Welcome to CS61B!

- You should be signed up for a lab and discussion section using the SignUpGenius poll, available from the course website. If you can't find a slot, attend any section you can (although you have second priority for seating).
- Labs start today. In (or preferably before) lab this week, get a CS61B Unix account from `https://inst.eecs.berkeley.edu/webacct`.
- Because labs will be crowded, you might want to bring your laptop.
- If you plan to work from home, try logging in remotely to one of the instructional servers.
- We'll be using Piazza for notices, on-line discussions, questions.
- General information about the course is on the home page (grading, lateness, cheating policy, etc.).
- Lectures will be screencast.

Crowding

- At this time, I don't if we will be able to admit any Concurrent Enrollment students. If you choose not to take this course please drop it as soon as possible for the benefit of others (the add/drop deadline is 18 September—6 September if you wish to avoid a fee).

Texts

- There are two readers currently on-line (see the website).
- You could do without printed versions, but might want to print out selected portions for exams (since we don't allow computers in tests).
- Textbook (for first part of the course only) is *Head First Java*. It's kind of silly, but has the necessary material.

Course Organization I

- You read; we illustrate.
- Labs are important: exercise of programming principles as well as practical dirty details go there. Generally we will give you homework points for doing them.
- Homework is important, but really not graded: use it as you see fit and *turn it in!* You get points for just putting some reasonable effort into it.
- Individual projects are *really* important! Expect to learn a lot. Projects are *not* team efforts (that's for later courses).

Course Organization II

- Use of tools *is* part of the course. Programming takes place in a *programming environment:*
  - Handles editing, debugging, compilation, archiving versions.
  - Personally, I keep it simple: Emacs + gjdb + make + git, (documented in one of the readers and on-line). But we'll look at IntelliJ in lab, and Eclipse is OK, too.
- Tests are challenging: better to stay on top than to cram.
- Tests, 40%; Projects, 50%; HW, 10%
- Stressed? Tell us!

Programming, not Java

- Here, we learn *programming,* not Java (or Unix, or Windows, or...)
- Programming principles span many languages
  - Look for connections.
  - Syntax (`x+y` *vs.* `(+ x y)`) is superficial.
  - Java, Python, and Scheme have a lot in common.
- Whether you use GUIs, text interfaces, or embedded systems, important ideas are the same.

For next time

- Please read Chapter 1 of *Head First Java*, plus §1.1–1.9 of the on-line book *A Java Reference*, available on the class website.
- This is an overview of most of Java's features.
- We'll start looking at examples on Friday.
- Always remember the questions that come up when you read something we assign:
  - Who knows? We might have made a mistake.
  - Feel free to ask at the start of lectures, by email, or by Piazza.

Acronyms of Wisdom

DBC

RTFM

A Quick Tour through the First Program

In Python, we would write

```
# Traditional first program
print("Hello, world")
```

But in Java,

```
/** Traditional first program.
 *  @author P. N. Hilfinger */
public class Hello {
    /** Print greeting. ARGS is ignored. */
    public static void main(String[] args) {
         System.out.println("Hello, world!");
    }
}
```

Commentary

```
/** Traditional first program.
 *  @author P. N. Hilfinger */
public class Hello {
    /** Print greeting.  ARGS is ignored.  */
    public static void main(String[] args) {
         System.out.println("Hello, world!");
    }
}
```

- Java comments can either start with '//' and go to the end of the line (like '#' in Python), or they can extend over any number of lines, bracketed by '/*' and '*/'.
- I don't use the '//' comments, except for things that are supposed to be replaced, and our style checks will flag them.
- The second, multiline kind of comment includes those that start with '/**', which are called *documentation comments* or *doc comments*.
- Documentation comments are just comments, having no effect, but various tools interpret them as providing documentation for the things that follow them. They're generally a good idea and our style checks require them.

Classes

```
/** Traditional first program.
 *  @author P. N. Hilfinger */
public class Hello {
    /** Print greeting. ARGS is ignored. */
    public static void main(String[] args) {
         System.out.println("Hello, world!");
    }
}
```

- Every function and variable in Java is contained in some *class*.
- These are like Python's classes, but with (of course) numerous differences in detail.
- All classes, in turn, belong to some *package*. The `Hello` class belongs to the *anonymous package*.
- We'll see named packages later,

Methods (Functions)

```
/** Traditional first program.
 *  @author P. N. Hilfinger */
public class Hello {
    /** Print greeting. ARGS is ignored. */
    public static void main(String[] args) {
         System.out.println("Hello, world!");
    }
}
```

- Function headers in Java contain more information than those in Python. They specify the *types* of values *returned* by the function and taken as *parameters* to the functions.
- The "type" `void` has no possible values; the *main* function here returns nothing. The type `String` is like Python's `str`. The trailing '`[]`' means *array of*. Arrays are like Python lists, except that their size is fixed once created.
- Hence, *main* takes a list of strings and returns nothing.
- Functions named "main" and defined like the example about are special: they are what get called when one runs a Java program (in Python, the main function is essentially anonymous).

Selection

```
/** Traditional first program.
 *  @author P. N. Hilfinger */
public class Hello {
    /** Print greeting. ARGS is ignored. */
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

- As in Python, $\mathcal{E}.N$ means "the thing named N that is in or that applies to the thing identified (or computed) by $\mathcal{E}$."
- Thus "`System.out`" means "the variable named '`out`' that is found in the class named '`System`'."
- Likewise, "`System.out.println`" means "the method named '`println`' that applies to the object referenced by the value of variable '`System.out`'."

Access

```
/** Traditional first program.
 *  @author P. N. Hilfinger */
public class Hello {
    /** Print greeting. ARGS is ignored. */
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

- Every declared entity in Java has *access permissions* indicating what pieces of code may mention it.
- In particular, *public* classes, methods, and variables may be referred to anywhere else in the program.
- We sometimes refer to them as *exported* from their class (for methods or varialbles) or package (for classes).

Access

```
/** Traditional first program.
 *  @author P. N. Hilfinger */
public class Hello {
    /** Print greeting. ARGS is ignored. */
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

- Static methods and variables are "one-of" things.
- A static method is just like an ordinary Python function (outside of any class) or a function in a Python class that is annotated `@staticmethod`.
- A static variable is like a Python variable defined outside of any class or a variable selected from a class, as opposed to from a class instance.
- Other variables are local variables (in functions) or instance variables (in classes), and these are as in Python.

Administrivia

- Please make sure you have obtained a Unix account. If you have very recently (i.e., since today) signed up for concurrent enrollment please email us your name, email, and SID. After we have a chance to process it, you will be able to use WebAcct, as Lab #1 specifies.
- Lab #1 is due Wednesday (end of Wednesday at midnight). Usually, labs are due Friday midnight of the week they occur. It is especially important to set up your central reppository.
- If you decide not to take this course after all, please tell CalCentral ASAP, so that we can adjust the waiting list accordingly.
- HW #0 now up; due next Friday at midnight. You get credit for any submission, but we suggest you give the problems a serious try.

Lecture #2: Let's Write a Program: Prime Numbers

Problem: want `java Primes` U to print prime numbers through U.

You type: `java Primes 101`

It types:
```
2 3 5 7 11 13 17 19 23 29
31 37 41 43 47 53 59 61 67 71
73 79 83 89 97 101
```

Definition: A *prime* number is an integer greater than 1 that has no divisors smaller than itself other than 1.
(Alternatively: $p > 1$ is prime iff gcd$(p, x) = 1$ for all $0 < x < p$.)

Useful Facts:

- $k \leq \sqrt{N}$ iff $N/k \geq \sqrt{N}$, for $N, k > 0$.
- If k divides N then N/k divides N.

So: Try all potential divisors up to and including the square root.

Plan

```
public class Primes {
  /** Print all primes up to ARGS[0] (interpreted as an
   *  integer), 10 to a line. */
  public static void main(String[] args) {
    printPrimes(Integer.parseInt(args[0]));
  }

  /** Print all primes up to and including LIMIT, 10 to
   *  a line. */
  private static void printPrimes(int limit) {
    /*{ For every integer, x,  between 2 and LIMIT, print it if
        isPrime(x), 10 to a line. }*/
  }

  /** True iff X is prime */
  private static boolean isPrime(int x) {
    return /*( X is prime )*/;
  }
}
```

Testing for Primes

```
private static boolean isPrime(int x) {
  if (x <= 1)
    return false;
  else
    return !isDivisible(x, 2);  // "!" means "not"
}

/** True iff X is divisible by any positive number >=K and < X,
 *  given K > 1. */
private static boolean isDivisible(int x, int k) {
  if (k >= x)             // a "guard"
    return false;
  else if (x % k == 0)  // "%" means "remainder"
    return true;
  else // if (k < x && x % k != 0)
    return isDivisible(x, k+1);
}
```

Thinking Recursively

Understand and check `isDivisible(13,2)` by *tracing one level.*

```
/** True iff X is divisible by
 *  some number >=K and < X,
 *  given K > 1. */
private static boolean isDivisible...
  if (k >= x)
    return false;
  else if (x % k == 0)
    return true;
  else
    return isDivisible(x, k+1);
}
```

Lesson: Comments aid understanding. Make them *count*!

- Call assigns `x=13, k=2`
- Body has form '`if (k >= x)` S_1 `else` S_2'.
- Since $2 < 13$, we evaluate the first `else`.
- Check if $13 \bmod 2 = 0$; it's not.
- Left with `isDivisible(13,3)`.
- Rather than tracing it, instead use the *comment:*
- Since 13 is *not* divisible by any integer in the range 3..12 (and $3 > 1$), `isDivisible(13,3)` must be *false*, and we're done!
- Sounds like that last step begs the question. Why doesn't it?

Iteration

- `isDivisible` is *tail recursive*, and so creates an *iterative process*.
- Traditional "Algol family" production languages have special syntax for iteration. Four equivalent versions of `isDivisible`:

```
if (k >= x)
  return false;
else if (x % k == 0)
  return true;
else
  return isDivisible(x, k+1);
```

```
while (k < x) { // !(k >= x)
  if (x % k == 0)
    return true;
  k = k+1;
  // or k += 1, or (yuch) k++
}
return false;
```

```
int k1 = k;
while (k1 < x) {
  if (x % k1 == 0)
    return true;
  k1 += 1;
}
return false;
```

```
for (int k1 = k; k1 < x; k1 += 1) {
  if (x % k1 == 0)
    return true;
}
return false;
```

Using Facts about Primes

- We haven't used the Useful Facts from an earlier slide. Only have to check for divisors up to the square root.
- So, reimplement the iterative version of `isDivisible`:

```
/** True iff X is divisible by some number >=K and < X,
 *  given that K > 1, and that X is not divisible by
 *  any number >1 and <K. */
private static boolean isDivisible(int x, int k) {
  int limit = (int) Math.round(Math.sqrt(x));
  for (int k1 = k; k1 <= limit; k1 += 1) {
    if (x % k1 == 0)
      return true;
  }
  return false;
}
```

- Why the additional (blue) condition in the comment?

Cautionary Aside: Floating Point

- In the last slide, we had

  ```
  int limit = (int) Math.round(Math.sqrt(x));
  for (int k1 = k; k1 <= limit; k1 += 1) {
      ...
  ```

  intending that this would check all values of k1 up to and including the square root of x.

- Since floating-point operations yield *approximations* to the corresponding mathematical operations, you might ask the following about `(int) Math.round(Math.sqrt(x))`:

  - Is it always at least $\lfloor\sqrt{x}\rfloor$, where $\lfloor z\rfloor$ is the largest integer $\leq z$? (If not, we might miss testing $\sqrt{x}$ when x is a perfect square.)

- As it happens, the answer is "yes" for IEEE floating-point square roots.

- Just an example of the sort of detail that must be checked in edge cases.

Final Task: printPrimes (Simplified)

```
/** Print all primes up to and including LIMIT. */
private static void printPrimes(int limit) {






}
```

Simplified printPrimes Solution

```
/** Print all primes up to and including LIMIT. */
private static void printPrimes(int limit) {
    for (int p = 2; p <= limit; p += 1) {
        if (isPrime(p)) {
            System.out.print(p + " ");
        }
    }
    System.out.println();
}
```

printPrimes (full version)

