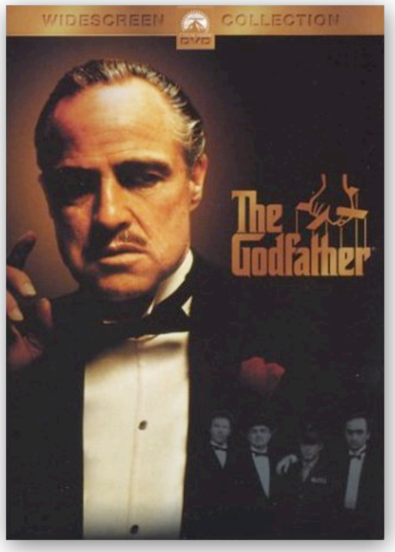
CMSC 420

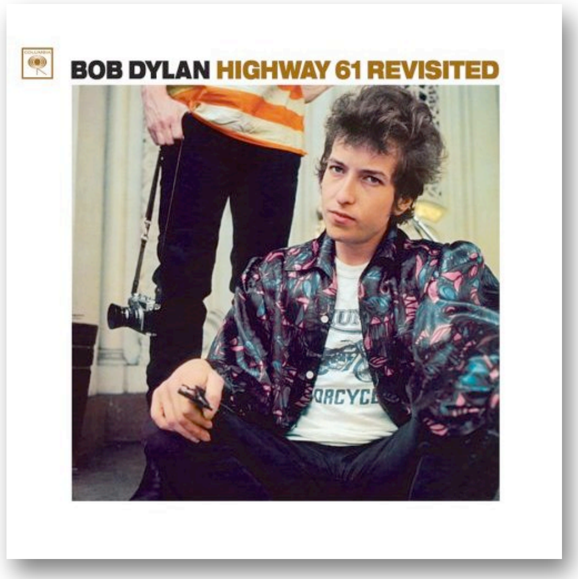
Data Structures

Lecture 1: Introduction

Digital Data



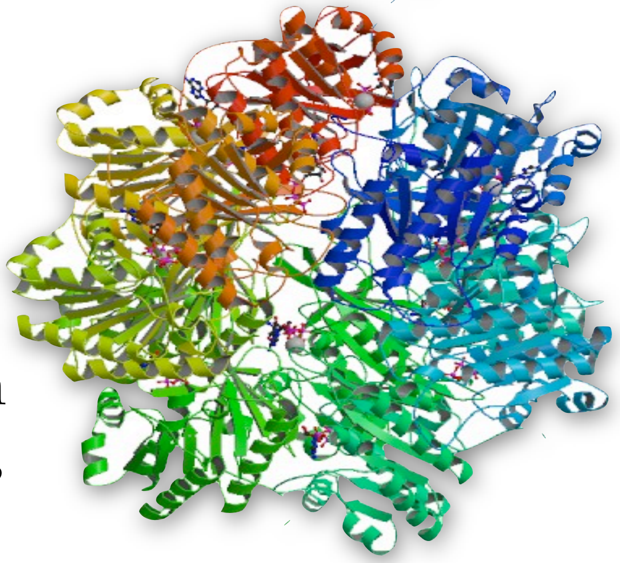
Movies



Music

Photos

Protein Shapes



Maps

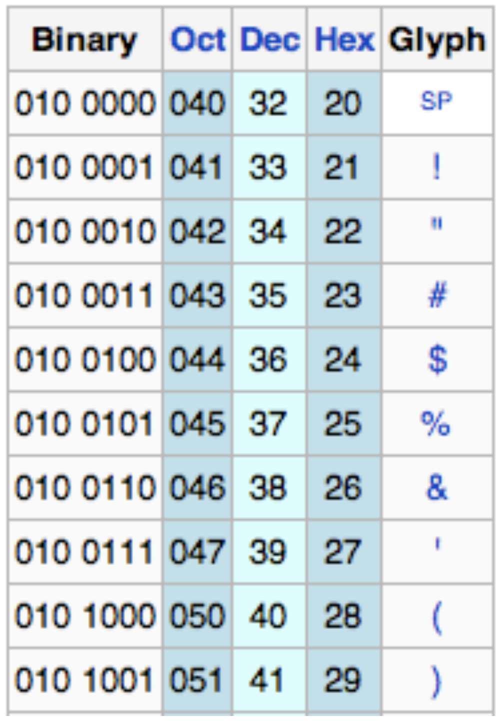
DNA

```
gatcttttta tttaaacgat ctctttatta gatctcttat taggatcatg atcctctgtg
gataagtgat tattcacatg gcagatcata taattaagga ggatcgtttg ttgtgagtga
ccggtgatcg tattgcgtat aagctgggat ctaaatggca tgttatgcac agtcactcgg
cagaatcaag gttgttatgt ggatatctac tggttttacc ctgcttttaa gcatagttat
acacattcgt tcgcgcgatc tttgagctaa ttagagtaaa ttaatccaat ctttgaccca
```

0010101001010101010100100100101010000010010010100....

