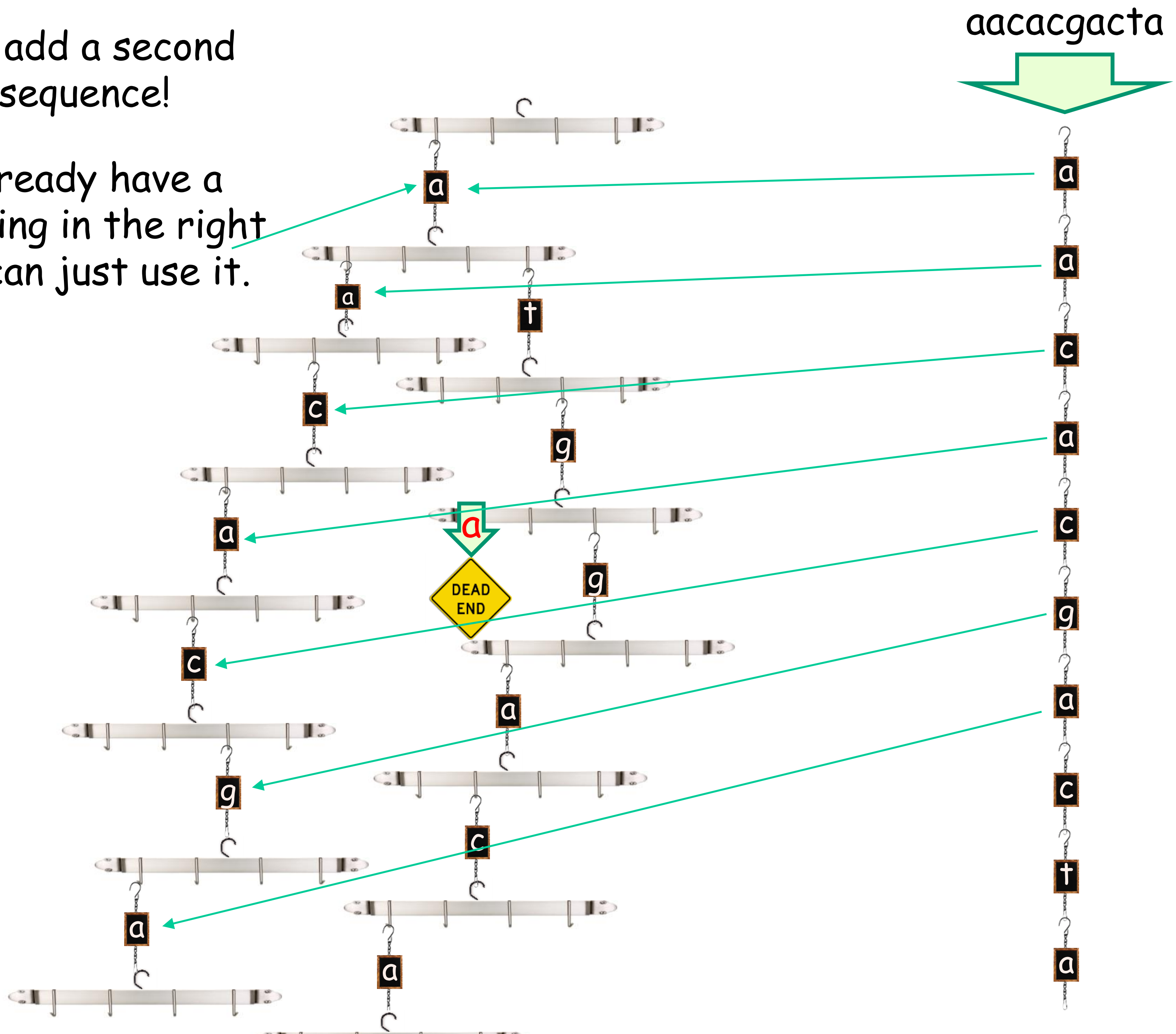
Add a new bar under the chain.

Repeat for all bases.



Ok, let's add a second DNA sequence!

If we already have a chain hanging in the right spot, we can just use it.