```
/** Print all primes up to and including LIMIT, 10 to
 *  a line. */
private static void printPrimes(int limit) {
    int np;
    np = 0;
    for (int p = 2; p <= limit; p += 1) {
        if (isPrime(p)) {
            System.out.print(p + " ");
            np += 1;
            if (np % 10 == 0)
                System.out.println();
        }
    }
    if (np % 10 != 0)
        System.out.println();
}
```

Recreation

Prove that $\lfloor (2+\sqrt{3})^n \rfloor$ is odd for all integer $n \geq 0$.

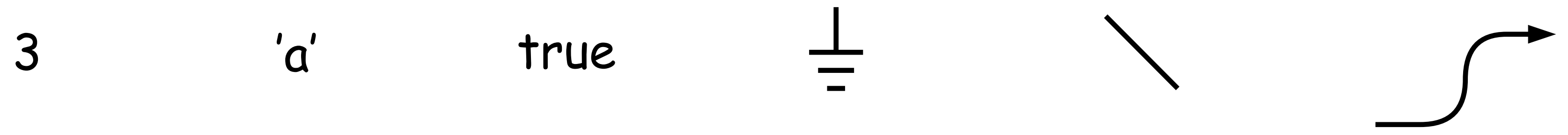
[Source: D. O. Shklarsky, N. N. Chentzov, I. M. Yaglom, *The USSR Olympiad Problem Book*, Dover ed. (1993), from the W. H. Freeman edition, 1962.]

CS61B Lecture #3: Values and Containers

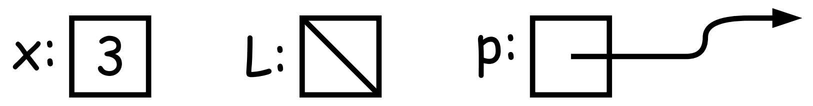
- Labs are normally due at midnight Friday. Last week's is due tonight.
- **Today.** Simple classes. Scheme-like lists. Destructive vs. non-destructive operations. Models of memory.

Values and Containers

- *Values* are numbers, booleans, and pointers. Values never change.



- *Simple containers* contain values:

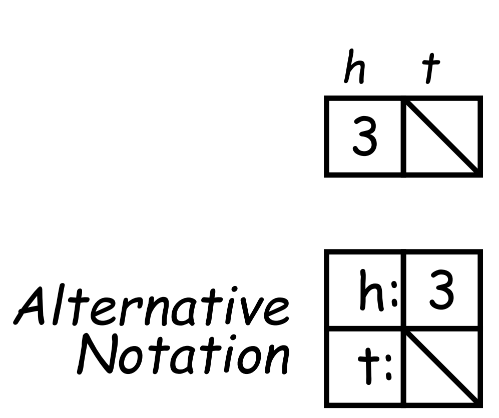


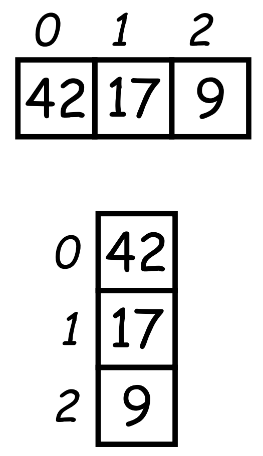
Examples: variables, fields, individual array elements, parameters.

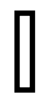
Structured Containers

Structured containers contain (0 or more) other containers:

Class Object *Array Object* *Empty Object*

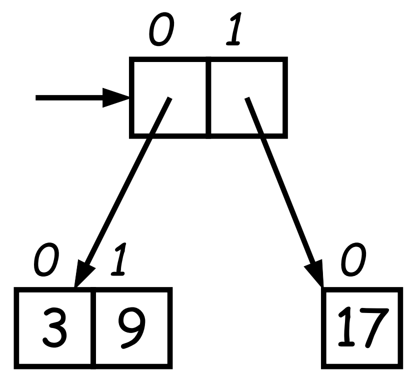






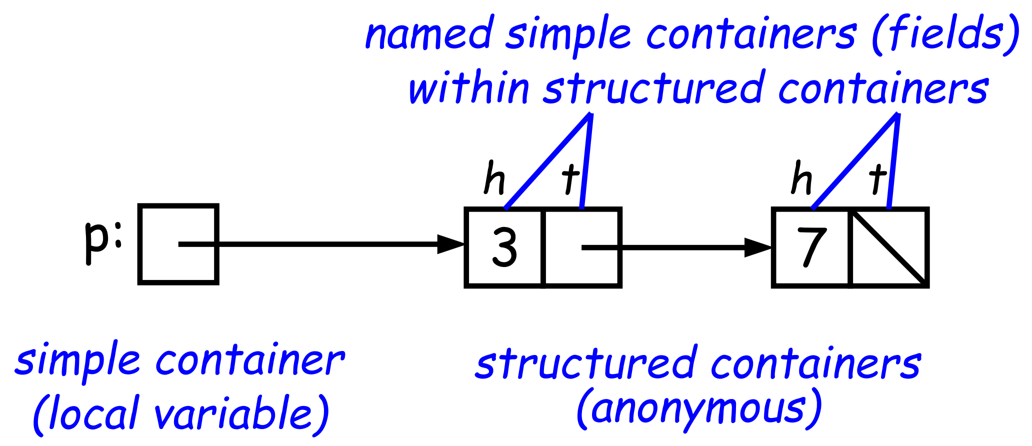
Pointers

- *Pointers* (or *references*) are values that *reference* (point to) containers.
- One particular pointer, called **null**, points to nothing.
- In Java, structured containers contain only simple containers, but pointers allow us to build arbitrarily big or complex structures anyway.



Containers in Java

- Containers may be *named* or *anonymous*.
- In Java, *all* simple containers are named, *all* structured containers are anonymous, and pointers point only to structured containers. (Therefore, structured containers contain only simple containers).



- In Java, assignment copies values into simple containers.
- *Exactly* like Scheme and Python!
- (Python also has slice assignment, as in `x[3:7]=...`, which is shorthand for something else entirely.)

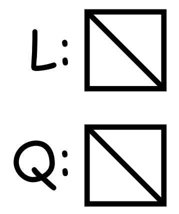
Defining New Types of Object

- Class declarations introduce new types of objects.
- Example: list of integers:

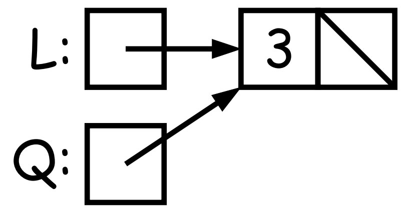
```
public class IntList {
  // Constructor function (used to initialize new object)
  /** List cell containing (HEAD, TAIL). */
  public IntList(int head, IntList tail) {
    this.head = head; this.tail = tail;
  }

  // Names of simple containers (fields)
  // WARNING: public instance variables usually bad style!
  public int head;
  public IntList tail;
}
```

Primitive Operations

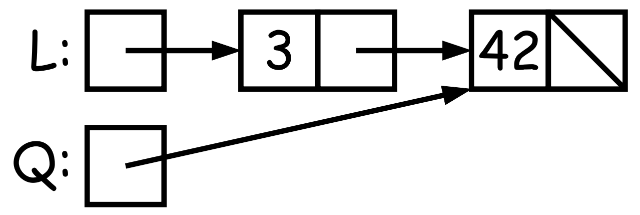
```
IntList Q, L;
```



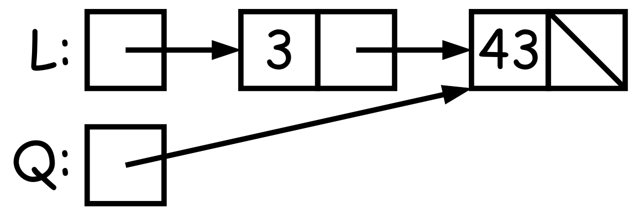
```
L = new IntList(3, null);
Q = L;
```



```
Q = new IntList(42, null);
L.tail = Q;
```

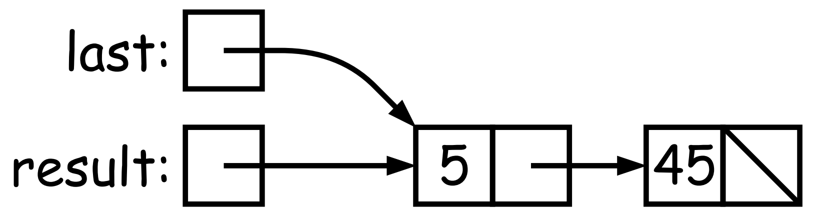