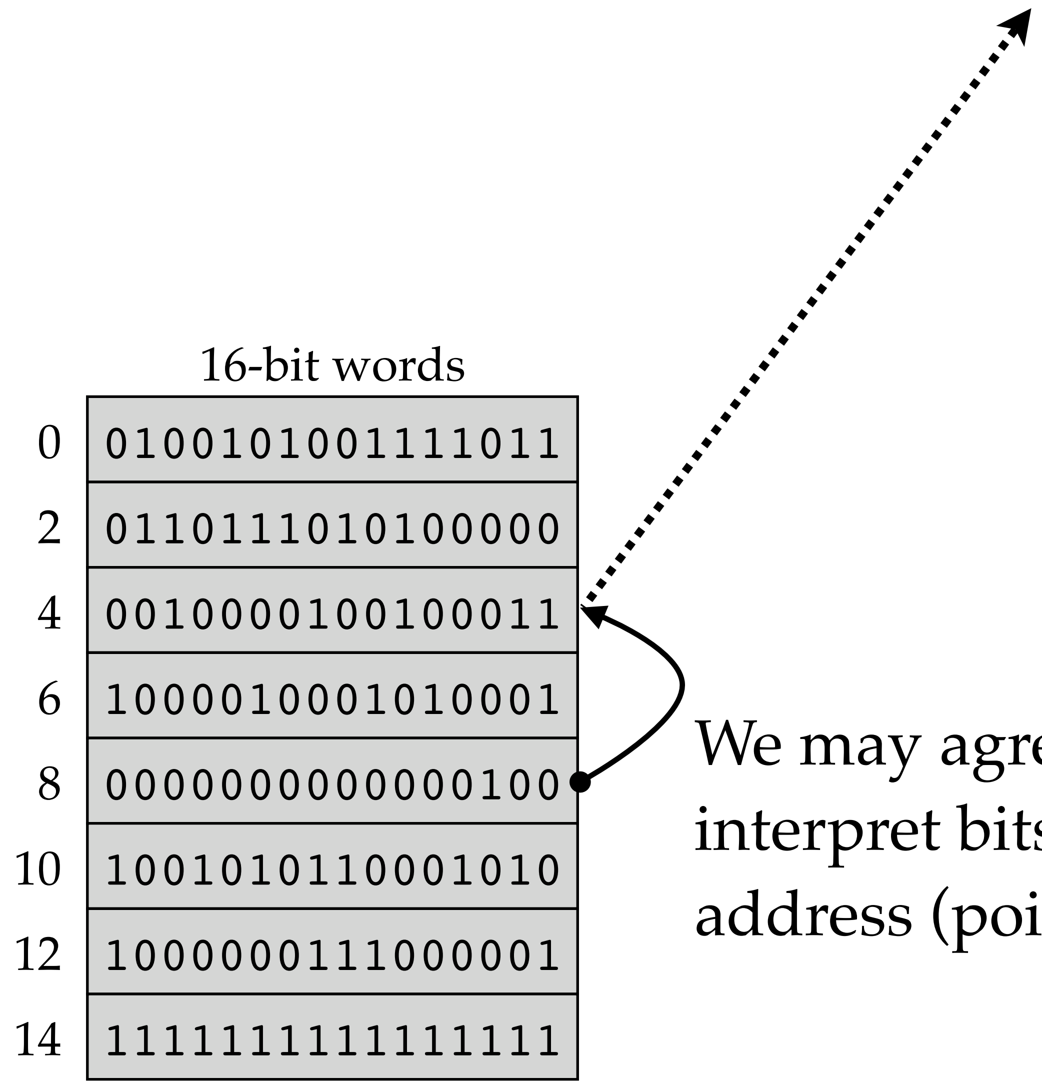
RAM = Symbols + Pointers

(for our purposes)



Binary	Oct	Dec	Hex	Glyph
010 0000	040	32	20	SP
010 0001	041	33	21	!
010 0010	042	34	22	"
010 0011	043	35	23	#
010 0100	044	36	24	$
010 0101	045	37	25	%
010 0110	046	38	26	&
010 0111	047	39	27	'
010 1000	050	40	28	(
010 1001	051	41	29	)

ASCII table: agreement for the meaning of bits



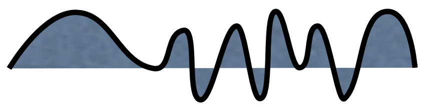
	16-bit words
0	0100101001111011
2	0110111010100000
4	0010000100100011
6	1000010001010001
8	0000000000000100
10	1001010110001010
12	1000000111000001
14	1111111111111111

We may agree to interpret bits as an address (pointer)

Physically, RAM is a random accessible array of bits.

=> We can store and manipulate arbitrary symbols (like letters) and associations between them.

Digital Data Must Be ...

- Encoded (e.g. 01001001 <-> )

- Arranged
  - Stored in an orderly way in memory / disk

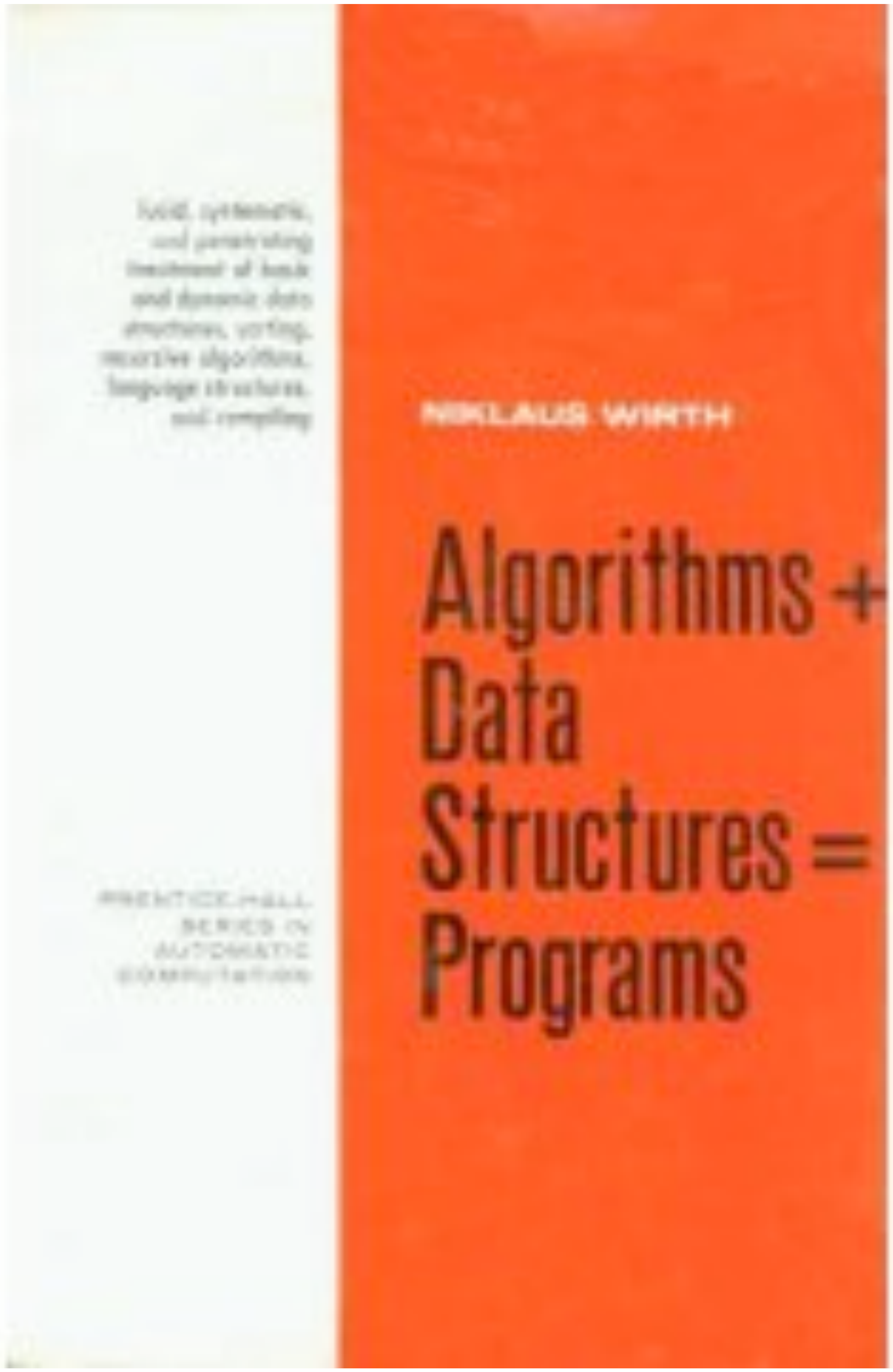
- Accessed
  - Insert new data
  - Remove old data
  - Find data matching some condition

$\left.\right\}$ The focus of this class

- Processed
  - Algorithms: shortest path, minimum cut, FFT, ...