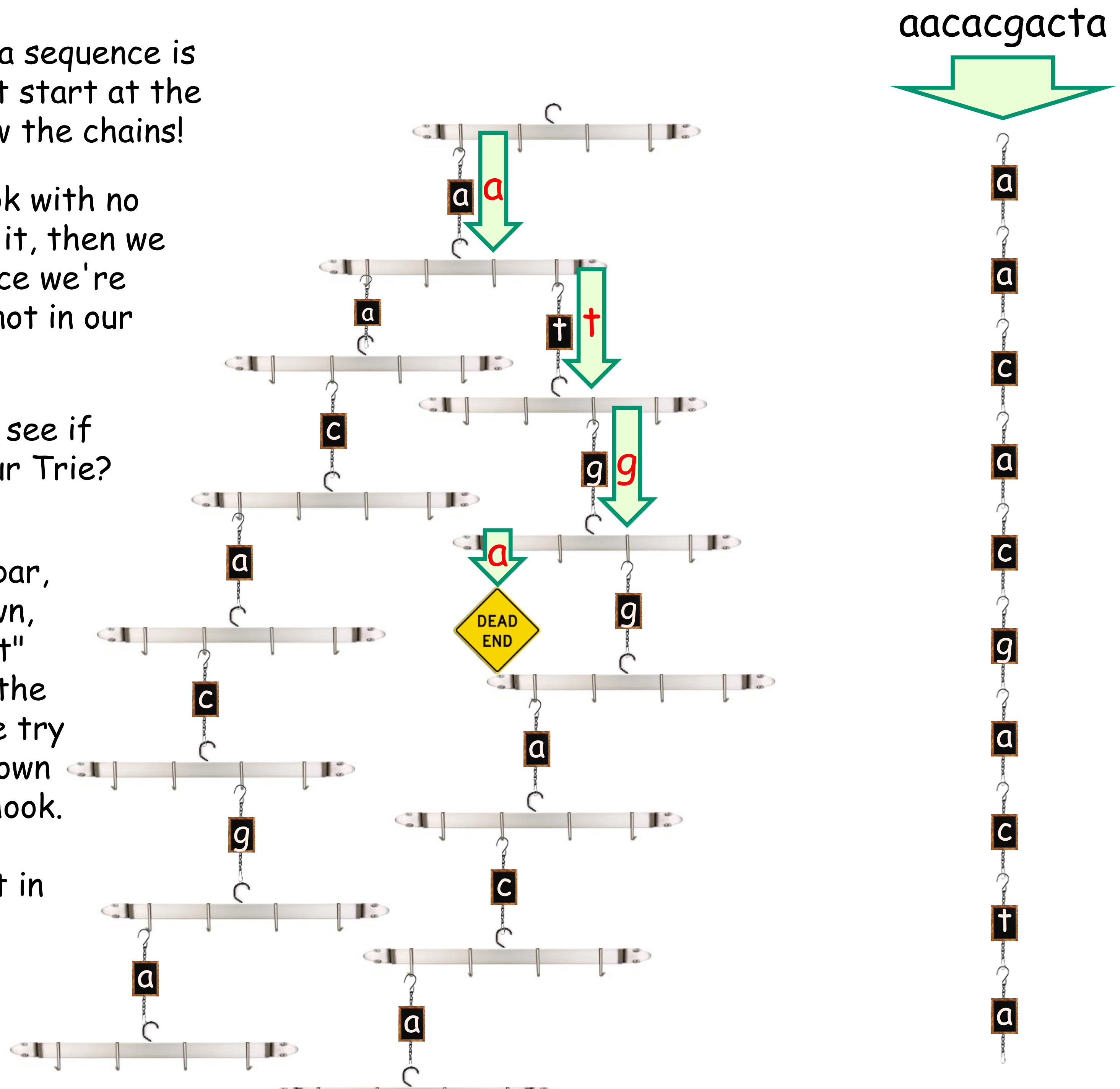


Now, to search if a sequence is in our Trie, we just start at the top bar and follow the chains!

If we reach a hook with no chain hanging from it, then we know the sequence we're searching for is not in our Trie!

For instance, let's see if "atgattacgt" is in our Trie?