```
L.tail.head += 1;
// Now Q.head == 43
// and L.tail.head == 43
```

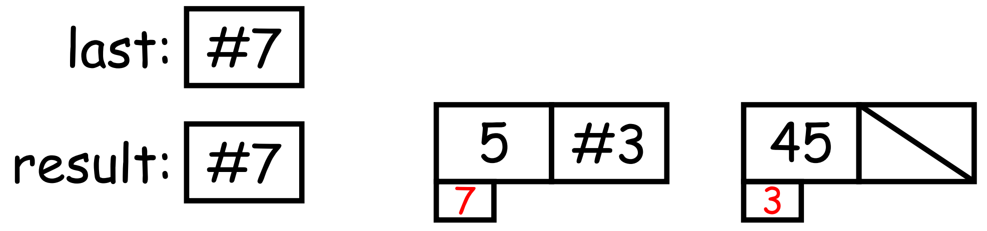


Side Excursion: Another Way to View Pointers

- Some folks find the idea of "copying an arrow" somewhat odd.
- Alternative view: think of a pointer as a *label*, like a street address.
- Each object has a permanent label on it, like the address plaque on a house.
- Then a variable containing a pointer is like a scrap of paper with a street address written on it.
- One view:

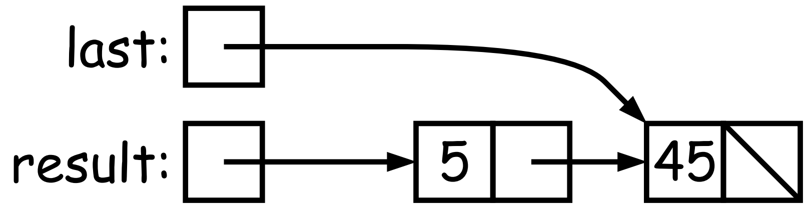


- Alternative view:

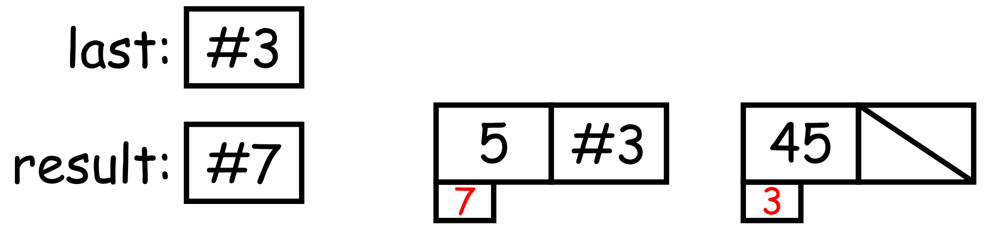


Another Way to View Pointers (II)

- *Assigning a pointer to a variable looks just like assigning an integer to a variable.*
- So, after executing “last = last.tail;” we have



- Alternative view:



- Under alternative view, you might be less inclined to think that assignment would change object #7 itself, rather than just “last”.
- BEWARE! Internally, pointers really are just numbers, but Java treats them as more than that: they have *types,* and you can’t just change integers into pointers.

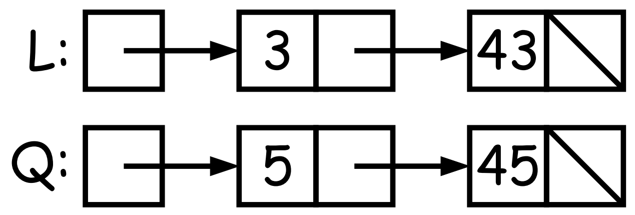
Destructive vs. Non-destructive

Problem: Given a (pointer to a) list of integers, L, and an integer increment n, return a list created by incrementing all elements of the list by n.

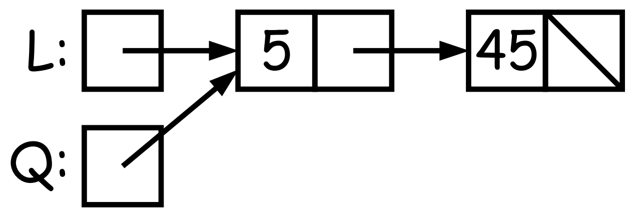
```
/** List of all items in P incremented by n. Does not modify
 *  existing IntLists. */
static IntList incrList(IntList P, int n) {
    return /*( P, with each element incremented by n )*/
}
```

We say `incrList` is *non-destructive,* because it leaves the input objects unchanged, as shown on the left. A *destructive* method may modify the input objects, so that the original data is no longer available, as shown on the right:

After `Q = incrList(L, 2)`:



After `Q = dincrList(L, 2)` (destructive):



Nondestructive IncrList: Recursive

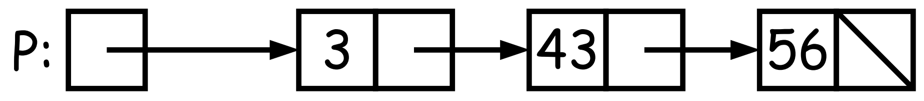
```
/** List of all items in P incremented by n. */
static IntList incrList(IntList P, int n) {
  if (P == null)
    return null;
  else return new IntList(P.head+n, incrList(P.tail, n));
}
```

- Why does `incrList` have to return its result, rather than just setting `P`?
- In the call `incrList(P, 2)`, where `P` contains 3 and 43, which `IntList` object gets created first?

An Iterative Version

An iterative `incrList` is tricky, because it is *not* tail recursive.
Easier to build things first-to-last, unlike recursive version:

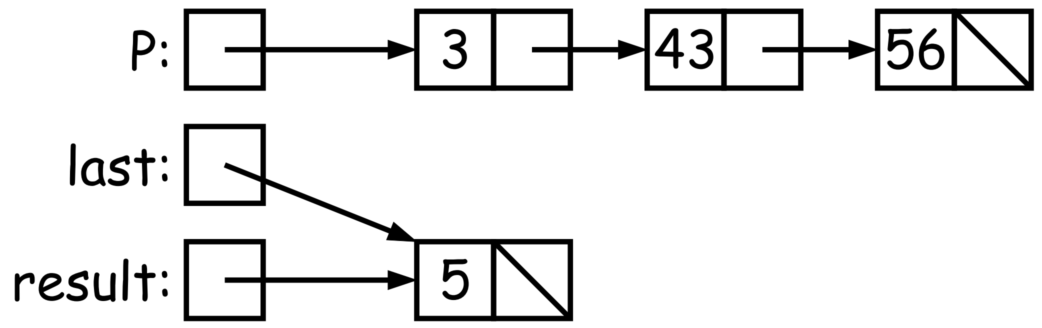
```
static IntList incrList(IntList P, int n) {
  if (P == null)       <<<
    return null;
  IntList result, last;
  result = last
     = new IntList(P.head+n, null);
  while (P.tail != null) {
    P = P.tail;
    last.tail
      = new IntList(P.head+n, null);
    last = last.tail;
  }
  return result;
}
```



An Iterative Version

An iterative `incrList` is tricky, because it is *not* tail recursive.
Easier to build things first-to-last, unlike recursive version:

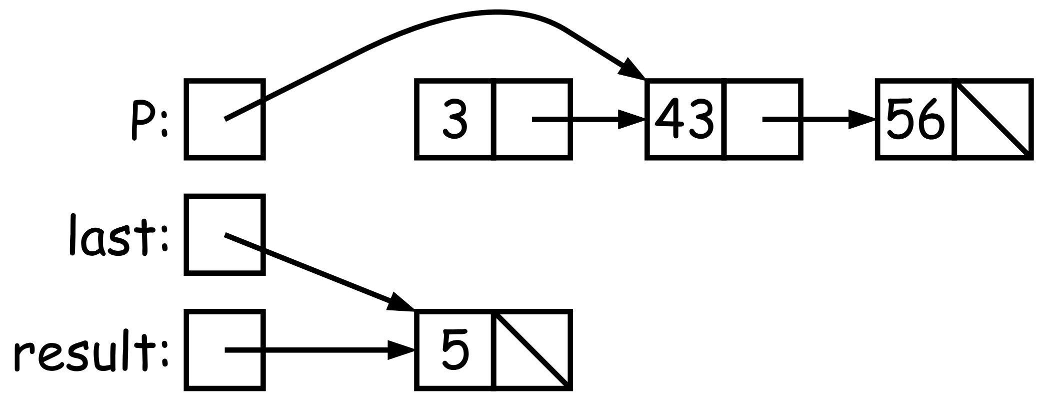
```
static IntList incrList(IntList P, int n) {
  if (P == null)
    return null;
  IntList result, last;
  result = last        <<<
     = new IntList(P.head+n, null);
  while (P.tail != null) {
    P = P.tail;
    last.tail
      = new IntList(P.head+n, null);
    last = last.tail;
  }
  return result;
}
```



An Iterative Version

An iterative `incrList` is tricky, because it is *not* tail recursive.
Easier to build things first-to-last, unlike recursive version:

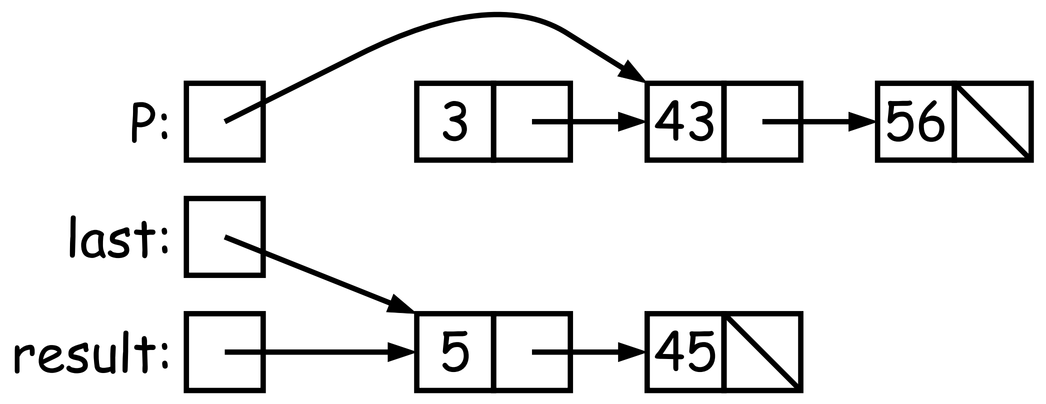
```
static IntList incrList(IntList P, int n) {
  if (P == null)
    return null;
  IntList result, last;
  result = last
     = new IntList(P.head+n, null);
  while (P.tail != null) {
    P = P.tail;        <<<
    last.tail
      = new IntList(P.head+n, null);
    last = last.tail;
  }
  return result;
}
```



An Iterative Version

An iterative `incrList` is tricky, because it is *not* tail recursive.
Easier to build things first-to-last, unlike recursive version:

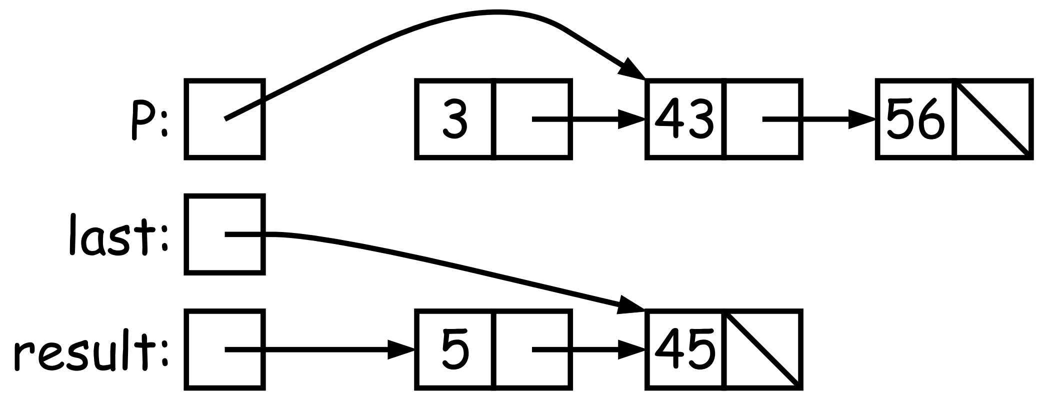
```
static IntList incrList(IntList P, int n) {
  if (P == null)
    return null;
  IntList result, last;
  result = last
     = new IntList(P.head+n, null);
  while (P.tail != null) {
    P = P.tail;
    last.tail          <<<
      = new IntList(P.head+n, null);
    last = last.tail;
  }
  return result;
}
```



An Iterative Version

An iterative `incrList` is tricky, because it is *not* tail recursive.
Easier to build things first-to-last, unlike recursive version:

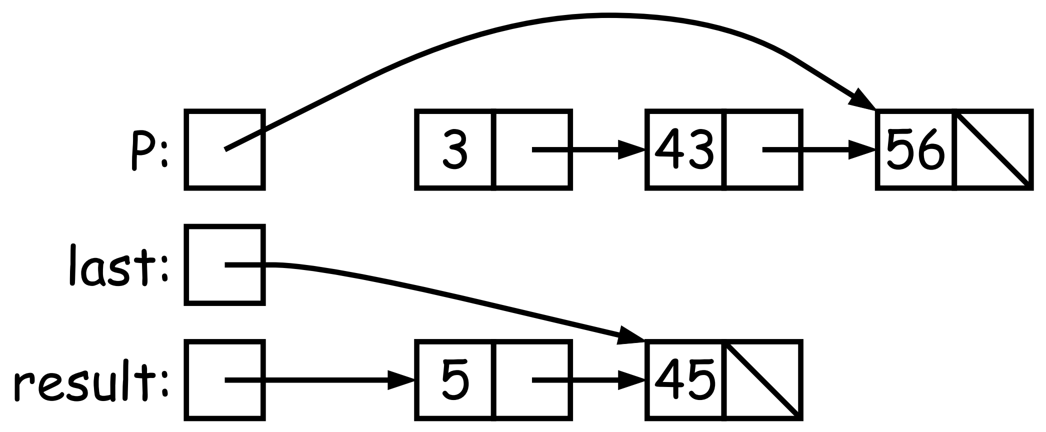
```
static IntList incrList(IntList P, int n) {
  if (P == null)
    return null;
  IntList result, last;
  result = last
     = new IntList(P.head+n, null);
  while (P.tail != null) {
    P = P.tail;
    last.tail
      = new IntList(P.head+n, null);
    last = last.tail; <<<
  }
  return result;
}
```



An Iterative Version

An iterative `incrList` is tricky, because it is *not* tail recursive.
Easier to build things first-to-last, unlike recursive version:

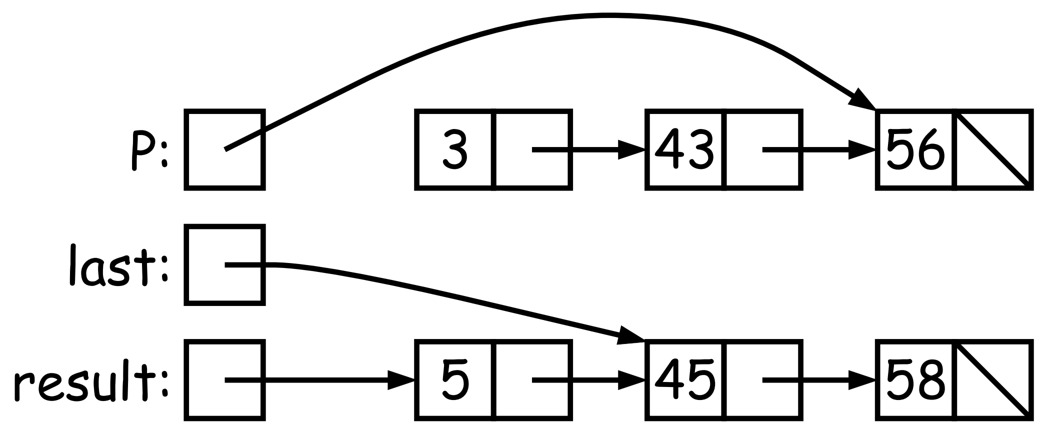
```
static IntList incrList(IntList P, int n) {
  if (P == null)
    return null;
  IntList result, last;
  result = last
     = new IntList(P.head+n, null);
  while (P.tail != null) {
    P = P.tail;        <<<
    last.tail
      = new IntList(P.head+n, null);
    last = last.tail;
  }
  return result;
}
```



An Iterative Version

An iterative `incrList` is tricky, because it is *not* tail recursive.
Easier to build things first-to-last, unlike recursive version:

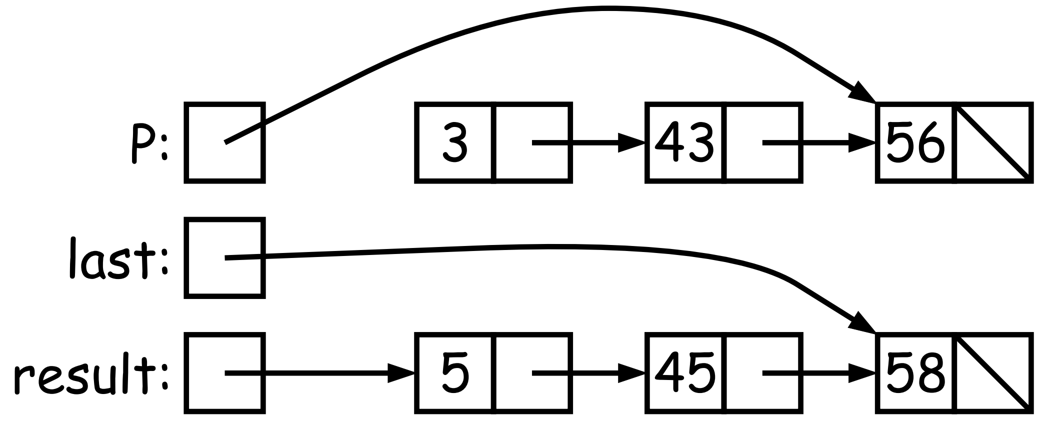
```
static IntList incrList(IntList P, int n) {
  if (P == null)
    return null;
  IntList result, last;
  result = last
     = new IntList(P.head+n, null);
  while (P.tail != null) {
    P = P.tail;
    last.tail          <<<
      = new IntList(P.head+n, null);
    last = last.tail;
  }
  return result;
}
```



An Iterative Version

An iterative `incrList` is tricky, because it is *not* tail recursive.
Easier to build things first-to-last, unlike recursive version:

```
static IntList incrList(IntList P, int n) {
  if (P == null)
    return null;
  IntList result, last;
  result = last
     = new IntList(P.head+n, null);
  while (P.tail != null) {
    P = P.tail;
    last.tail
      = new IntList(P.head+n, null);
    last = last.tail; <<<
  }
  return result;
}
```



CS61B Lecture #4: Simple Pointer Manipulation

Recreation Prove that for every acute angle $\alpha > 0$,

$$\tan \alpha + \cot \alpha \geq 2$$

Announcements

- **Today:** More pointer hacking.
- **Handing in labs and homework:** We'll be lenient about accepting late homework and labs for lab1, lab2, and hw0. Just get it done: part of the point is getting to understand the tools involved. We will *not* accept submissions by email.
- We will feel free to interpret the absence of a central repository for you or a lack of a lab1 submission from you as indicating that you intend to drop the course.
- Project 0 to be released tonight.
- HW1 is released.

Small Test of Understanding

- In Java, the keyword **final** in a variable declaration means that the variable's value may not be changed after the variable is initialized.
- Is the following class valid?

```
public class Issue {

    private final IntList aList = new IntList(0, null);

    public void modify(int k) {
        this.aList.head = k;
    }
}
```

Why or why not?

Small Test of Understanding

- In Java, the keyword **final** in a variable declaration means that the variable's value may not be changed after the variable is initialized.
- Is the following class valid?