Data Structures -> Data StructurING

How do we organize information so that we can find, update, add, and delete portions of it efficiently?



lucid, systematic, and penetrating treatment of basic and dynamic data structures, sorting, recursive algorithms, language structures, and compiling

PRENTICE-HALL SERIES IN AUTOMATIC COMPUTATION

NIKLAUS WIRTH

Algorithms + Data Structures = Programs

Niklaus Wirth, designer of Pascal

Data Structure Example Applications

1. How does Google quickly find web pages that contain a search term?
2. What's the fastest way to broadcast a message to a network of computers?
3. How can a subsequence of DNA be quickly found within the genome?
4. How does your operating system track which memory (disk or RAM) is free?
5. In the game Half-Life, how can the computer determine which parts of the scene are visible?

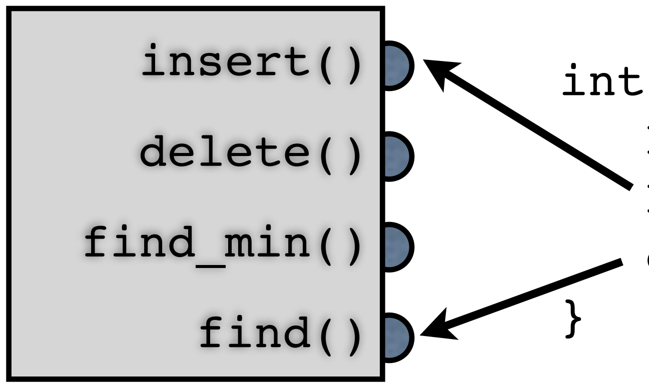
What is a Data Structure Anyway?

- It’s an agreement about:
  - how to store a collection of objects in memory,
  - what operations we can perform on that data,
  - the algorithms for those operations, and
  - how time and space efficient those algorithms are.

- Ex. `vector` in C++:
  - Stores objects sequentially in memory
  - Can access, change, insert or delete objects
  - Algorithms for insert & delete will shift items as needed
  - Space: $O(n)$, Access/change = $O(1)$, Insert/delete = $O(n)$

Abstract Data Types (ADT)

```
class Dictionary {
   Dictionary();
   void insert(int x, int y);
   void delete(int x);
   ...
}
```



```
int main() {
   D = new Dictionary()
   D.insert(3,10);
   cout << D.find(3);
}
```

- Data storage & operations encapsulated by an ADT.
- ADT specifies permitted **operations** as well as **time** and **space** guarantees.
- User unconcerned with how it's implemented
  (but we are concerned with implementation in this class).
- ADT is a **concept** or **convention**:
  - not something that directly appears in your code
  - programming language may provide support for communicating ADT to users (e.g. classes in Java & C++)

Dictionary ADT

- Most basic and most useful ADT:
  - `insert(key, value)`
  - `delete(key, value)`
  - `value = find(key)`

- Many languages have it built in:

```
awk:     D["AAPL"] = 130                              # associative array
perl:    my %D; $D["AAPL"] = 130;                     # hash
python:  D = {}; D["AAPL"] = 130                      # dictionary
C++:     map<string,string> D = new map<string, string>();
         D["AAPL"] = 130;                             // map
```

- **Insert**, **delete, find** each either O(log n) [C++] or expected constant [perl, python]

- Any guesses how dictionaries are implemented?

C++ STL

- Data structures = “containers”
- Interface specifies both operations & time guarantees

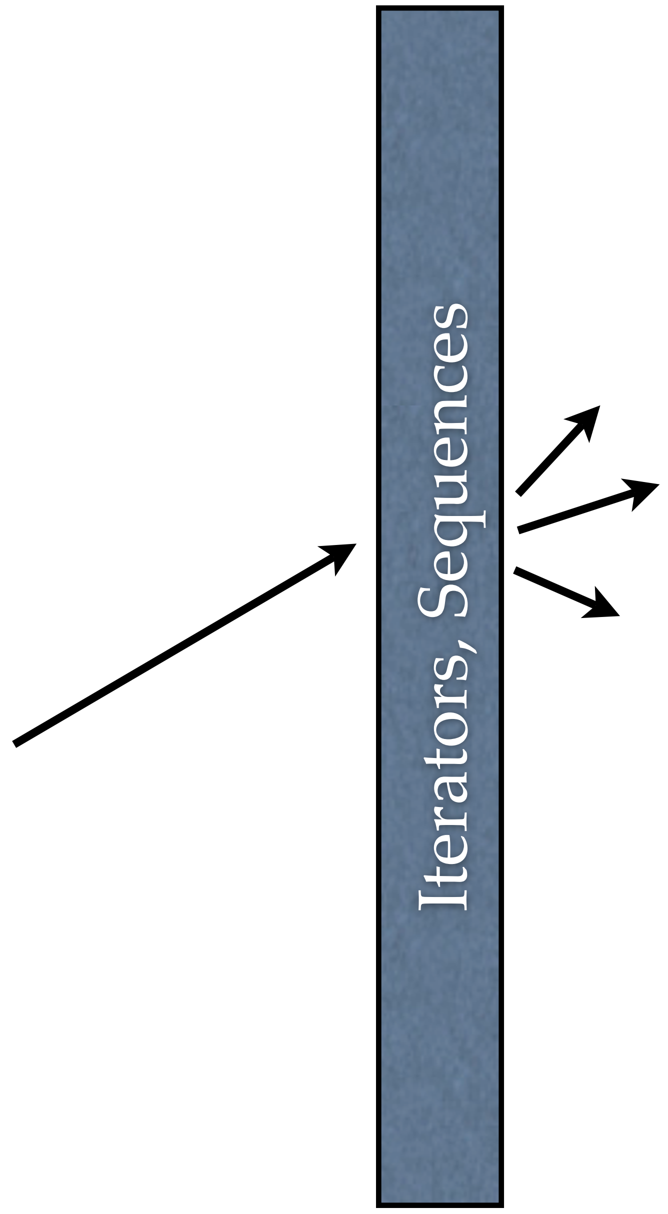
Container	Element Access	Insert / Delete	Iterator Patterns
`vector`	const	O(n)	`Random`
`list`	O(n)	const	`Bidirectional`
`stack`	const (limited)	O(n)	`Front`
`queue`	const (limited)	O(n)	`Front, Back`
`deque`	const	O(n), const @ ends	`Random`
`map`	O(log n)	O(log n)	`Bidirectional`
`set`	O(log n)	O(log n)	`Bidirectional`
`string`	const	O(n)	`Bidirectional`
`array`	const	O(n)	`Random`
`valarray`	const	O(n)	`Random`
`bitset`	const	O(n)	`Random`

Some STL Operations

- Select operations to be *orthogonal*: they don't significantly duplicate each other's functionality.
- Choose operations to be useful building blocks.