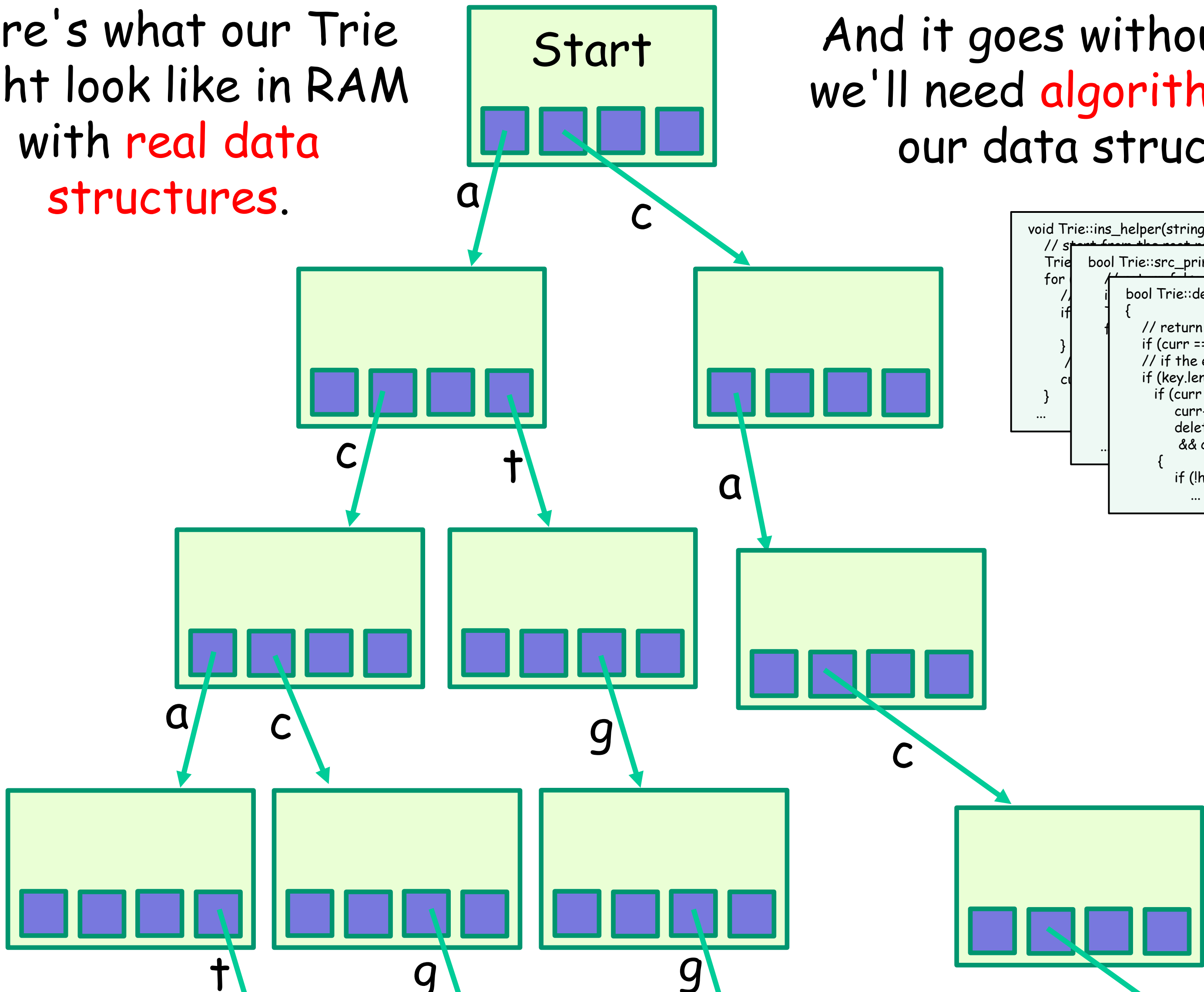
We start the top bar, follow the "a" down, then follow the "t" down, then follow the "g" down. Finally, we try to follow the "a" down and find an empty hook. The sequence "atgattacgt" is not in our Trie!



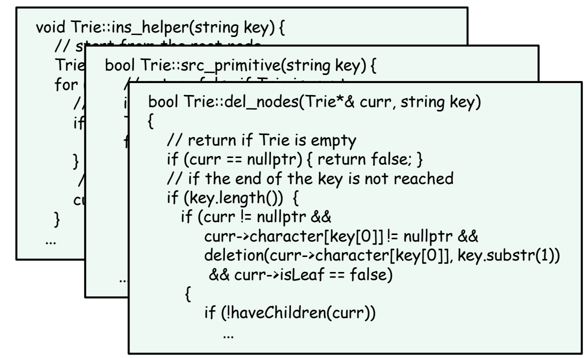
A Trie for DNA

Enough analogies with chains!

Here's what our Trie might look like in RAM with real data structures.