```
public class Issue {

    private final IntList aList = new IntList(0, null);

    public void modify(int k) {
        this.aList.head = k;
    }
}
```

Why or why not?

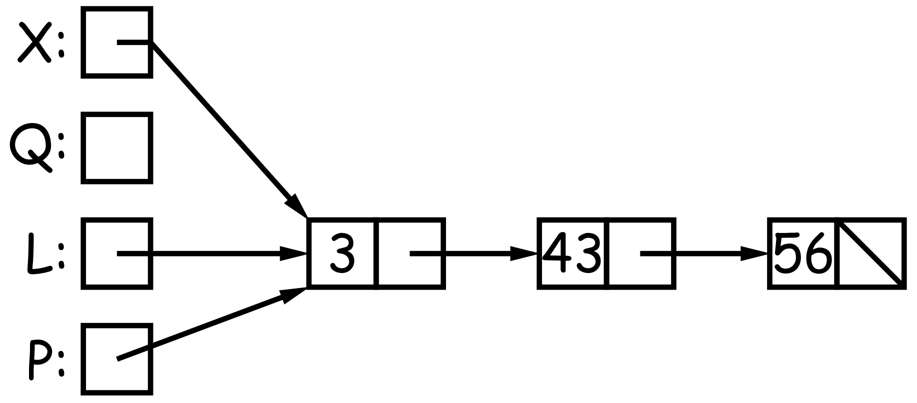
Answer: This is *valid*. Although `modify` changes the `head` variable of the object pointed to by `aList`, it does *not* modify the contents of `aList` itself (which is a pointer).

Destructive Incrementing

Destructive solutions may modify objects in the original list to save time or space:

```
/** Destructively add N to L's items. */
static IntList dincrList(IntList P, int n) {
  if (P == null)
    return null;
  else {
    P.head += n;
    P.tail = dincrList(P.tail, n);
    return P;
  }
}
/** Destructively add N to L's items. */
static IntList dincrList(IntList L, int n)
{
  // 'for' can do more than count!
  for (IntList p = L; p != null; p = p.tail)
    p.head += n;
  return L;
}
```

```
X = IntList.list(3, 43, 56);
/* IntList.list from HW #1 */
Q = dincrList(X, 2);
```

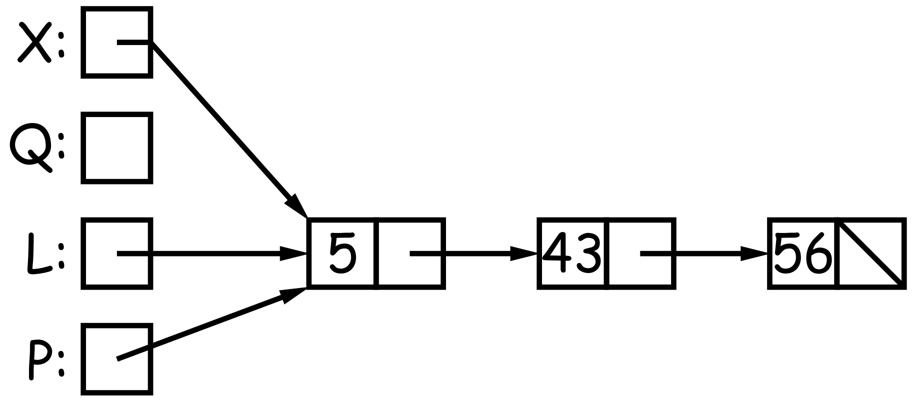


Destructive Incrementing

Destructive solutions may modify objects in the original list to save time or space:

```
/** Destructively add N to L's items. */
static IntList dincrList(IntList P, int n) {
  if (P == null)
    return null;
  else {
    P.head += n;
    P.tail = dincrList(P.tail, n);
    return P;
  }
}
/** Destructively add N to L's items. */
static IntList dincrList(IntList L, int n)
{
  // 'for' can do more than count!
  for (IntList p = L; p != null; p = p.tail)
    p.head += n;
  return L;
}
```

```
X = IntList.list(3, 43, 56);
/* IntList.list from HW #1 */
Q = dincrList(X, 2);
```

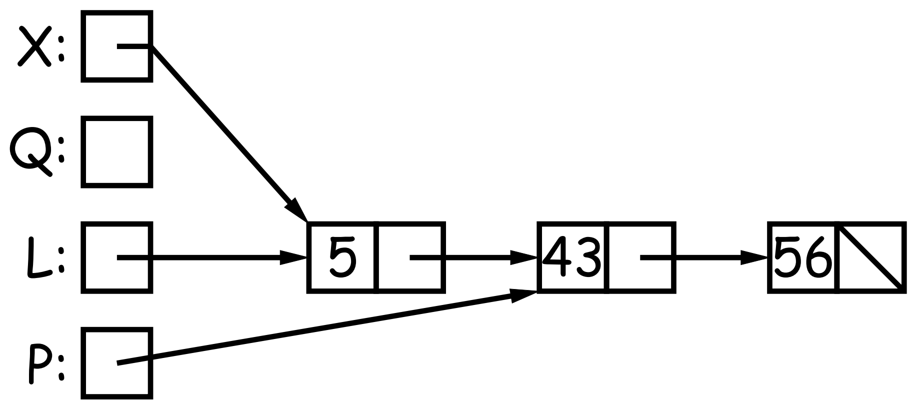


Destructive Incrementing

Destructive solutions may modify objects in the original list to save time or space:

```
/** Destructively add N to L's items. */
static IntList dincrList(IntList P, int n) {
  if (P == null)
    return null;
  else {
    P.head += n;
    P.tail = dincrList(P.tail, n);
    return P;
  }
}
/** Destructively add N to L's items. */
static IntList dincrList(IntList L, int n)
{
  // 'for' can do more than count!
  for (IntList p = L; p != null; p = p.tail)
    p.head += n;
  return L;
}
```

```
X = IntList.list(3, 43, 56);
/* IntList.list from HW #1 */
Q = dincrList(X, 2);
```

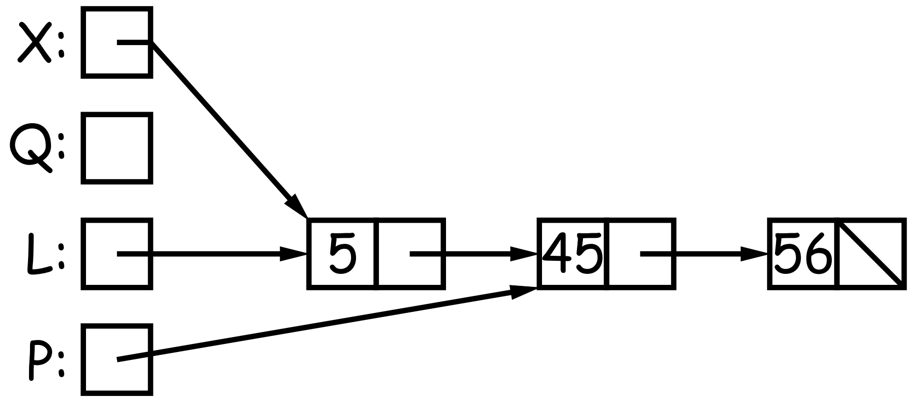


Destructive Incrementing

Destructive solutions may modify objects in the original list to save time or space:

```
/** Destructively add N to L's items. */
static IntList dincrList(IntList P, int n) {
  if (P == null)
    return null;
  else {
    P.head += n;
    P.tail = dincrList(P.tail, n);
    return P;
  }
}
/** Destructively add N to L's items. */
static IntList dincrList(IntList L, int n)
{
  // 'for' can do more than count!
  for (IntList p = L; p != null; p = p.tail)
    p.head += n;
  return L;
}
```

```
X = IntList.list(3, 43, 56);
/* IntList.list from HW #1 */
Q = dincrList(X, 2);
```

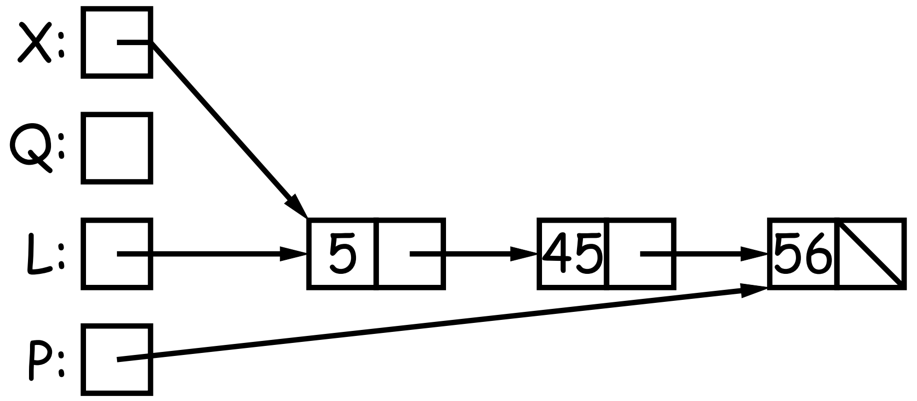


Destructive Incrementing

Destructive solutions may modify objects in the original list to save time or space:

```
/** Destructively add N to L's items. */
static IntList dincrList(IntList P, int n) {
  if (P == null)
    return null;
  else {
    P.head += n;
    P.tail = dincrList(P.tail, n);
    return P;
  }
}
/** Destructively add N to L's items. */
static IntList dincrList(IntList L, int n)
{
  // 'for' can do more than count!
  for (IntList p = L; p != null; p = p.tail)
    p.head += n;
  return L;
}
```

```
X = IntList.list(3, 43, 56);
/* IntList.list from HW #1 */
Q = dincrList(X, 2);
```

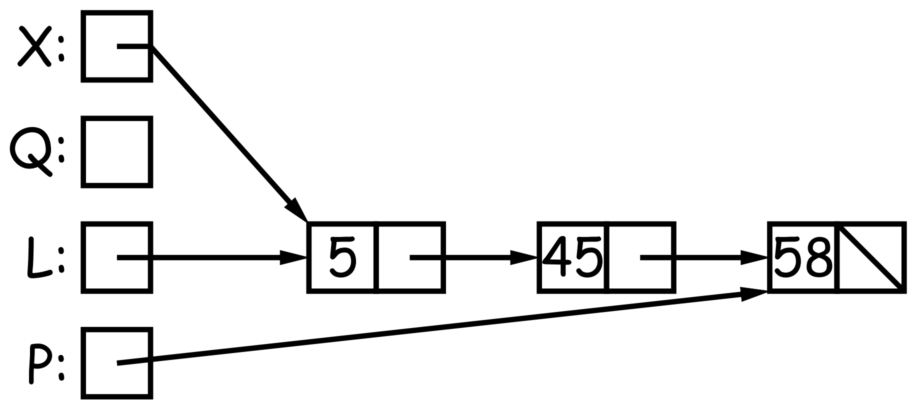


Destructive Incrementing

Destructive solutions may modify objects in the original list to save time or space:

```
/** Destructively add N to L's items. */
static IntList dincrList(IntList P, int n) {
  if (P == null)
    return null;
  else {
    P.head += n;
    P.tail = dincrList(P.tail, n);
    return P;
  }
}
/** Destructively add N to L's items. */
static IntList dincrList(IntList L, int n)
{
  // 'for' can do more than count!
  for (IntList p = L; p != null; p = p.tail)
    p.head += n;
  return L;
}
```

```
X = IntList.list(3, 43, 56);
/* IntList.list from HW #1 */
Q = dincrList(X, 2);
```

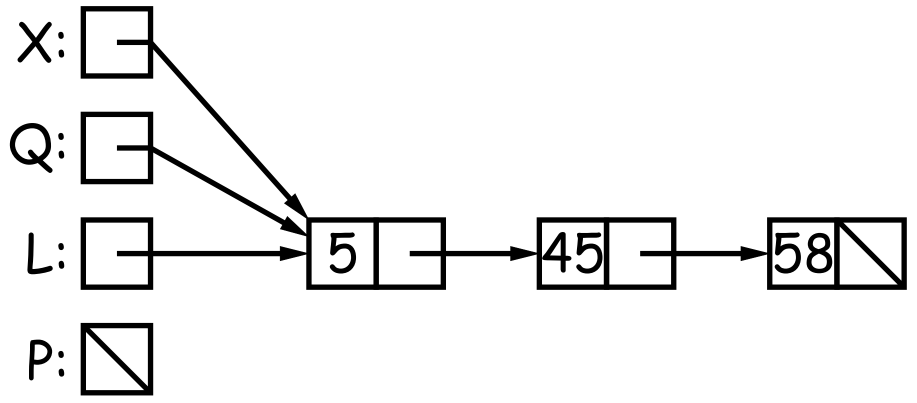


Destructive Incrementing

Destructive solutions may modify objects in the original list to save time or space:

```
/** Destructively add N to L's items. */
static IntList dincrList(IntList P, int n) {
  if (P == null)
    return null;
  else {
    P.head += n;
    P.tail = dincrList(P.tail, n);
    return P;
  }
}
/** Destructively add N to L's items. */
static IntList dincrList(IntList L, int n)
{
  // 'for' can do more than count!
  for (IntList p = L; p != null; p = p.tail)
    p.head += n;
  return L;
}
```

```
X = IntList.list(3, 43, 56);
/* IntList.list from HW #1 */
Q = dincrList(X, 2);
```



Another Example: Non-destructive List Deletion

If L is the list [2, 1, 2, 9, 2], we want `removeAll(L,2)` to be the new list [1, 9].

```
/** The list resulting from removing all instances of X from L
 *  non-destructively. */
static IntList removeAll(IntList L, int x) {
  if (L == null)
     return /*( null with all x's removed )*/;
  else if (L.head == x)
     return /*( L with all x's removed (L!=null, L.head==x) )*/;
  else
     return /*( L with all x's removed (L!=null, L.head!=x) )*/;
}
```

Another Example: Non-destructive List Deletion

If L is the list [2, 1, 2, 9, 2], we want `removeAll(L,2)` to be the new list [1, 9].

```
/** The list resulting from removing all instances of X from L
 *  non-destructively. */
static IntList removeAll(IntList L, int x) {
  if (L == null)
     return null;
  else if (L.head == x)
     return /*( L with all x's removed (L!=null, L.head==x) )*/;
  else
     return /*( L with all x's removed (L!=null, L.head!=x) )*/;
}
```

Another Example: Non-destructive List Deletion

If L is the list [2, 1, 2, 9, 2], we want `removeAll(L,2)` to be the new list [1, 9].