E.g. Data Structure Operations

- `push_back`
- `find`
- `insert`
- `erase`
- `size`
- `begin, end` (iterators)
- `operator[]`
- `front`
- `back`

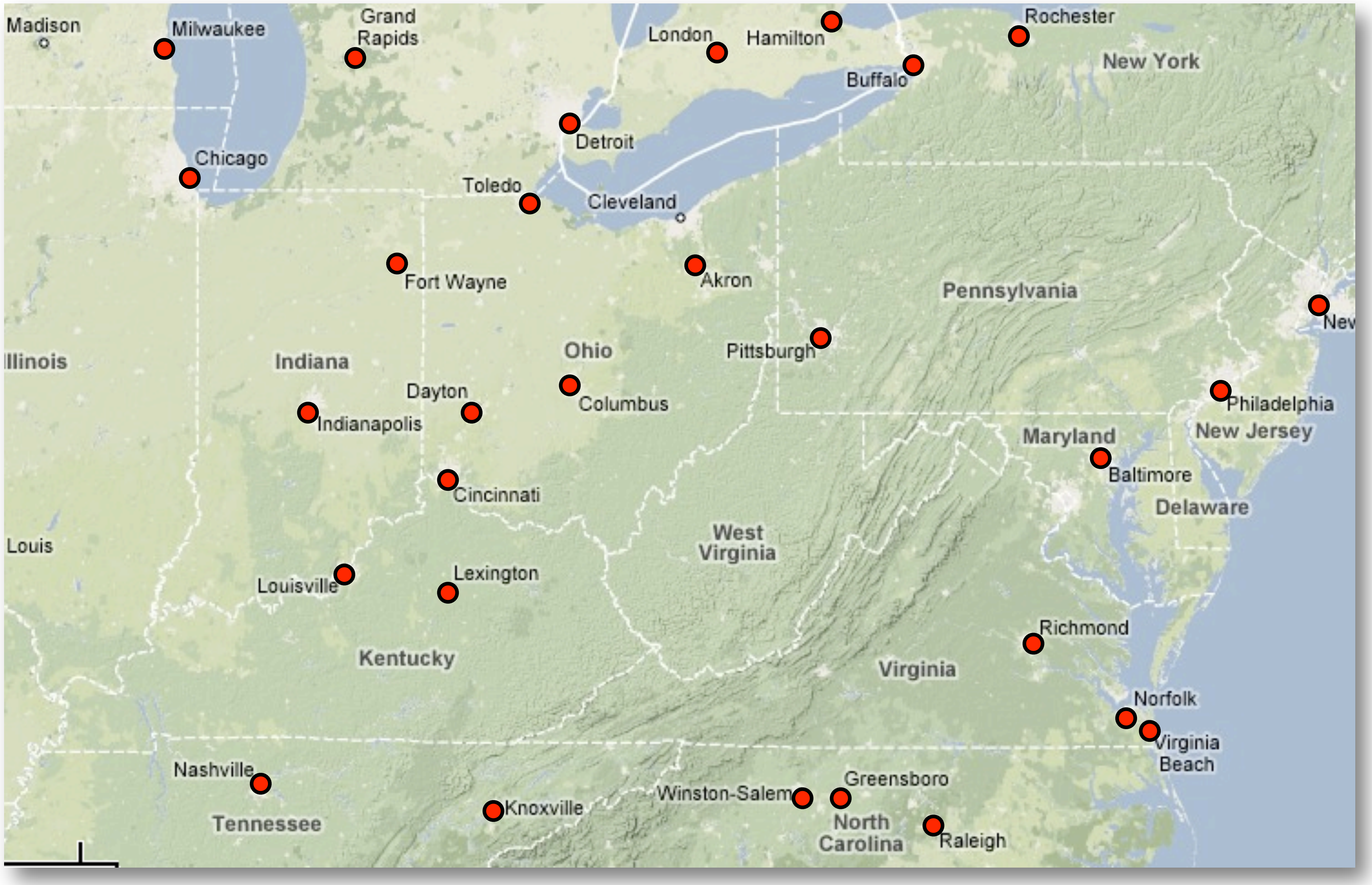


E.g. Algorithms

- `for_each`
- `find_if`
- `count`
- `copy`
- `reverse`
- `sort`
- `set_union`
- `min`
- `max`

Suppose You're Google Maps...

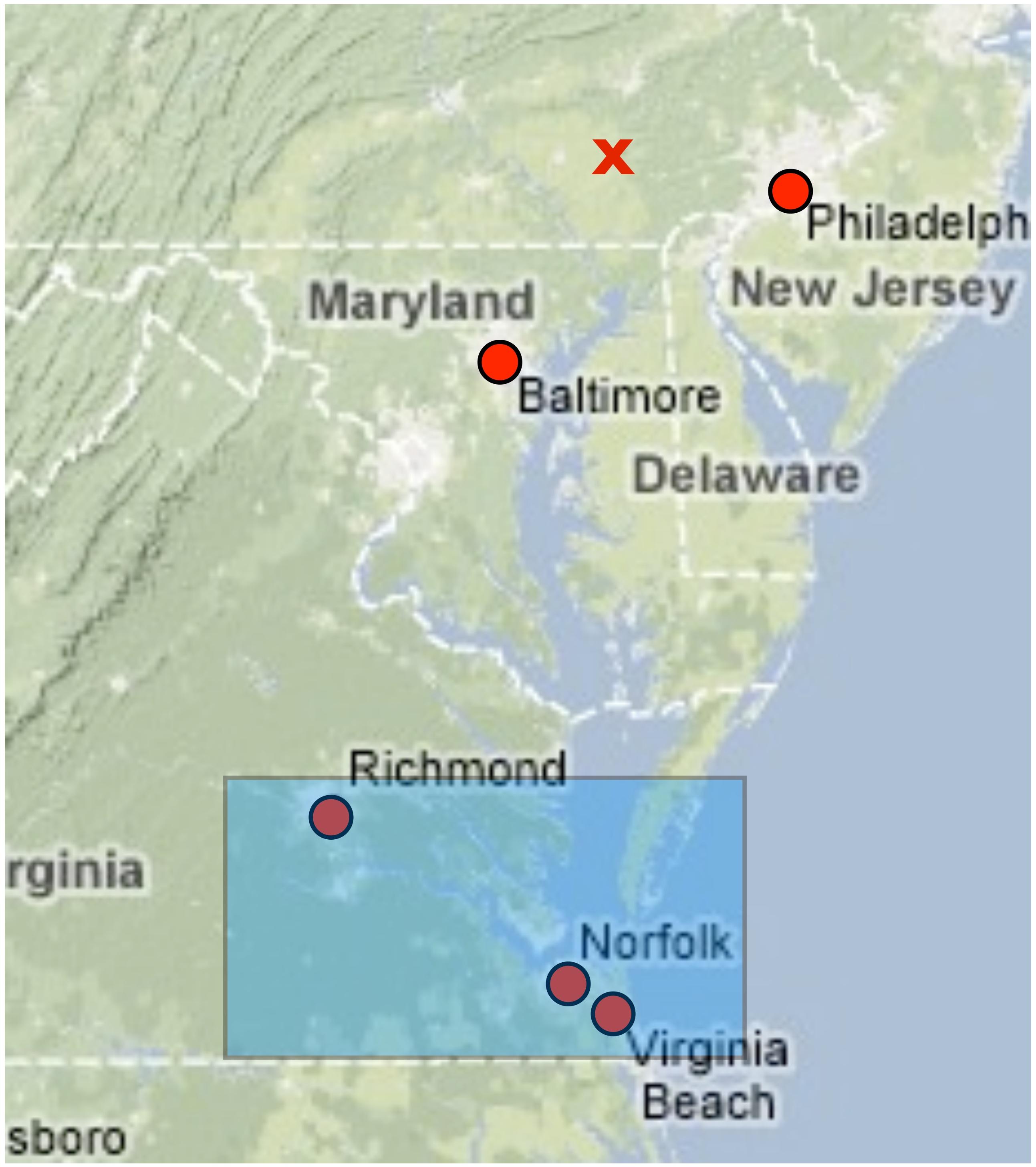
You want to store data about cities (location, elevation, population)...