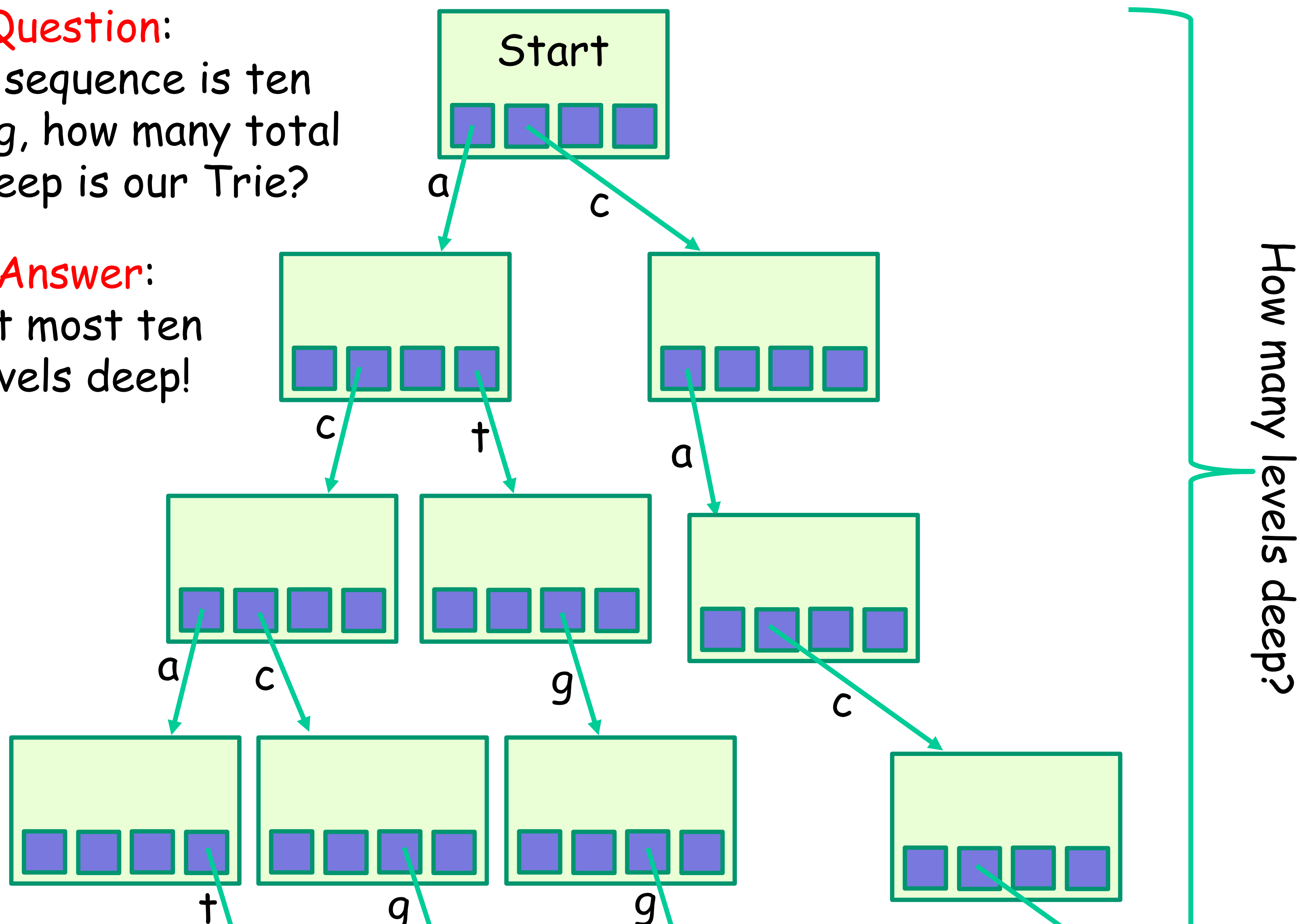
And it goes without saying that we'll need algorithms to process our data structure too...