```
/** The list resulting from removing all instances of X from L
 *  non-destructively. */
static IntList removeAll(IntList L, int x) {
  if (L == null)
     return null;
  else if (L.head == x)
     return removeAll(L.tail, x);
  else
     return /*( L with all x's removed (L!=null, L.head!=x) )*/;
}
```

Another Example: Non-destructive List Deletion

If L is the list [2, 1, 2, 9, 2], we want `removeAll(L,2)` to be the new list [1, 9].

```
/** The list resulting from removing all instances of X from L
 *  non-destructively. */
static IntList removeAll(IntList L, int x) {
  if (L == null)
     return null;
  else if (L.head == x)
     return removeAll(L.tail, x);
  else
     return new IntList(L.head, removeAll(L.tail, x));
}
```

Iterative Non-destructive List Deletion

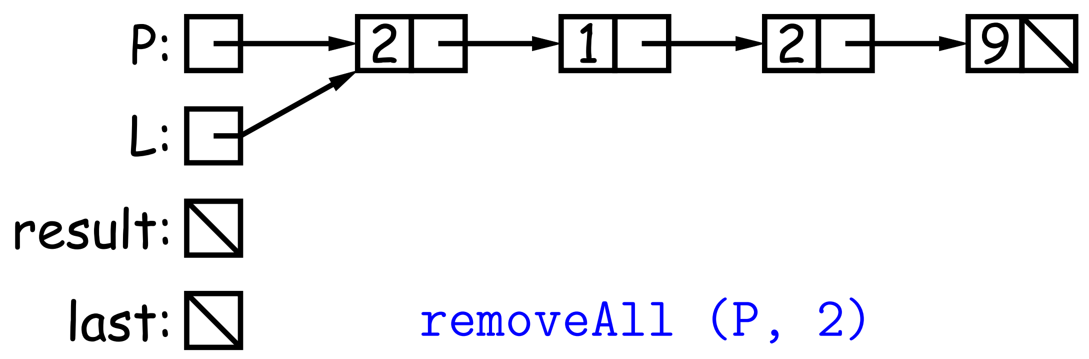
Same as before, but use front-to-back iteration rather than recursion.

```
/** The list resulting from removing all instances
 *  of X from L non-destructively. */
static IntList removeAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  for ( ; L != null; L = L.tail) {
    if (x == L.head)
      continue;
    else if (last == null)
      result = last = new IntList(L.head, null);
    else
      last = last.tail = new IntList(L.head, null);
  }
  return result;
}
```

Iterative Non-destructive List Deletion

Same as before, but use front-to-back iteration rather than recursion.

```
/** The list resulting from removing all instances
 *  of X from L non-destructively. */
static IntList removeAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  for ( ; L != null; L = L.tail) {
    if (x == L.head)
      continue;
    else if (last == null)
      result = last = new IntList(L.head, null);
    else
      last = last.tail = new IntList(L.head, null);
  }
  return result;
}
```

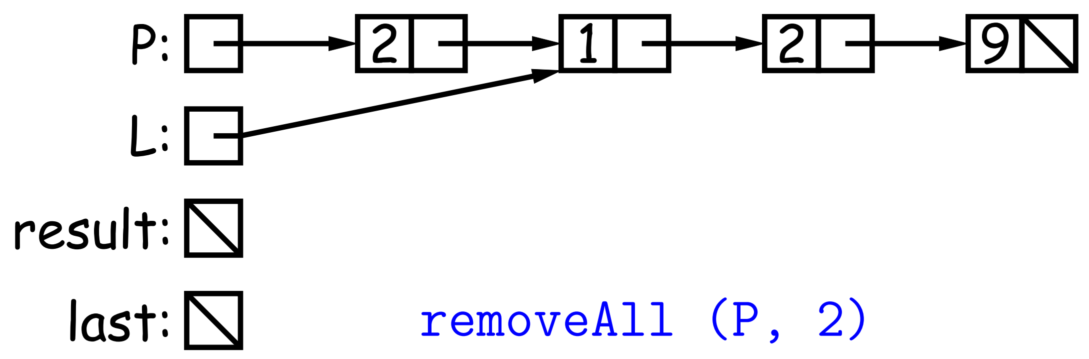


removeAll (P, 2)

Iterative Non-destructive List Deletion

Same as before, but use front-to-back iteration rather than recursion.

```
/** The list resulting from removing all instances
 *  of X from L non-destructively. */
static IntList removeAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  for ( ; L != null; L = L.tail) {
    if (x == L.head)
      continue;
    else if (last == null)
      result = last = new IntList(L.head, null);
    else
      last = last.tail = new IntList(L.head, null);
  }
  return result;
}
```



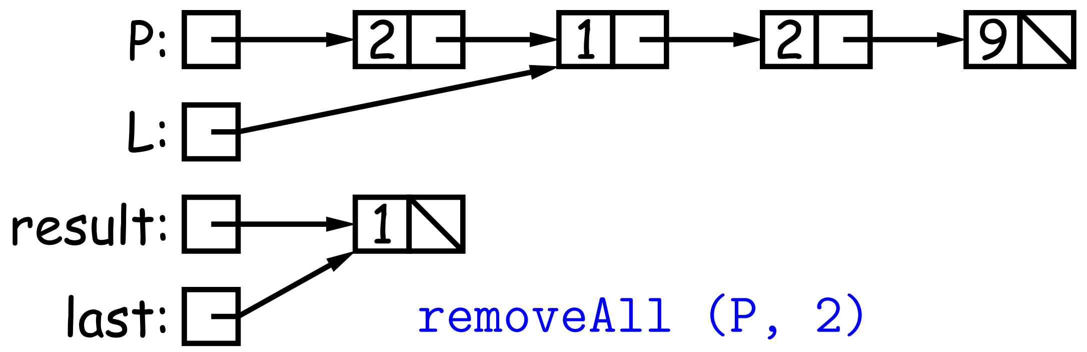
`removeAll (P, 2)`

P does *not* change!

Iterative Non-destructive List Deletion

Same as before, but use front-to-back iteration rather than recursion.

```
/** The list resulting from removing all instances
 *  of X from L non-destructively. */
static IntList removeAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  for ( ; L != null; L = L.tail) {
    if (x == L.head)
      continue;
    else if (last == null)
      result = last = new IntList(L.head, null);
    else
      last = last.tail = new IntList(L.head, null);
  }
  return result;
}
```



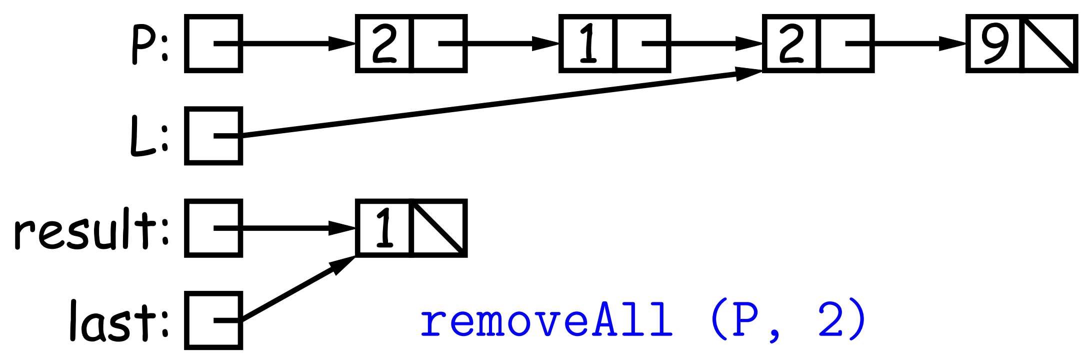
`removeAll (P, 2)`

P does *not* change!

Iterative Non-destructive List Deletion

Same as before, but use front-to-back iteration rather than recursion.

```
/** The list resulting from removing all instances
 *  of X from L non-destructively. */
static IntList removeAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  for ( ; L != null; L = L.tail) {
    if (x == L.head)
      continue;
    else if (last == null)
      result = last = new IntList(L.head, null);
    else
      last = last.tail = new IntList(L.head, null);
  }
  return result;
}
```



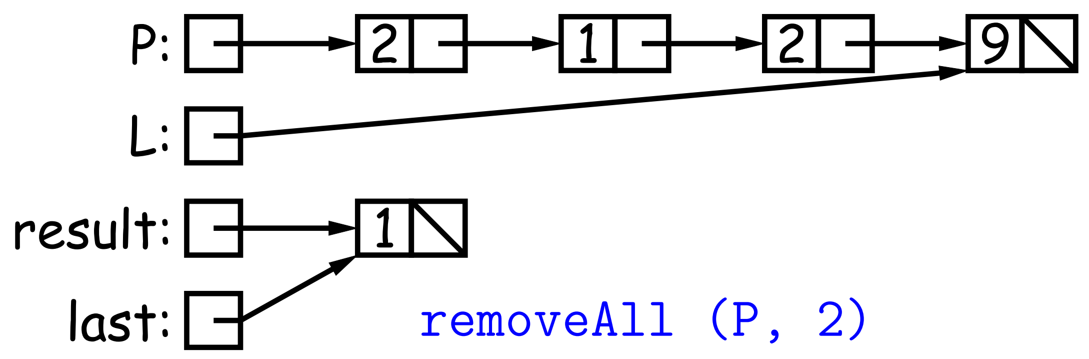
`removeAll (P, 2)`

P does *not* change!

Iterative Non-destructive List Deletion

Same as before, but use front-to-back iteration rather than recursion.

```
/** The list resulting from removing all instances
 *  of X from L non-destructively. */
static IntList removeAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  for ( ; L != null; L = L.tail) {
    if (x == L.head)
      continue;
    else if (last == null)
      result = last = new IntList(L.head, null);
    else
      last = last.tail = new IntList(L.head, null);
  }
  return result;
}
```



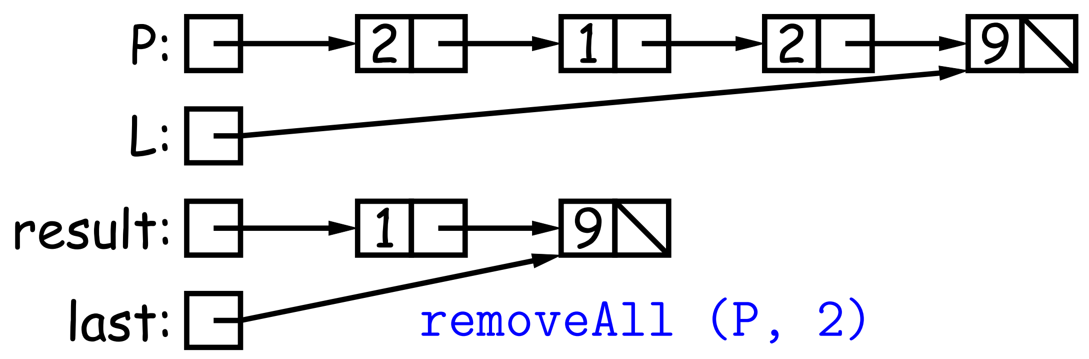
removeAll (P, 2)

P does *not* change!

Iterative Non-destructive List Deletion

Same as before, but use front-to-back iteration rather than recursion.