What kind of operations should your data structure(s) support?

Operations to support the following scenarios...

- Finding addresses on map?
  - Lookup city by name...
- Mobile iPhone user?
  - Find nearest point to me...
- Car GPS system?
  - Calculate shortest-path between cities...
  - Show cities within a given window...
- Political revolution?
  - Insert, delete, rename cities



Data Organizing Principles

- **Ordering**:
  - Put keys into some order so that we know something about where each key is are relative to the other keys.
  - Phone books are easier to search because they are alphabetized.
- **Linking**:
  - Add pointers to each record so that we can find related records quickly.
  - E.g. The index in the back of book provides links from words to the pages on which they appear.
- **Partitioning**:
  - Divide the records into 2 or more groups, each group sharing a particular property.
  - E.g. Multi-volume encyclopedias (Aa-Be, W-Z)
  - E.g. Folders on your hard drive

Ordering

Pheasant,	10
Grouse,	89
Quail,	55
Pelican,	3
Partridge,	32
Duck,	18
Woodpecker,	50
Robin,	89
Cardinal,	102
Eagle,	43
Chicken,	7
Pigeon,	201
Swan,	57
Loon,	213
Turkey,	99
Albatross,	0
Ptarmigan,	22
Finch,	38
Bluejay,	24
Heron,	70
Egret,	88
Goose,	67

Sequential Search – O(n)

Albatross,	0	
Bluejay,	24	
Cardinal,	102	
Chicken,	7	
Duck,	18	$\leftarrow$ (2)
Eagle,	43	
Egret,	88	$\leftarrow$ (3)
Finch,	38	
Goose,	67	$\leftarrow$ (4)
Grouse,	89	
Heron,	70	$\leftarrow$ (1)
Loon,	213	
Partridge,	32	
Pelican,	3	
Pheasant,	10	
Pigeon,	201	
Ptarmigan,	22	
Quail,	55	
Robin,	89	
Swan,	57	
Turkey,	99	
Woodpecker,	50	

Search for "Goose"

Binary Search
O(log n)

Every step discards half the remaining entries:

$$n/2^k = 1$$
$$2^k = n$$
$$k = \log n$$

Big-O notation

- O() notation focuses on the largest term and ignores constants
  - Largest term will dominate eventually for large enough n.
  - Constants depend on "irrelevant" things like machine speed, architecture, etc.

- **Definition:** $T(n)$ is $O(f(n))$ if the limit of $T(n) / f(n)$, is a constant as n goes to infinity.

- **Example**:
  - Suppose $T(n) = 12n^2 + n + 2 \log n$.
  - Consider $f(n) = n^2$
  - Then $\lim [T(n) / f(n)] = \lim [12 + (1/n) + (2 \log n) / n^2] = 12$

- **Example**:
  - Is $T(n) = n \log n$ in $O(n)$?
  - Check: $\lim [(n \log n) / n] = \lim \log n =$ infinity! So no!

Alternative Definition of Big-O

- $T(n)$ is in $O(f(n))$ if there are some constants n_0 and c such that
  - $T(n) < c\, f(n)$ for all $n \geq n_0$

- In other words, once the input size n gets big enough (bigger than n_0), then $T(n)$ is always less than some constant multiple of $f(n)$.

- (c allows us to shift $f(n)$ up by a constant amount to account for machine speed, etc.)

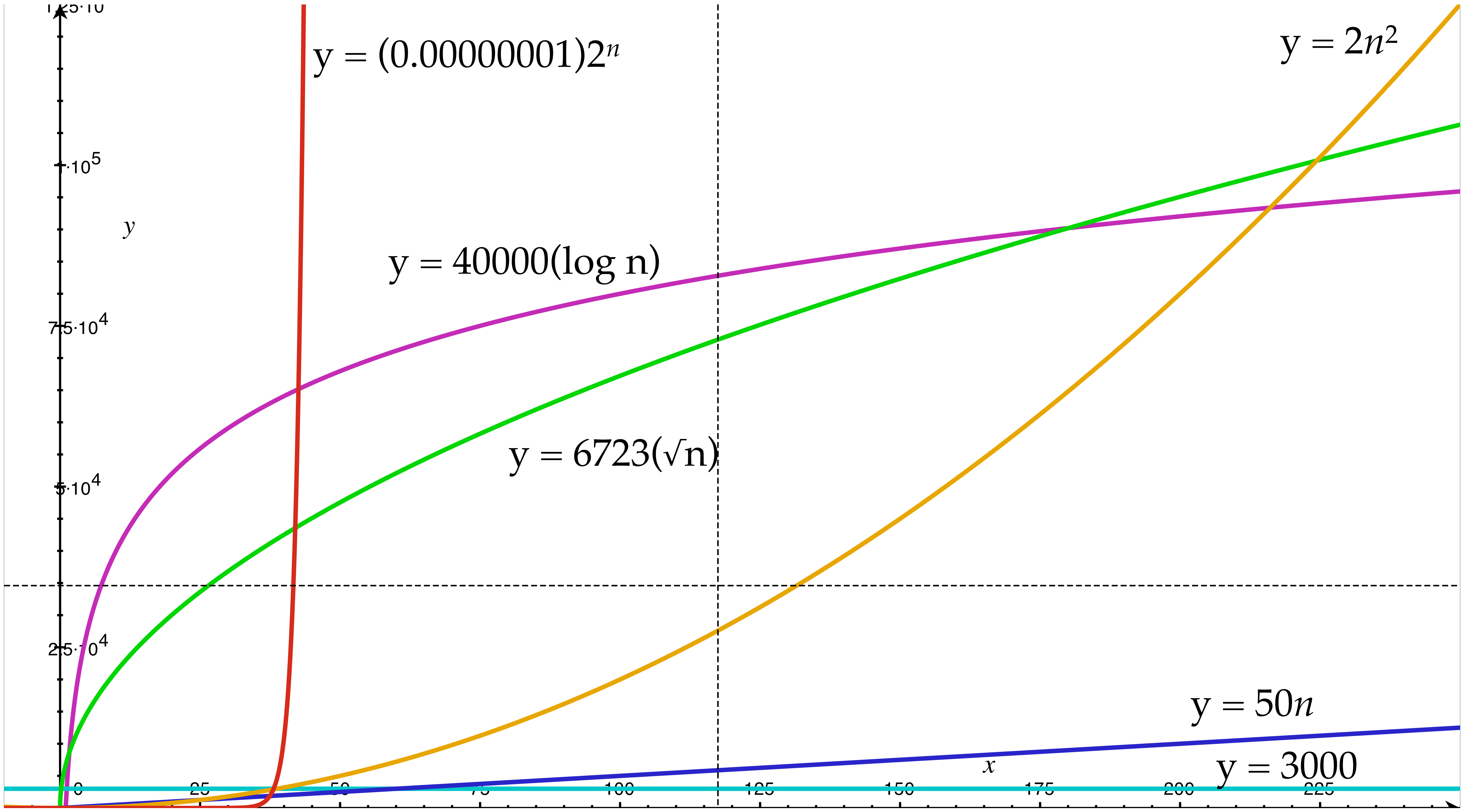
- [Introduced in 1894 by Paul Bachmann, popularized by Don Knuth.]