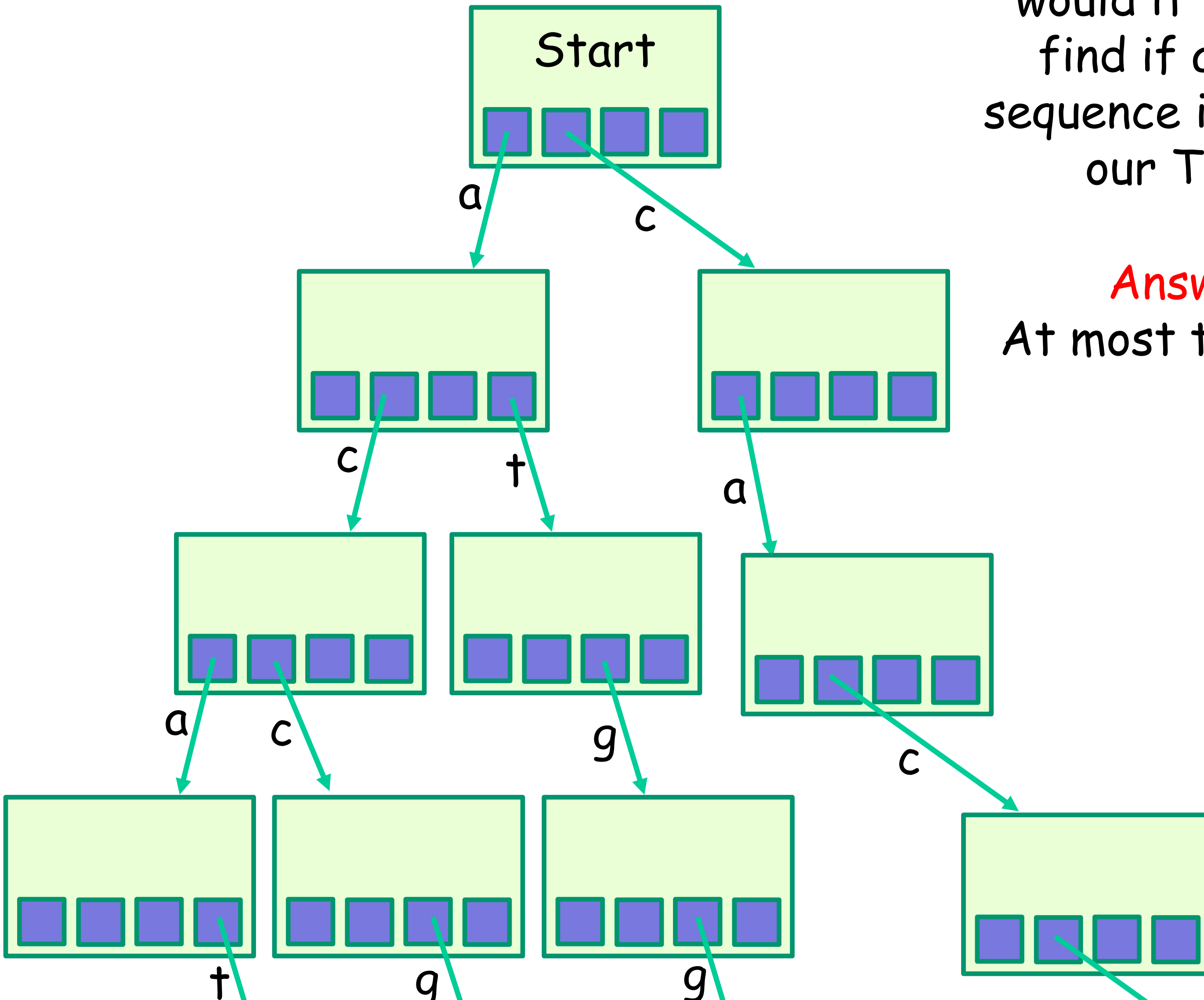
A Trie for DNA

Question:
If each sequence is ten bases long, how many total levels deep is our Trie?

Answer:
At most ten levels deep!



A Trie for DNA

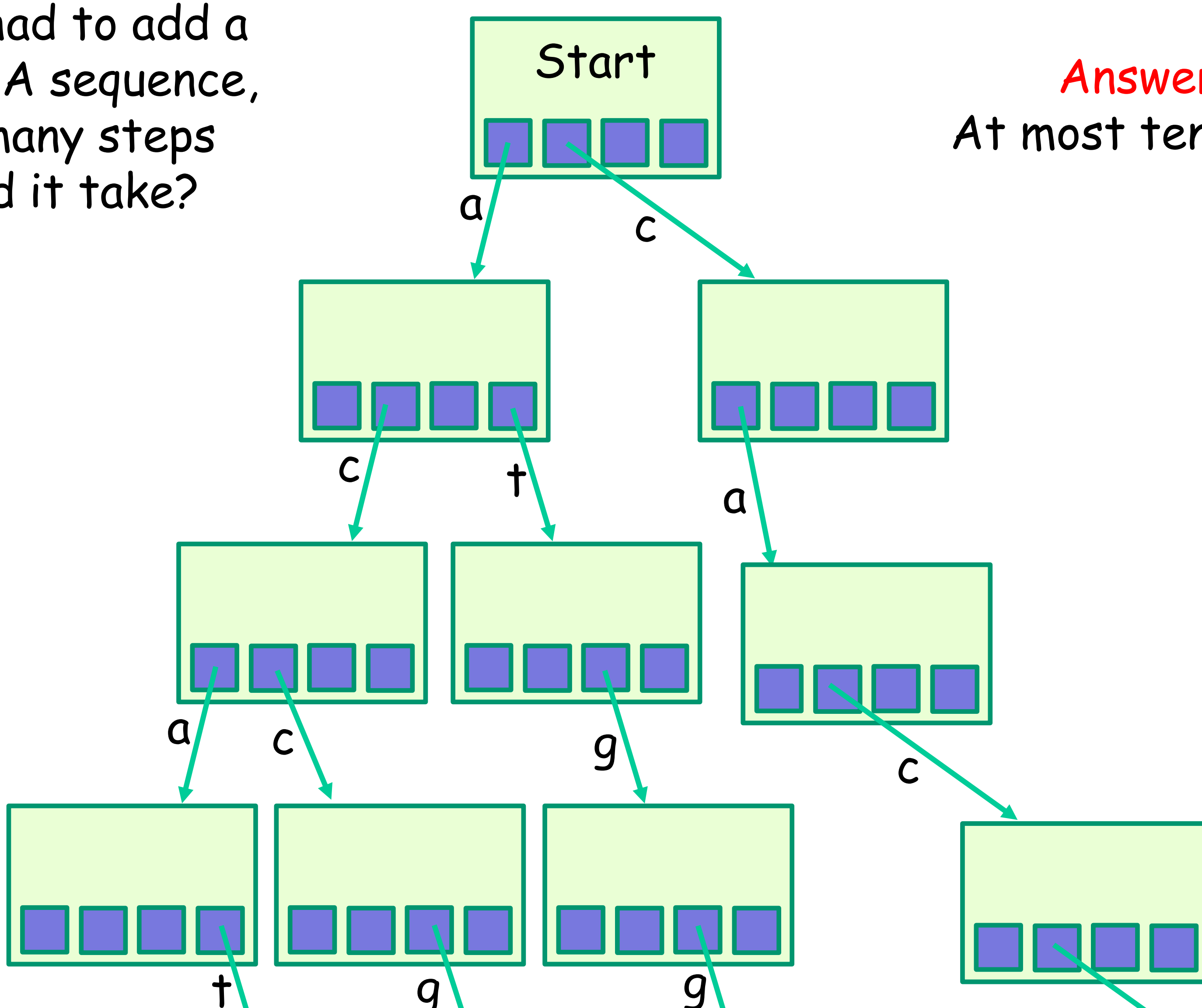


Question:
How many steps would it take to find if a DNA sequence is held in our Trie?