```
/** The list resulting from removing all instances
 *  of X from L non-destructively. */
static IntList removeAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  for ( ; L != null; L = L.tail) {
    if (x == L.head)
      continue;
    else if (last == null)
      result = last = new IntList(L.head, null);
    else
      last = last.tail = new IntList(L.head, null);
  }
  return result;
}
```



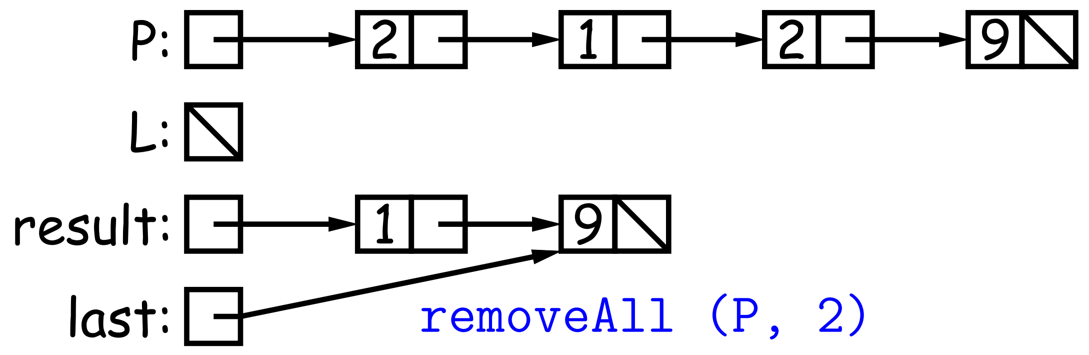
removeAll (P, 2)

P does *not* change!

Iterative Non-destructive List Deletion

Same as before, but use front-to-back iteration rather than recursion.

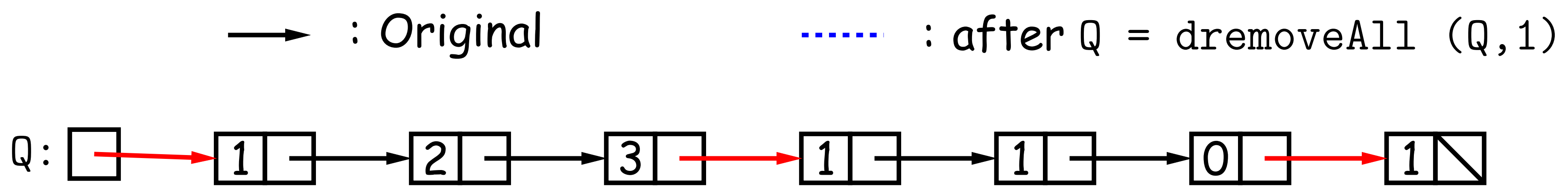
```
/** The list resulting from removing all instances
 *  of X from L non-destructively. */
static IntList removeAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  for ( ; L != null; L = L.tail) {
    if (x == L.head)
      continue;
    else if (last == null)
      result = last = new IntList(L.head, null);
    else
      last = last.tail = new IntList(L.head, null);
  }
  return result;
}
```



removeAll (P, 2)

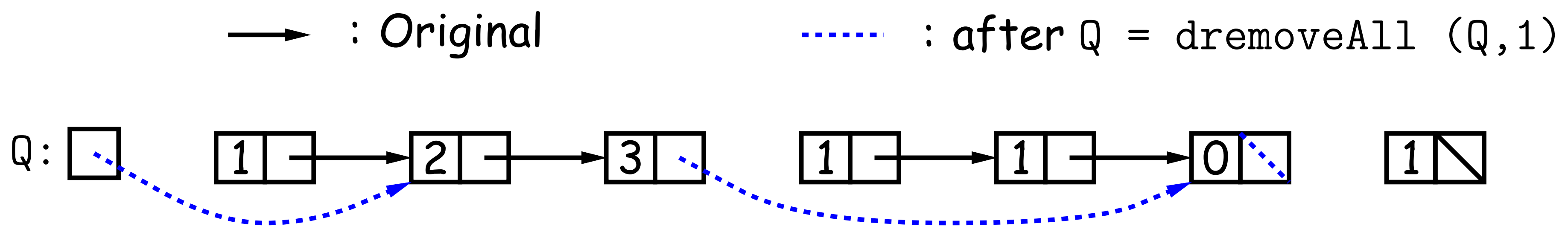
P does not change!

Destructive Deletion



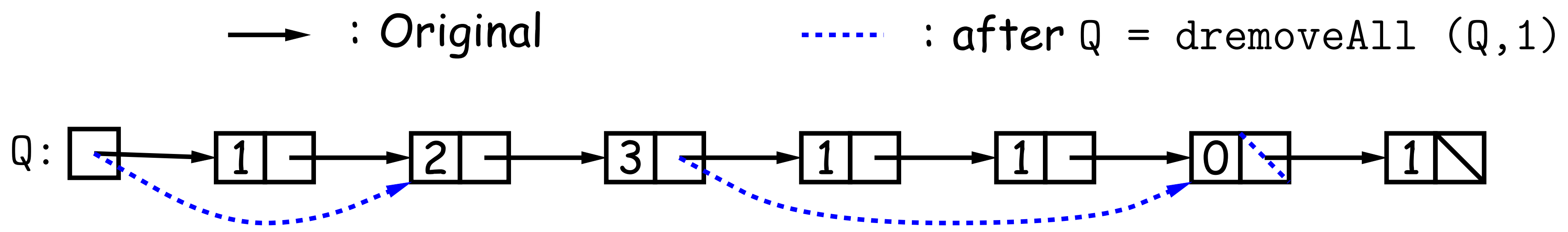
```
/** The list resulting from removing all instances of X from L.
 *  The original list may be destroyed. */
static IntList dremoveAll(IntList L, int x) {
  if (L == null)
     return /*( null with all x's removed )*/;
  else if (L.head == x)
     return /*( L with all x's removed (L != null) )*/;
  else {
     /*{ Remove all x's from L's tail. }*/;
     return L;
  }
}
```

Destructive Deletion



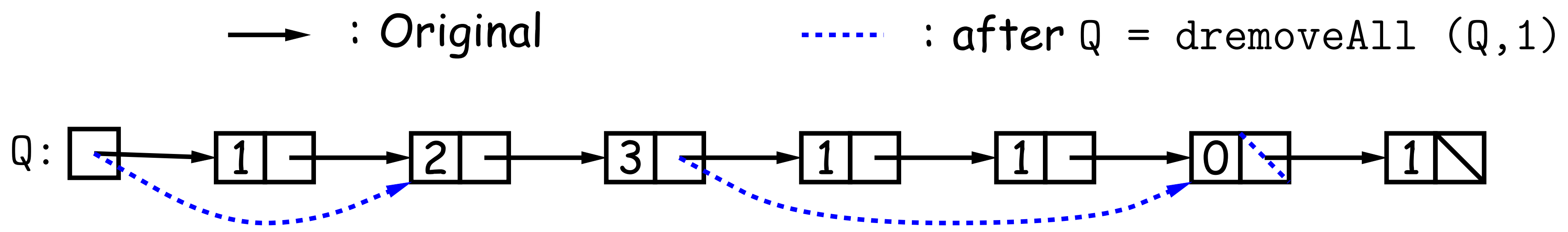
```
/** The list resulting from removing all instances of X from L.
 *  The original list may be destroyed. */
static IntList dremoveAll(IntList L, int x) {
  if (L == null)
     return /*( null with all x's removed )*/;
  else if (L.head == x)
     return /*( L with all x's removed (L != null) )*/;
  else {
     /*{ Remove all x's from L's tail. }*/;
     return L;
  }
}
```

Destructive Deletion



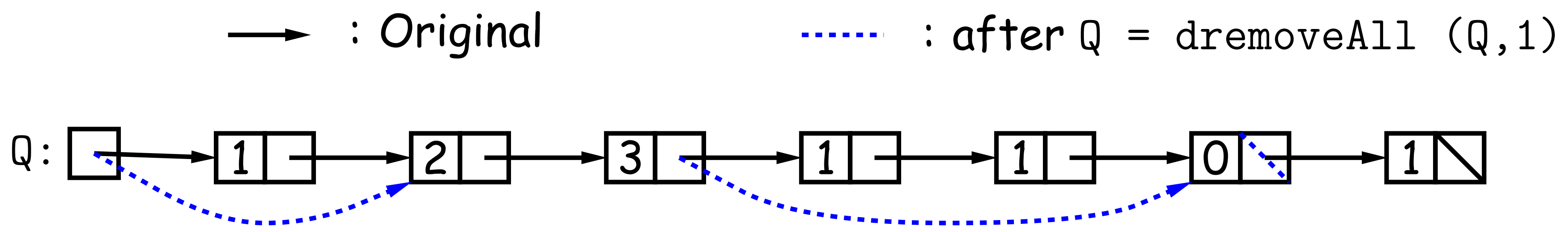
```
/** The list resulting from removing all instances of X from L.
 *  The original list may be destroyed. */
static IntList dremoveAll(IntList L, int x) {
  if (L == null)
     return /*( null with all x's removed )*/;
  else if (L.head == x)
     return /*( L with all x's removed (L != null) )*/;
  else {
     /*{ Remove all x's from L's tail. }*/;
     return L;
  }
}
```

Destructive Deletion



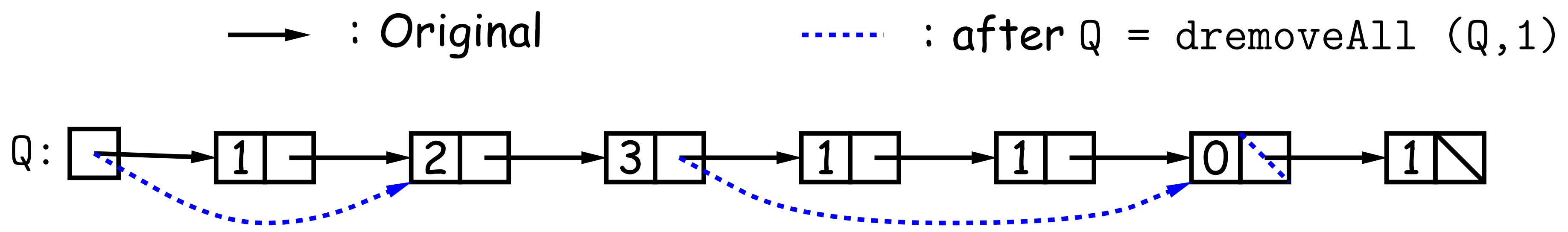
```
/** The list resulting from removing all instances of X from L.
 *  The original list may be destroyed. */
static IntList dremoveAll(IntList L, int x) {
  if (L == null)
     return /*( null with all x's removed )*/;
  else if (L.head == x)
     return /*( L with all x's removed (L != null) )*/;
  else {
     /*{ Remove all x's from L's tail. }*/;
     return L;
  }
}
```

Destructive Deletion



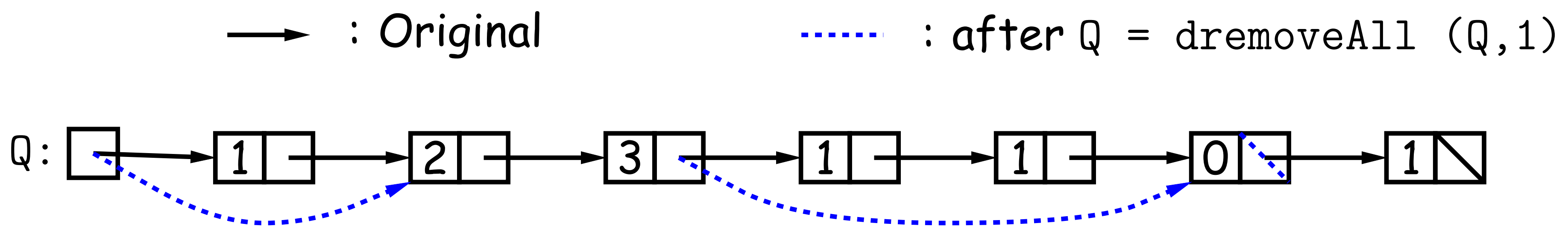
```
/** The list resulting from removing all instances of X from L.
 *  The original list may be destroyed. */
static IntList dremoveAll(IntList L, int x) {
  if (L == null)
     return null;
  else if (L.head == x)
     return /*( L with all x's removed (L != null) )*/;
  else {
     /*{ Remove all x's from L's tail. }*/;
     return L;
  }
}
```

Destructive Deletion



```
/** The list resulting from removing all instances of X from L.
 *  The original list may be destroyed. */
static IntList dremoveAll(IntList L, int x) {
  if (L == null)
     return
  else if (L.head == x)
     return dremoveAll(L.tail, x);
  else {
     /*{ Remove all x's from L's tail. }*/;
     return L;
  }
}
```

Destructive Deletion



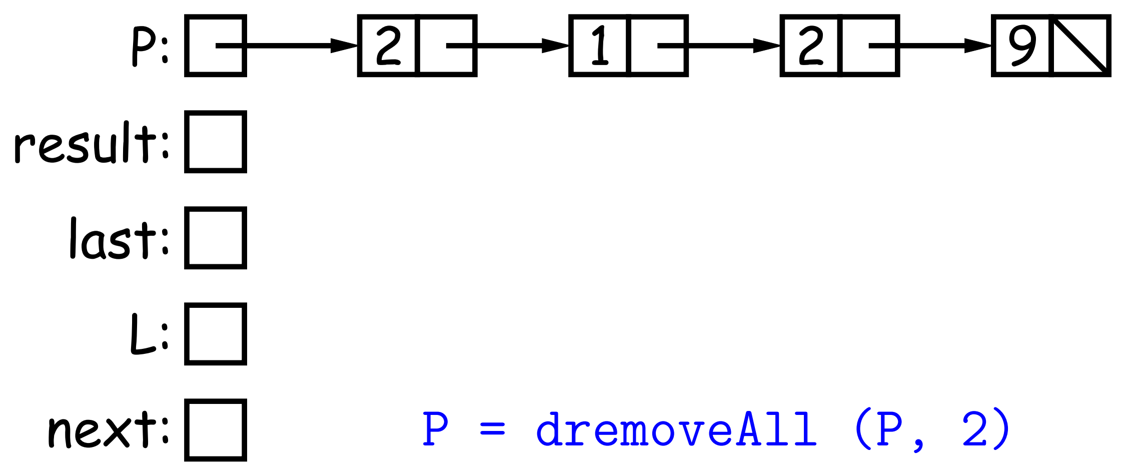
```
/** The list resulting from removing all instances of X from L.
 *  The original list may be destroyed. */
static IntList dremoveAll(IntList L, int x) {
  if (L == null)
     return
  else if (L.head == x)
     return dremoveAll(L.tail, x);
  else {
     L.tail = dremoveAll(L.tail, x);
     return L;
  }
}
```

Iterative Destructive Deletion

```
/** The list resulting from removing all X's from L
 *  destructively. */
static IntList dremoveAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  while (L != null) {
    IntList next = L.tail;
    if (x != L.head) {
      if (last == null)
        result = last = L;
      else
        last = last.tail = L;
      L.tail = null;
    }
    L = next;
  }
  return result;
}
```

Iterative Destructive Deletion

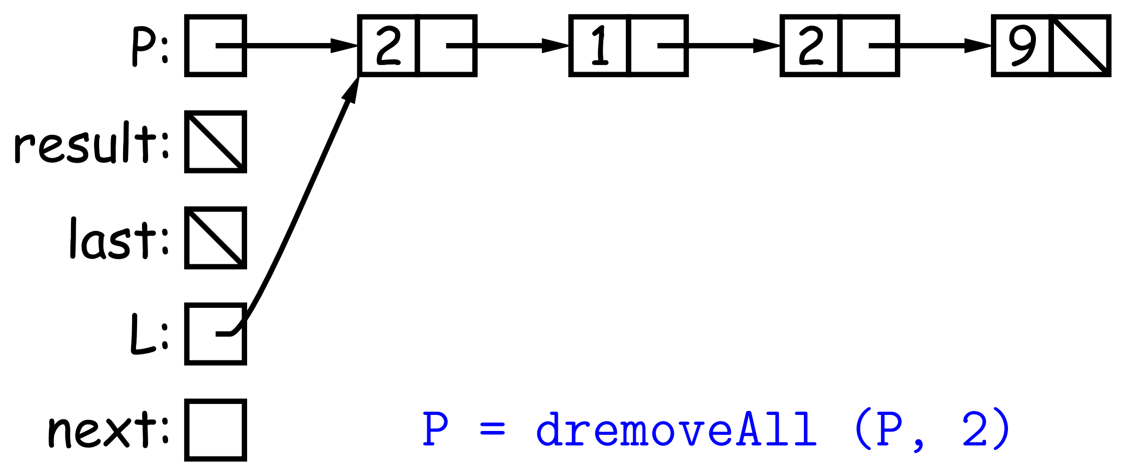
```
/** The list resulting from removing all X's from L
 *  destructively. */
static IntList dremoveAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  while (L != null) {
    IntList next = L.tail;
    if (x != L.head) {
      if (last == null)
        result = last = L;
      else
        last = last.tail = L;
      L.tail = null;
    }
    L = next;
  }
  return result;
}
```