- Typically, in this class, we'll want things to be *sub-linear:* we don't want to look at every data item.

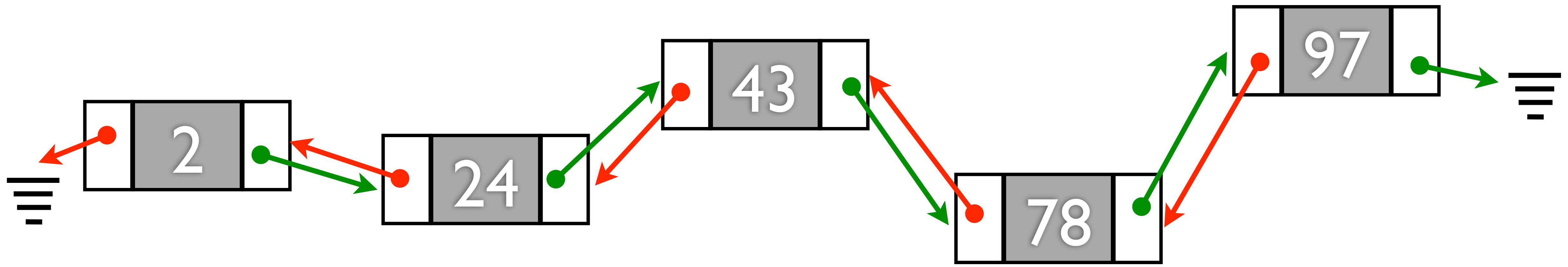
Big-O Taxonomy

Growth Rate	Name	Notes
$O(1)$	constant	Best, independent of input size
$O(\log \log n)$		very fast
$O(\log n)$	logarithmic	often for tree-based data structures
$O(\log^k n)$	polylogarithmic	
$O(n^p)$, $0 < p < 1$	E.g. $O(n1/2) = O(\sqrt{n})$	Still sub-linear
$O(n)$	linear	Have to look at all data
$O(n \log n)$		Time to sort
$O(n^2)$	quadratic	Ok if n is small enough;
$O(n^k)$	polynomial	Tractable
$O(2^n)$, $O(n!)$	exponential, factorial	bad

Big-O Examples



Linking



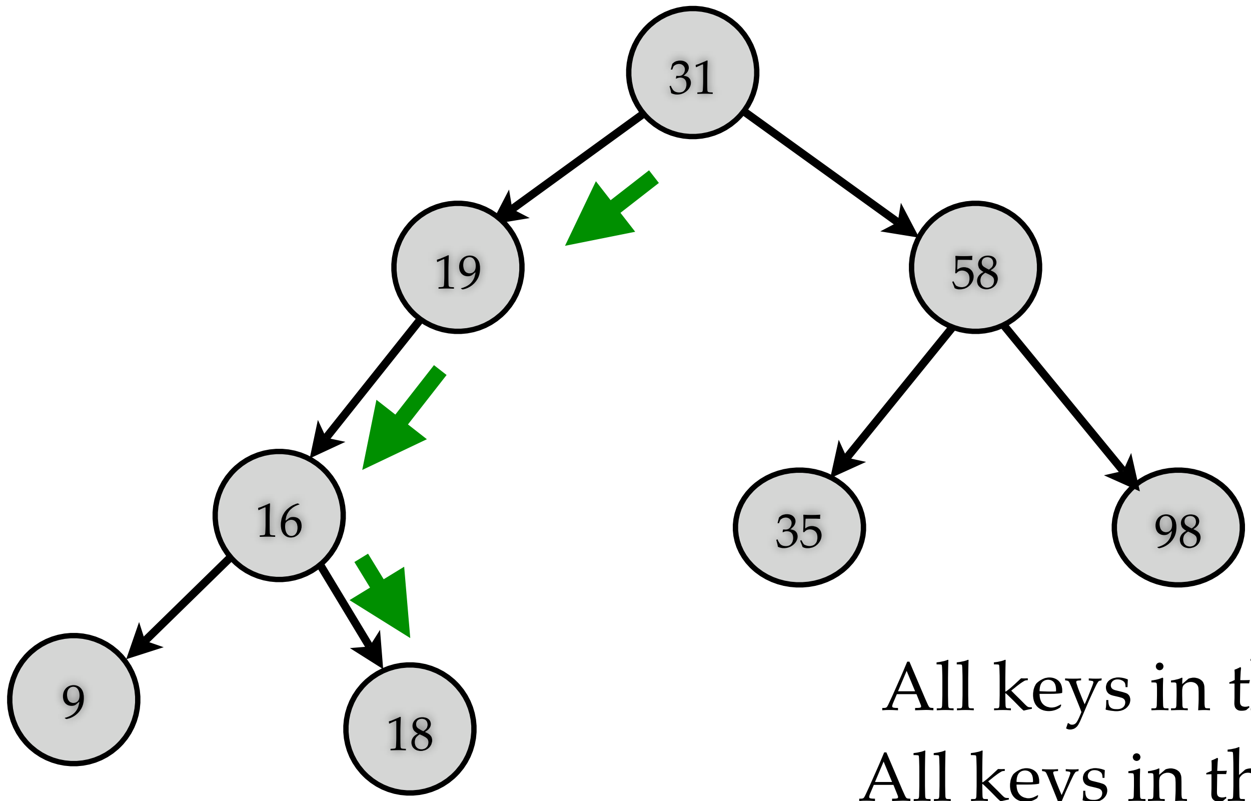
- Records located any where in memory
- Green pointers give "next" element
- Red pointers give "previous" element
- Insertion & deletion easy if you have a pointer to the middle of the list

- Don't have to know size of data at start
- Pointers let us express relationships between pieces of information.