Answer:
At most ten steps!

A Trie for DNA

Question:
If we had to add a
new DNA sequence,
how many steps
would it take?



Answer:
At most ten steps!

Sorted Array vs. Trie

Sorted Array	Trie
Limited capacity	Infinitely expandable
Need to move millions of items every time we add an item	Just ten steps to add a new item!
Searchable in dozens of steps	Searchable in ten steps every time
Easy to implement	More complex to implement

Important point: Always choose the simplest data structure possible that meets your project's requirements. It's never a good idea to show off with a more complex data structure if you can use a simpler, easier-to-code, easier-to-understand one!

Data Structures

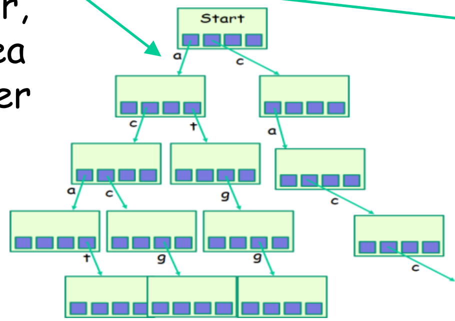
As we can see, the right data structure can make your algorithms far more efficient!

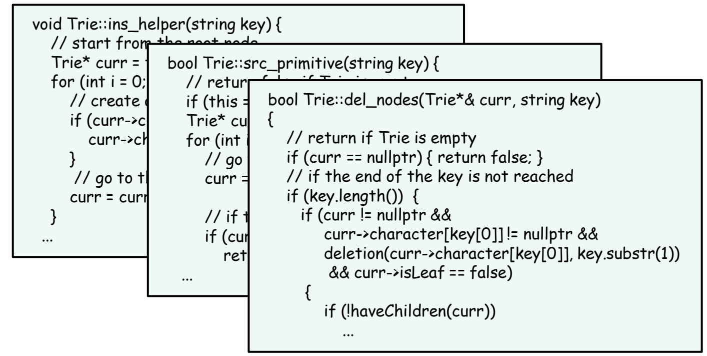
So in CS32, we'll also learn all of the most efficient data structures!

Data Structures + Algorithms = Confusion!

As we've seen, your data structures and algorithms can get quite complex.

If you gave your Trie code to another programmer, they would have no idea how to use it! It's super complex!





So it always helps to also create a few simple functions that hide the gory details…

```
bool findSequence(string dna_sequence);
void addSequence(string dna_sequence);
void deleteSequence(string dna_sequence);
```

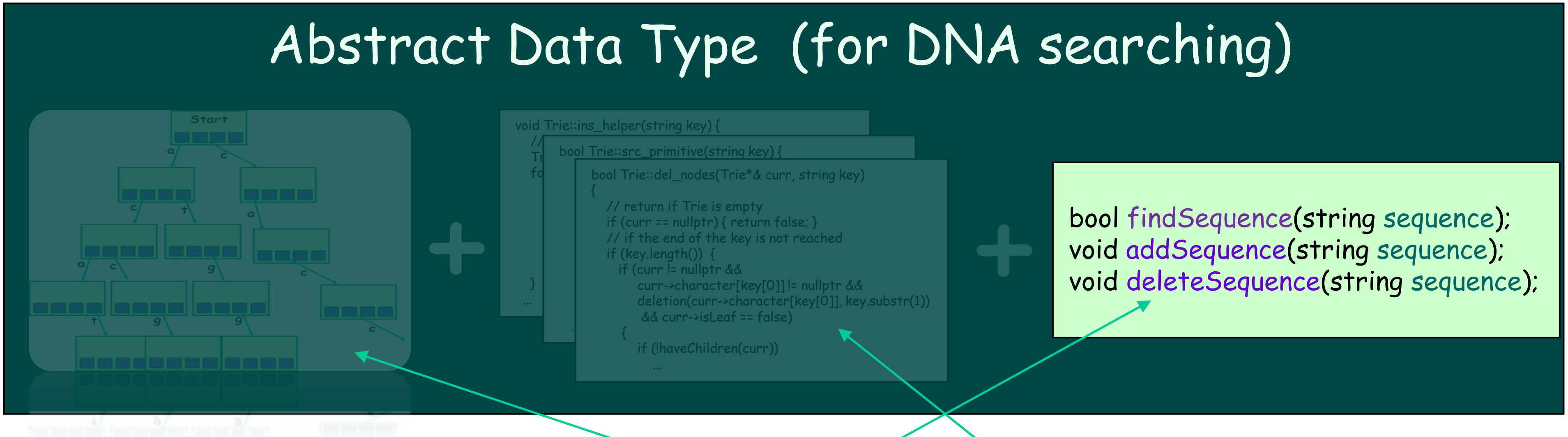
Such a collection of simple functions is called an "interface."