```
P = dremoveAll (P, 2)
```

Iterative Destructive Deletion

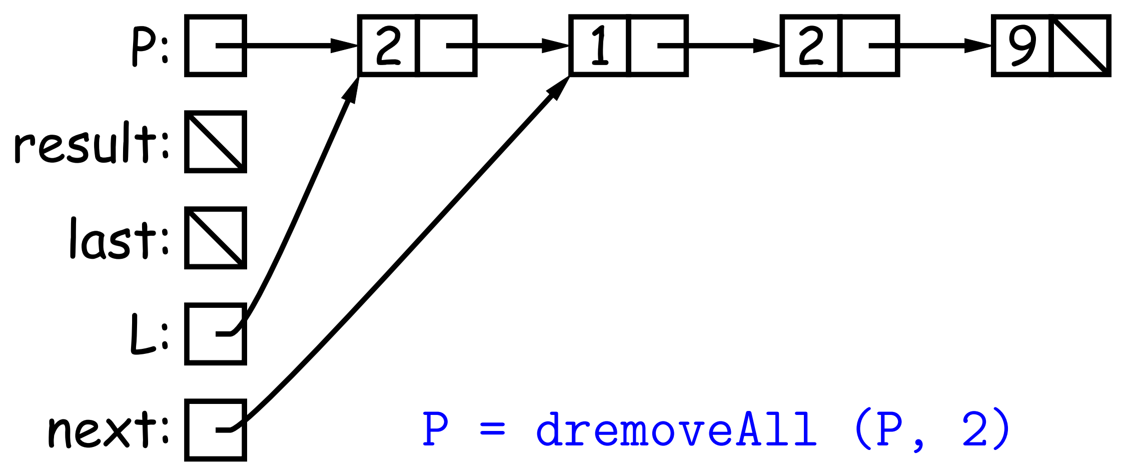
```
/** The list resulting from removing all X's from L
 *  destructively. */
static IntList dremoveAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  while (L != null) {
    IntList next = L.tail;
    if (x != L.head) {
      if (last == null)
        result = last = L;
      else
        last = last.tail = L;
      L.tail = null;
    }
    L = next;
  }
  return result;
}
```



```
P = dremoveAll (P, 2)
```

Iterative Destructive Deletion

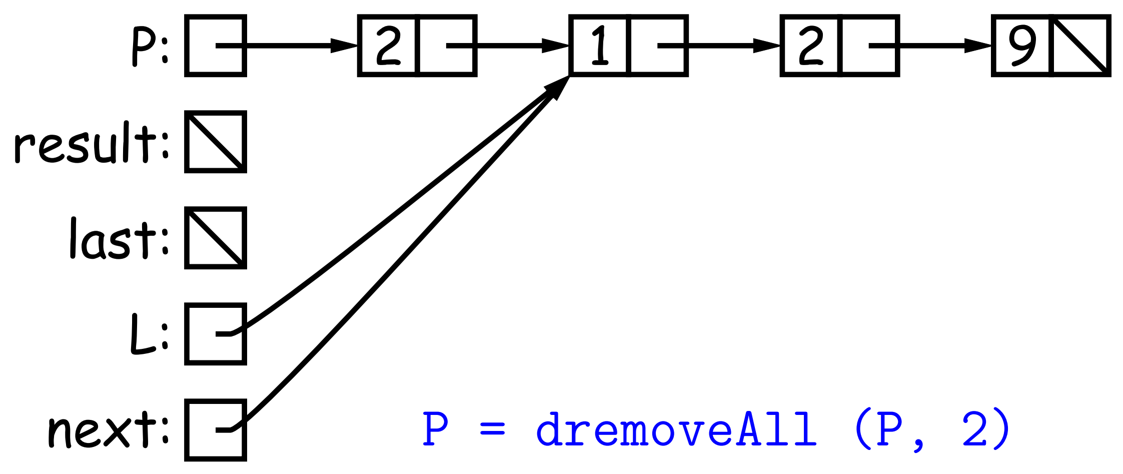
```
/** The list resulting from removing all X's from L
 *  destructively. */
static IntList dremoveAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  while (L != null) {
    IntList next = L.tail;
    if (x != L.head) {
      if (last == null)
        result = last = L;
      else
        last = last.tail = L;
      L.tail = null;
    }
    L = next;
  }
  return result;
}
```



```
P = dremoveAll (P, 2)
```

Iterative Destructive Deletion

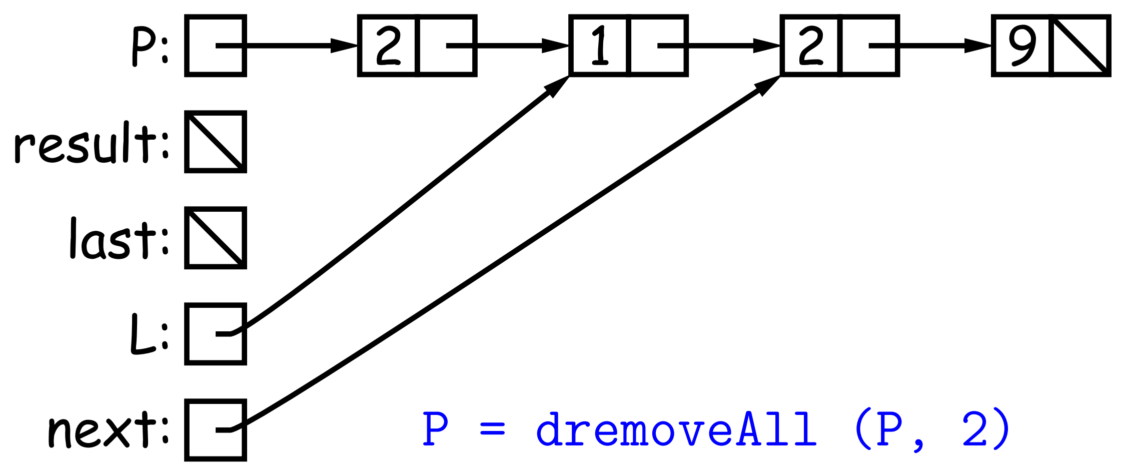
```
/** The list resulting from removing all X's from L
 *  destructively. */
static IntList dremoveAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  while (L != null) {
    IntList next = L.tail;
    if (x != L.head) {
      if (last == null)
        result = last = L;
      else
        last = last.tail = L;
      L.tail = null;
    }
    L = next;
  }
  return result;
}
```



P = dremoveAll (P, 2)

Iterative Destructive Deletion

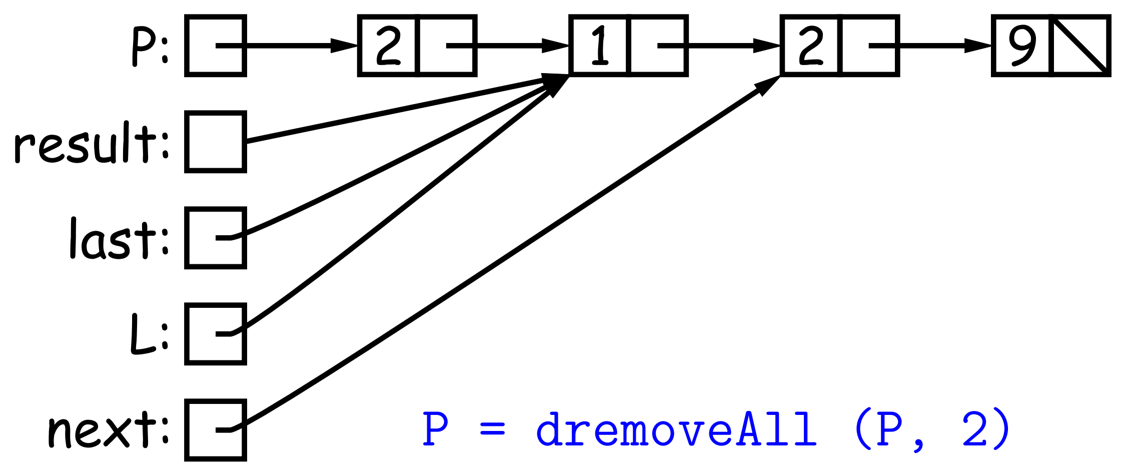
```
/** The list resulting from removing all X's from L
 *  destructively. */
static IntList dremoveAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  while (L != null) {
    IntList next = L.tail;
    if (x != L.head) {
      if (last == null)
        result = last = L;
      else
        last = last.tail = L;
      L.tail = null;
    }
    L = next;
  }
  return result;
}
```



`P = dremoveAll (P, 2)`

Iterative Destructive Deletion

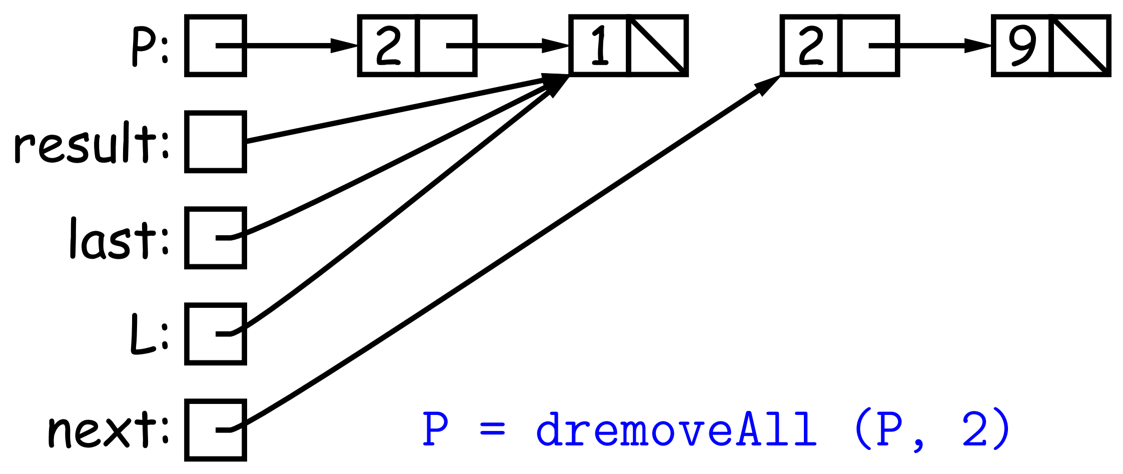
```
/** The list resulting from removing all X's from L
 *  destructively. */
static IntList dremoveAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  while (L != null) {
    IntList next = L.tail;
    if (x != L.head) {
      if (last == null)
        result = last = L;
      else
        last = last.tail = L;
      L.tail = null;
    }
    L = next;
  }
  return result;
}
```



```
P = dremoveAll (P, 2)
```

Iterative Destructive Deletion

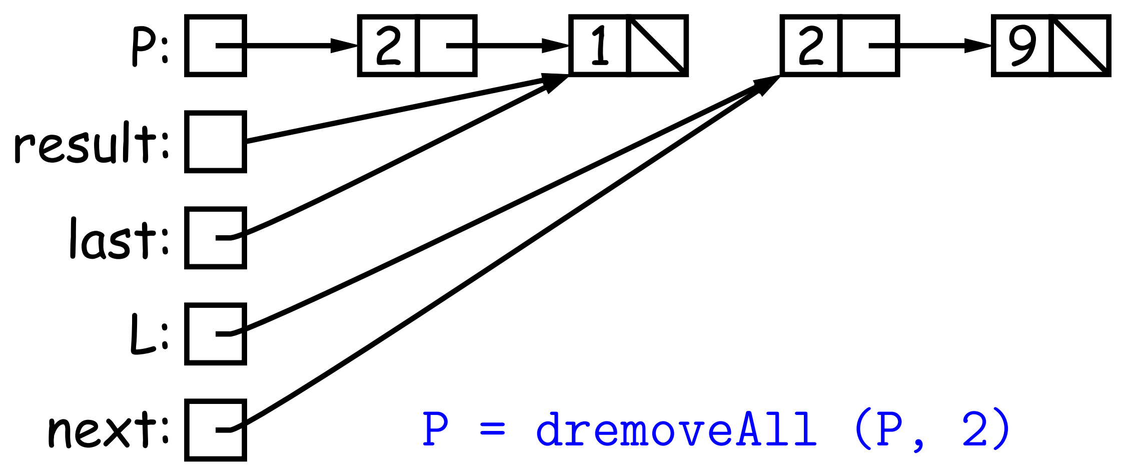
```
/** The list resulting from removing all X's from L
 *  destructively. */
static IntList dremoveAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  while (L != null) {
    IntList next = L.tail;
    if (x != L.head) {
      if (last == null)
        result = last = L;
      else
        last = last.tail = L;
      L.tail = null;
    }
    L = next;
  }
  return result;
}
```



`P = dremoveAll (P, 2)`

Iterative Destructive Deletion

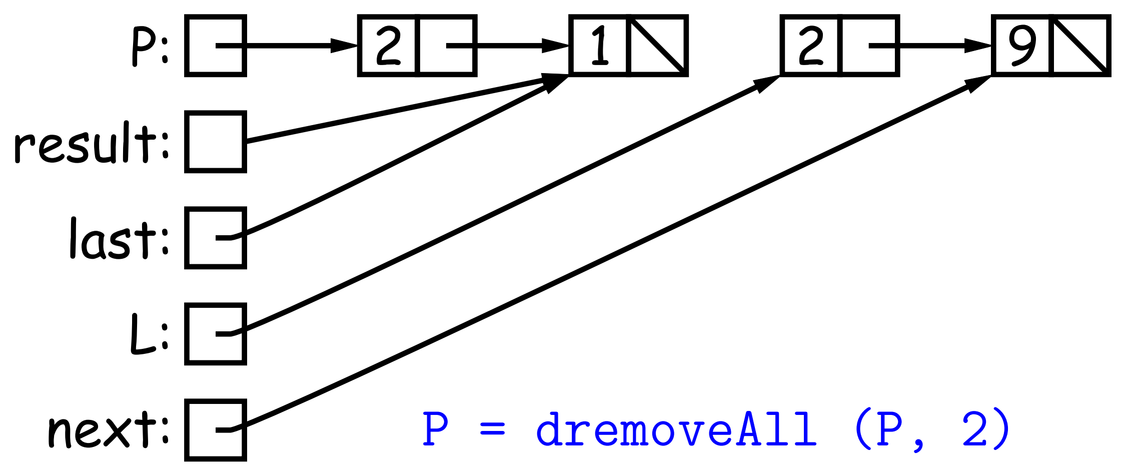
```
/** The list resulting from removing all X's from L
 *  destructively. */
static IntList dremoveAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  while (L != null) {
    IntList next = L.tail;
    if (x != L.head) {
      if (last == null)
        result = last = L;
      else
        last = last.tail = L;
      L.tail = null;
    }
    L = next;
  }
  return result;
}
```