Partitioning

- **Ordering implicitly gives a partitioning based on the “<“ relation.**
- **Partitioning usually combined with linking to point to the two halves.**
- **Prototypical example is the Binary Search Tree:**

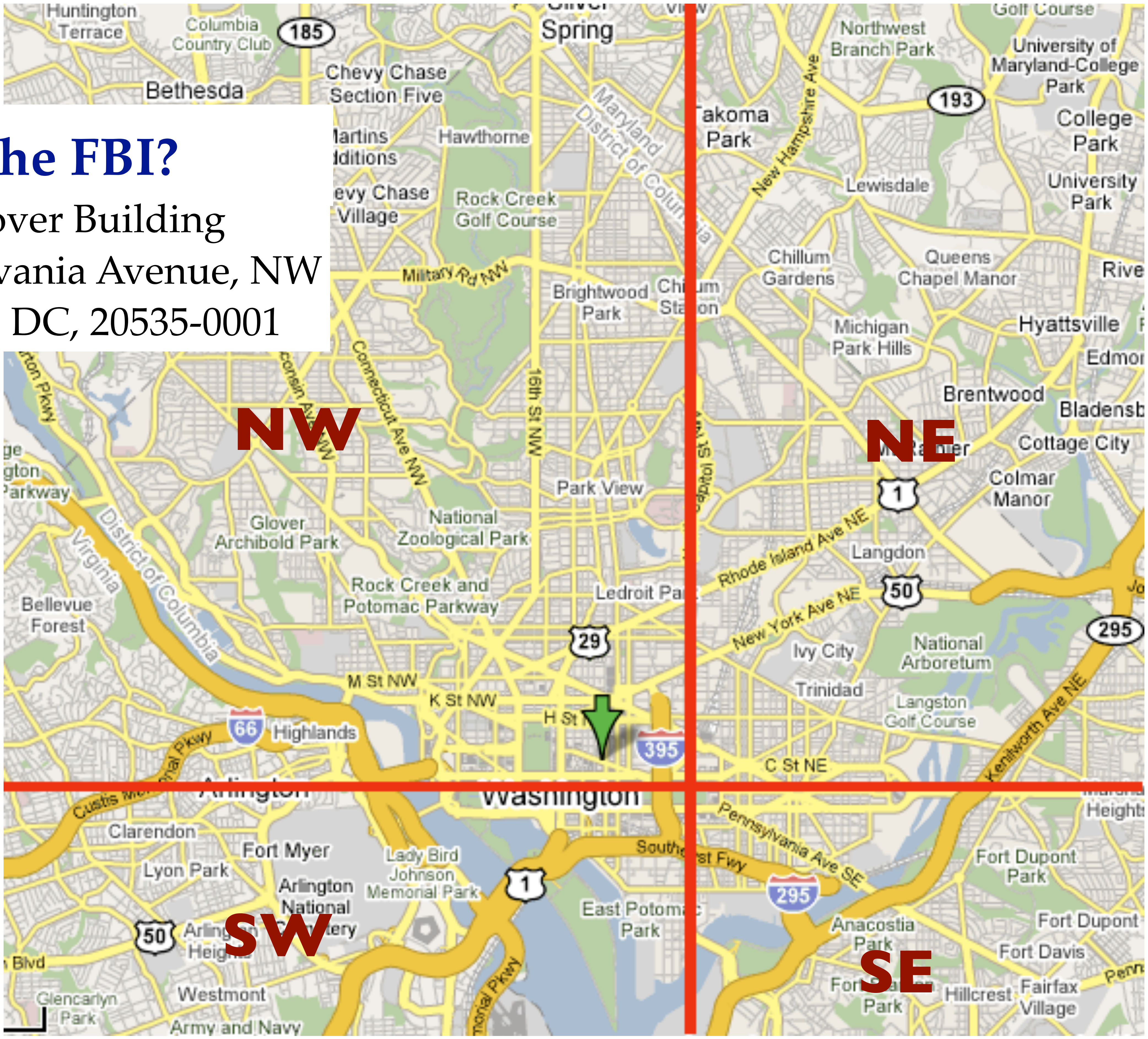
Find 18



All keys in the left subtree are < the root
All keys in the right subtree are ≥ the root

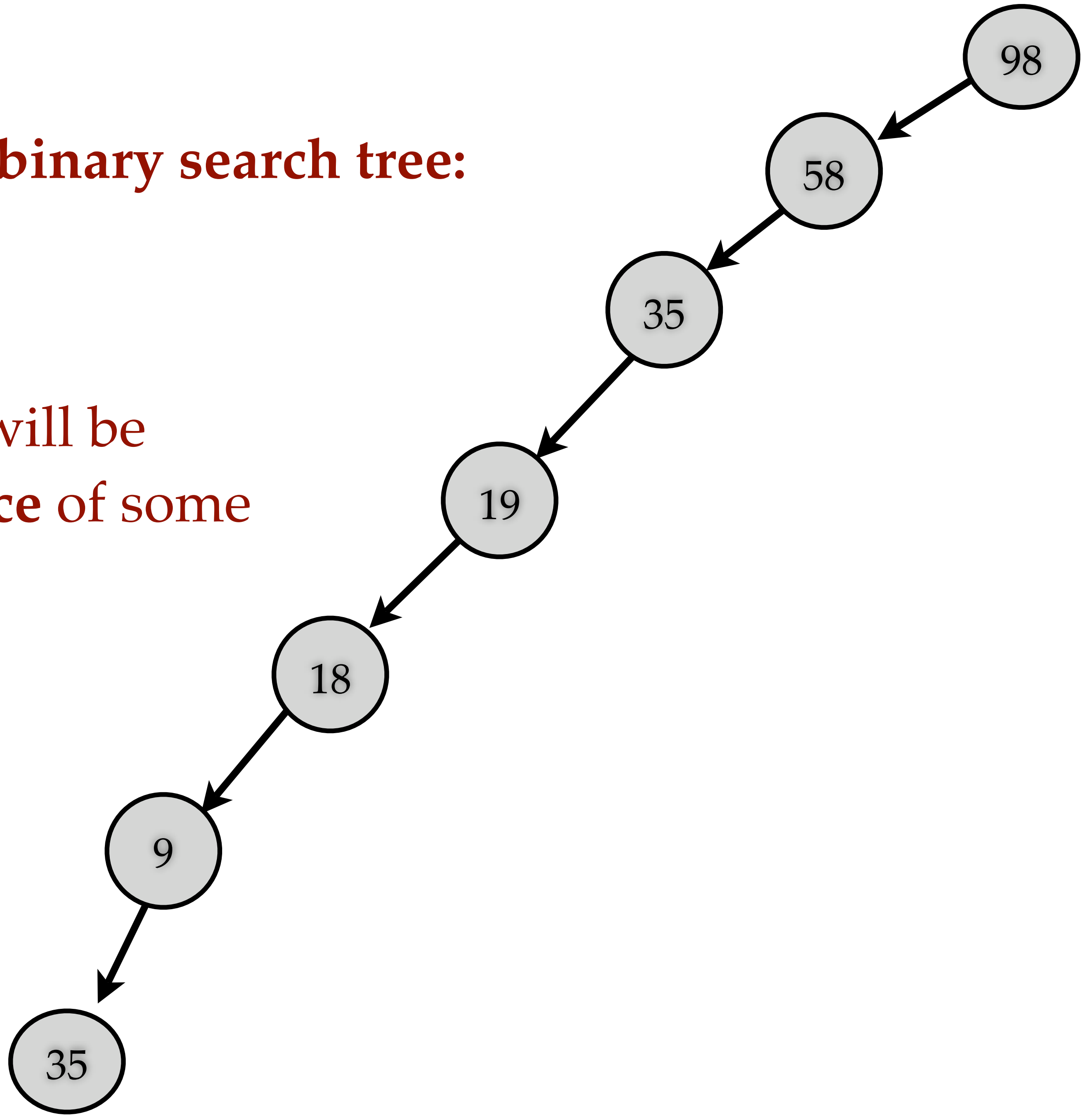
Where’s the FBI?