An interface lets any programmer use your code without having to dive into your complex data structures or logic.

```
int main()
{
    addSequence("tgaccagact");
    if (findSequence("gcgttaacac") == true)
        cout << "This is a known DNA snippet!\n";
}
```

The Abstract Data Type (ADT)

In CS, we call a coordinated group of data structures, algorithms and interface functions an Abstract Data Type.



In an ADT, the data structures and algorithms are secret.

The ADT provides an interface (a simple set of functions) to enable the rest of the program to use it.

Typically, we build programs from a collection of ADTs, each of which solves a different sub-problem.

ADTs in C++

In C++, we use classes to define ADTs in our programs!
Each C++ class holds data structures, algorithms and interface functions!

Once we've defined our class, the rest of our program can use it trivially. All our program needs to do is call the functions in our class's public interface! The rest of the program can ignore the details of how our class works and just use its features!

```
int main()
{
   DNADatabase d;

   d.addSequence("gagagtcaca");
   d.addSequence("tcaggacata");
   …

   string dna_seq;

   cout << "Enter a 10-base sequence: ";
   cin >> dna_seq;
   if (d.findSequence(dna_seq) == true)
      cout << "This bacteria is known!";
}
```

```
// A C++ DNA sequencer class…
// (this is really an ADT!)

class DNADatabase
{
public:
  // our interface functions go here
  void findSequence(…);
  void addSequence(…);
    …
private:
  // secret algorithms go here
  …

 // secret data structures go here
 …

};
```

ADTs in C++


```
int main()
{
   DNADataba

   d.addSequence("gagagtcaca");
   d.addSequence("tcaggacata");
   …

   string dna_seq;

   cout << "Enter a 10-base sequence: ";
   cin >> dna_seq;
   if (d.findSequence(dna_seq) == true)
      cout << "This bacteria is known!";
}
```

Now what if I wanted to improve my class's implementation?

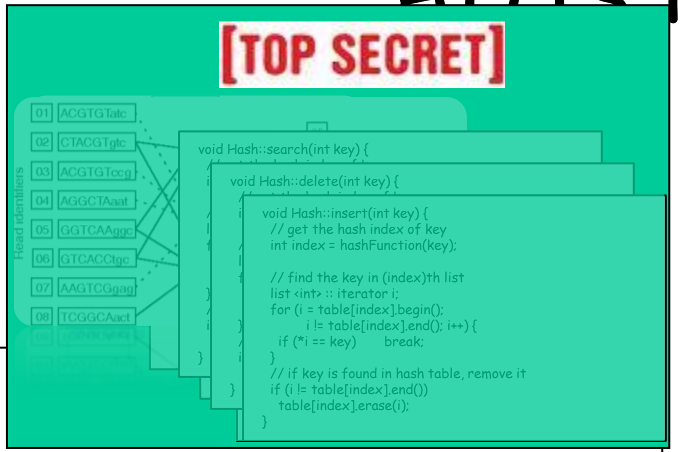
Let's say I made a radical change to my data structures & algorithms…

Would the user need to change any part of their program?

No! Because these details are hidden from the rest of our code!

The code that uses our class knows nothing about its private data structures and algorithms. All it knows how to use is the public interface.

ADTs in C++




```
int main()
{
   DNADatabase d;

   d.addSequence("gagagtcaca");
   d.addSequence("tcaggacata");
   ...

   string dna_seq;

   cout << "Enter a 10-base sequence: ";
   cin >> dna_seq;
   if (d.findSequence(dna_seq) == true)
      cout << "This bacteria is known!";
}
```

This is a huge benefit of Abstract Data Types!

We can break up our programs into small, self-contained ADTs...

And combine these smaller parts together to solve bigger problems.

What is Object Oriented Programming?

Object Oriented Programming (OOP) is simply a programming model based on the Abstract Data Type (ADT) concept we just learned!

In OOP, programs are constructed from multiple self-contained classes.

Each class holds a set of data structures and algorithms - we then access the class using a set of interface functions!

Classes talk to each other only by using public interface functions – each class knows nothing about how the others work inside.

Intermission Meme

C++ Class Review

As we've seen, a "class" is a self-contained problem solver that contains:

- Data structures
- Algorithms
- Interface functions

Since you've probably forgotten everything about classes…

Let's do a quick review of classes by defining our own Nerd class!

Defining a New Class

```
class Nerd
{
 public:
    Nerd(int stink, int IQ) {
      myStinkiness = stink;
      myIQ = IQ;
    }
    void study(int hours) {
      myStinkiness += 3*hours;
      myIQ *= 1.01;
    }
    int getStinkyLevel() {
       int total_stink = myIQ * 10 +
              myStinkiness;
       return total_stink;
    }
 private:
    int myStinkiness, myIQ;

};
```

Don't forget the semicolon!