J. Edgar Hoover Building
935 Pennsylvania Avenue, NW
Washington, DC, 20535-0001



Why is the DC partitioning bad?

- **Everything interesting is in the northwest quadrant.**
- **Want a <u>balanced</u> partition!**
- **Another example: an unbalanced binary search tree: (becomes sequential search)**



- Much of the first part of this class will be techniques for guaranteeing **balance** of some form.
- Binary search guarantees balance by always picking the **median**.
- When using a linked structure, not as easy to find the median.

Implementing Data Structures in C++

```
template<class K> struct Node {
   Node<K> * next;
   K key;
};
```

Structure holds the user data and some data structure bookkeeping info.

```
template<class K> class List {
   public:
      List();
      K front() {
         if(root) return root->key;
         throw EmptyException;
      }
      // ...
   protected:
      Node<K> *root;
};
```

Main data structure class implements the ADT.

Will work for any type K that supports the required operations (like <).

- Remember: for templates to work, you should put all the code into the .h file.
- Templates aren't likely to be required for the coding project, but they're a good mechanism for creating reusable data structures.

Any Data Type Can Be Compared:

- By overloading the < operator, we can define an order on any type (e.g. `MyType`)
- We can sort a vector of `MyTypes` via:

```
struct MyType {
   string name;
   // ...
};

bool operator<(const MyType & A, const MyType & B) {
  return A.name < B.name;
}

vector<MyType> vec;
// ...fill in vec with many MyType records...
sort(vec.begin(), vec.end());
```

- Thus, we can assume we can compare any types.

So,

- Much of programming (and thinking about programming) involves deciding how to arrange information in memory. [Aka data structures.]

- Choice of data structures can make a big speed difference.
  - Sequential search vs. Binary Search means O(n) vs. O(log n).
  - [log (1 billion) < 21].

- **Abstract Data Types** are a way to *encapsulate* and hide the implementation of a data structure, while presenting a clean interface to other programmers.

- Data structuring principles:
  - Ordering
  - Linking
  - (Balanced) partitioning

- Review Big-O notation, if you're fuzzy on it.

Syllabus

- 1/3 **Advanced data structures** (balanced trees, stacks, queues, lists, graphs, etc.)
- 1/3 **Geometric data structures** (Quad-trees, kd-trees, interval trees, range trees,...)
- 1/3 **Miscellaneous data structures** (skip lists, memory management, UNION-FIND)

Details

Basics and Review:

1/29: **Introduction & background.** Goals, administrivia, big-O notation.

1/31, 2/5: **Basic data structures.** Lists, queues, deques, stacks, graphs, trees.

Trees and Balanced Trees:

2/7, 2/12: **Trees.** Definitions, properties, traversals, implementations.

2/14: **Binary search trees.** Dictionary ADT, insertion, deletion, findmin.

2/19: **AVL trees.** Balanced trees, insertion, deletion.

2/21: **Splay trees.** Amortization, splaying.

2/26: **B-trees.** k-ary search trees, insertion, key rotation, deletion, RB-trees.

2/28: **Heaps.** Leftist, skew, binomial, and fibonacci heaps; priority queues.

3/4: **Sorting.** Heap, insertion, shell, radix, and quick sort; ordering space.

3/6: **Minimum spanning tree.** Prim's algorithm.

Resources

- **TA Office Hours:**
  - Radu Dondera, rdondera@cs, Mondays 12:30-2:30pm.
  - Subhojit Basu, subhojit.basu@gmail.com, Wednesdays 2-4pm, AVW 1105.

- **My Office Hours:** Tuesdays 3:30-5pm, AVW 3223.

- **Dave Mount's Lecture Notes:**
  - http://www.cs.umd.edu/~mount/420/Lects/420lects.pdf

- **Books:**
  - *Practical Introduction to Data Structures and Algorithm Analysis*, C++ Edition, 2nd Edition by Clifford A. Shaffer. Prentice Hall, 2000. ISBN: 0-13-028-446-7.
  - *Foundations of Multidimensional and Metric Data Structures* by Hanan Samet. Morgan Kaufmann, 2006. ISBN: 0-12-369-446-9 (recommended).

Assignments & Grading

- **Homeworks:**
  - 5 homeworks, short answers, pseudo-code or English descriptions.
  - Goal: if you really understand the material, each homework shouldn't take more than 2-3 hours.
  - PLEASE be neat with the write ups --- typed solutions make graders happy. In an unrelated note, happy graders probably tend to give higher grades.

- **Programming Project:**
  - Multiple parts (2 or 3).
  - Must be done in C/C++.
  - Homeworks + Project: ~ 50% of final grade.

- **Exams:**
  - Midterm (March 13, in class): ~ 20% of final grade
  - Final, comprehensive (according to university schedule): ~ 30% of final grade

Assignments & Grading

- **Academic Honesty:** All classwork should be done independently, unless explicitly stated otherwise on the assignment handout.
  - You may discuss general solution strategies, but must write up the solutions yourself.
  - If you discuss any problem with anyone else, you must write their name at the top of your assignment, labeling them “collaborators”.

- **NO LATE HOMEWORKS ACCEPTED**
  - Turn in what you have at the time it’s due.
  - All homeworks are due at the start of class.
  - If you will be away, turn in the homework early.

- Late Programming Assignments (projects) will be accepted, but penalized according to the percentages given on the syllabus.