First, we write the outer shell of our class and give it a name.

Then we define our class's public interface functions...

Then we define our class's private variables and functions...

Our class defines an entirely new data type, like string, that we can now use in our program.

Alert: Nerd is not a variable! It's a new C++ data type!

Using a New Class

nerd.h

```
class Nerd
{
 public:
   Nerd(int stink, int IQ) {
     myStinkiness = stink;
     myIQ = IQ;
   }
   void study(int hours) {
     myStinkiness += 3*hours;
     myIQ *= 1.01;
   }
   …
 private:
   int myStinkiness, myIQ;
};
```

ucla.cpp

```
#include "nerd.h" // #1

int main()
{
   int num_nerds = 1;
   Nerd david(30, 150); // #2

   david.study(10); // #3
}
```

Once we define a new class, like Nerd, we can use it to define variables like any traditional data type.

- The Nerd class defines a new data type like int, float, or string
- You typically define each new class in its own .h file ("header file") and put the file in the same folder as your .cpp files.
- A header file is similar to a .cpp file except you typically only put class declarations and constants in it (you typically put the actual class function {bodies} in your cpp file).
- To use your new class, simply include its header file using "quotation marks" (#1)
- You can then define variables with it throughout your program (#2).
- On line #2, david is a Nerd variable with an initial stinkiness of 30 and an IQ of 150.
- Once you've defined your variable (#3) you can call its member functions, like study.

Using a New Class

nerd.h

```
class Nerd
{
 public:
   Nerd(int stink, int IQ) {
     myStinkiness = stink;
     myIQ = IQ;
   }
   void study(int hours) {
     myStinkiness += 3*hours;
     myIQ *= 1.01;
   }
   ...
 private:
   int myStinkiness, myIQ;
};
```

ucla.cpp

```
#include "nerd.h"

int main()
{
   int num_nerds = 1;
   Nerd david(30, 150); // #1

   david.study(10);
}
```

Alright, let's see our class in action!

num_nerds

david

```
Nerd(int stink, int IQ) {
  myStinkiness = stink; // #2
  myIQ = IQ;
}
void study(int hours) {
  myStinkiness += 3*hours;   // #3
  myIQ *= 1.01;
}
...
```

myStinkiness	myIQ
-79342	12338

When you define your david variable, it gets its own copy of all of the functions and member variables defined in your class! As soon as the david variable is created (#1), C++ calls the constructor function inside the variable to initialize its state. Our constructor (#2) and other member functions (#3) have access to david's private member variables (like myIQ)! Note: A class's primitive member variables (e.g. int's, doubles like myStinkiness) all start out with random values and NOT zero! Your constructor must initialize them.

nerd.h

```
class Nerd
{
 public:
   Nerd(int stink, int IQ) {
     myStinkiness = stink;
     myIQ = IQ;
   }
   int getStinkyLevel() {
     int total_stink = myIQ
          * 10 + myStinkiness;
     return total_stink;
   }
 private:
   int myStinkiness, myIQ;
};
```

ucla.cpp

```
#include "nerd.h"

int main()
{
   int num_nerds = 1;
   Nerd david(30, 150);

   david.study(10);
}
```

Other Details

You typically only use member variables to store permanent attributes of your class.

- Stinkiness and IQ are inherent attributes of every Nerd, so we make these member variables.
- If you're using a variable for a temporary computation (like total_stink to the left), then just use a local variable for that.
- Never use member variables for temporary computation. Only use member variables to store values that you expect your object to retain over time (like the IQ of a nerd, their phone number, or address).

Other Details

nerd.h

```
class Nerd
{
 public:
   Nerd(int stink, int IQ) {
     myStinkiness = stink;
     myIQ = IQ;
   }
   void study(int hours) {
     myStinkiness += 3*hours;
     myIQ *= 1.01;
   }
   ...
 private:
   int myStinkiness, myIQ;
};
```

ucla.cpp

```
#include "nerd.h"

int main()
{
   int num_nerds = 1;
   Nerd david(30, 150);

   david.study(10);  // #1
}
```

- All functions in the public section of your class (like the constructor or study()) can be seen/called by all parts of your program.
- All functions and data defined in the private section of your class (like myStinkiness) are hidden from the rest of your program. They may only be used by other functions defined in the Nerd class.
- Notice how the main function calls david.study(10) on line #1. This is legal because the study() method is in Nerd's public section.
- Also notice that your Nerd constructor and study method may access/change the Nerd's private member variables (like myIQ).
- However, other functions outside of your class, like main(), are forbidden from accessing these private member variables. If your main() function were to try to cout << david.myIQ; this would result in an error.
- Hiding the internal implementation details of a class is called "encapsulation."
- Encapsulation makes your program simpler, since each class works without knowledge of how the other classes work internally.
- So we can change how one class works internally (in CS lingo, we "refactor" it), and the other classes will continue to work as-is without any changes!