```
P = dremoveAll (P, 2)
```

Iterative Destructive Deletion

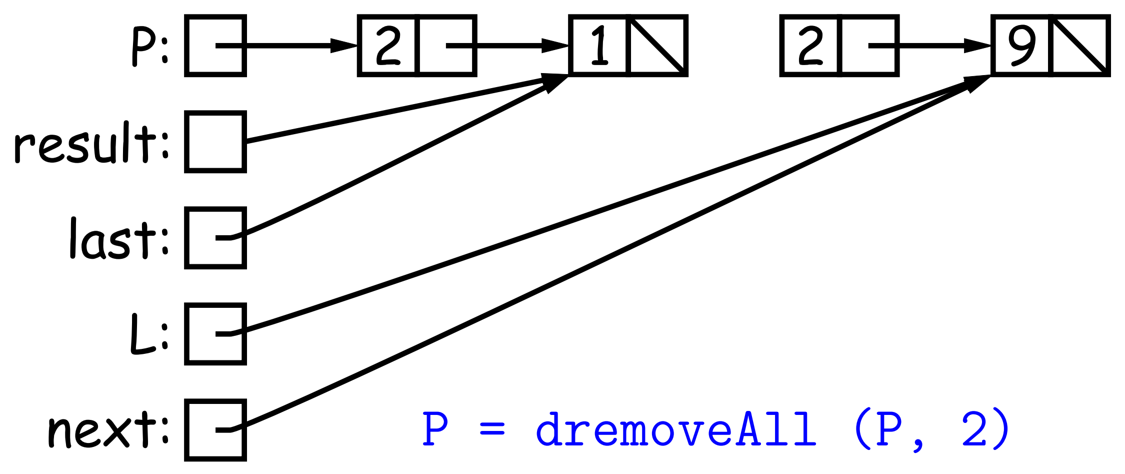
```
/** The list resulting from removing all X's from L
 *  destructively. */
static IntList dremoveAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  while (L != null) {
    IntList next = L.tail;
    if (x != L.head) {
      if (last == null)
        result = last = L;
      else
        last = last.tail = L;
      L.tail = null;
    }
    L = next;
  }
  return result;
}
```



```
P = dremoveAll (P, 2)
```

Iterative Destructive Deletion

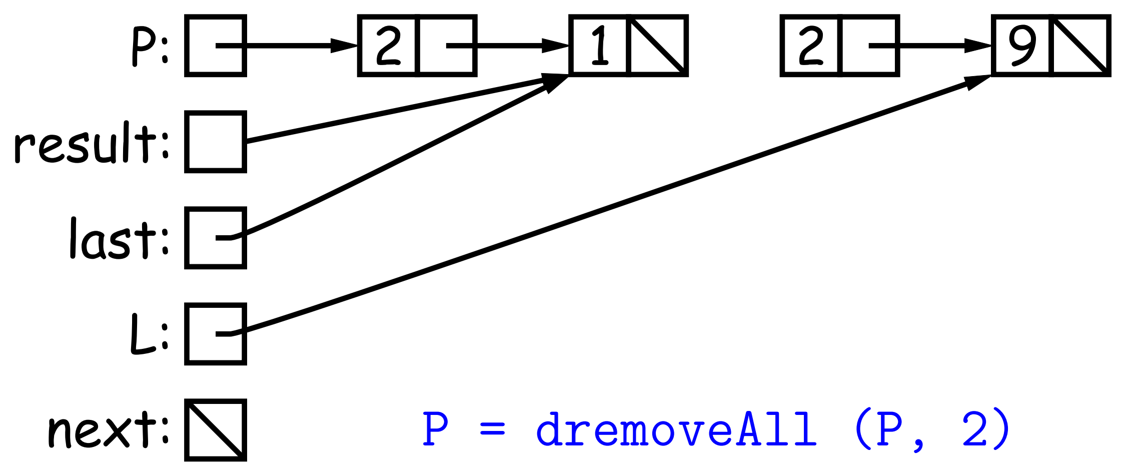
```
/** The list resulting from removing all X's from L
 *  destructively. */
static IntList dremoveAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  while (L != null) {
    IntList next = L.tail;
    if (x != L.head) {
      if (last == null)
        result = last = L;
      else
        last = last.tail = L;
      L.tail = null;
    }
    L = next;
  }
  return result;
}
```



P = dremoveAll (P, 2)

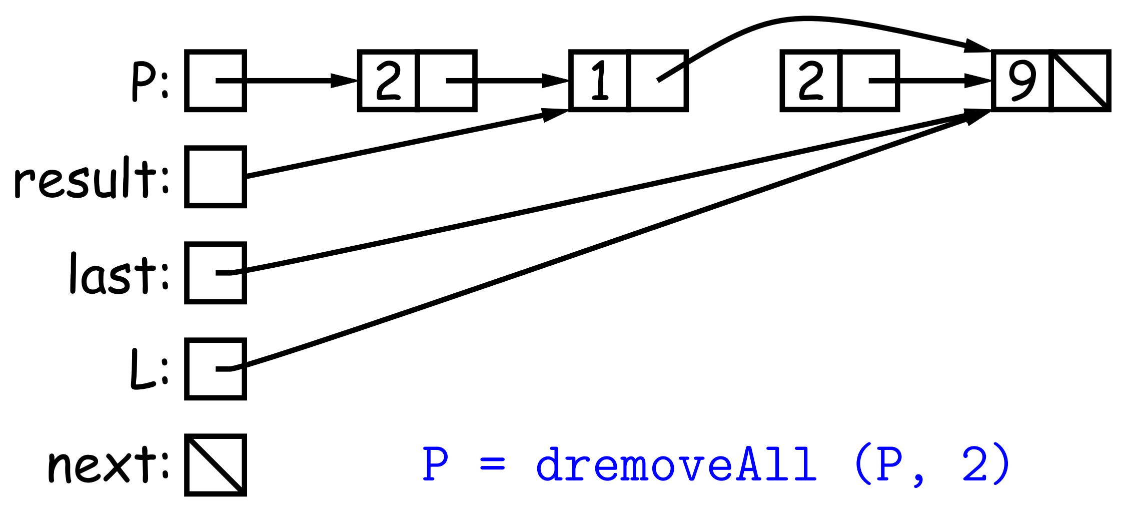
Iterative Destructive Deletion

```
/** The list resulting from removing all X's from L
 *  destructively. */
static IntList dremoveAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  while (L != null) {
    IntList next = L.tail;
    if (x != L.head) {
      if (last == null)
        result = last = L;
      else
        last = last.tail = L;
      L.tail = null;
    }
    L = next;
  }
  return result;
}
```



Iterative Destructive Deletion

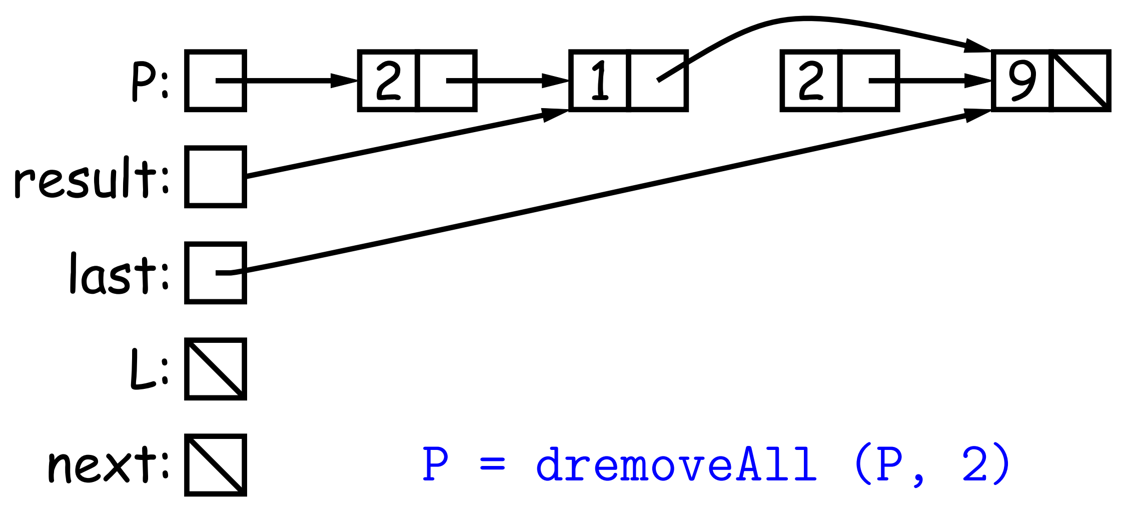
```
/** The list resulting from removing all X's from L
 *  destructively. */
static IntList dremoveAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  while (L != null) {
    IntList next = L.tail;
    if (x != L.head) {
      if (last == null)
        result = last = L;
      else
        last = last.tail = L;
      L.tail = null;
    }
    L = next;
  }
  return result;
}
```



P = dremoveAll (P, 2)

Iterative Destructive Deletion

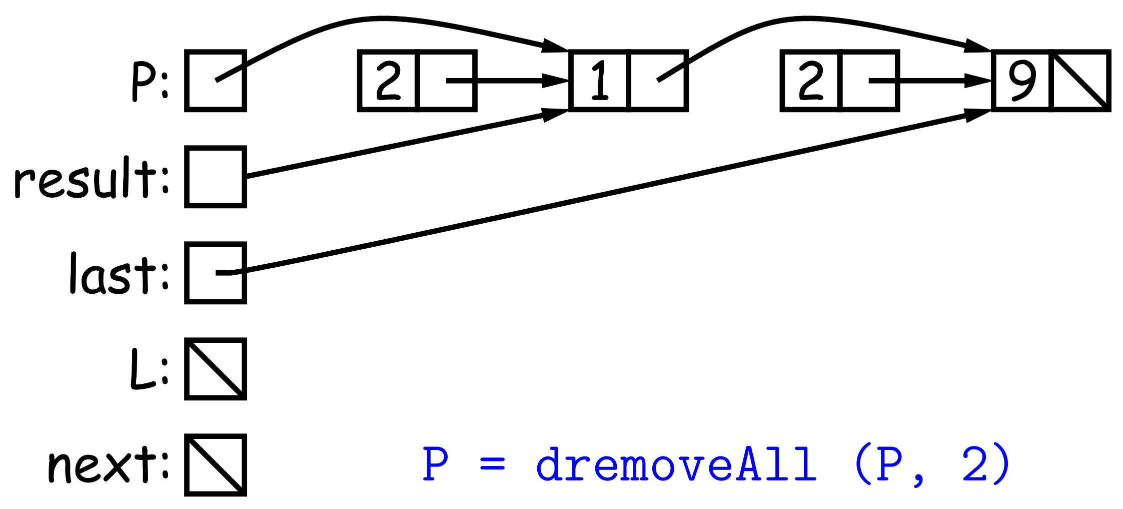
```
/** The list resulting from removing all X's from L
 *  destructively. */
static IntList dremoveAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  while (L != null) {
    IntList next = L.tail;
    if (x != L.head) {
      if (last == null)
        result = last = L;
      else
        last = last.tail = L;
      L.tail = null;
    }
    L = next;
  }
  return result;
}
```



```
P = dremoveAll (P, 2)
```

Iterative Destructive Deletion

```
/** The list resulting from removing all X's from L
 *  destructively. */
static IntList dremoveAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  while (L != null) {
    IntList next = L.tail;
    if (x != L.head) {
      if (last == null)
        result = last = L;
      else
        last = last.tail = L;
      L.tail = null;
    }
    L = next;
  }
  return result;
}
```



```
P = dremoveAll (P, 2)
```

Aside: How to Write a Loop (in Theory)

- Try to give a description of how things look on *any arbitrary iteration* of the loop.
- This description is known as a *loop invariant,* because it is always true at the start of each iteration.
- The loop body then must
  - Start from any situation consistent with the invariant;
  - Make progress in such a way as to make the invariant true again.

```
// Invariant must be true here
while (condition) { // condition must not have side-effects.
  // (Invariant will necessarily be true here.)
  loop body
  // Invariant must again be true here
}
// Invariant true and condition false.
```

- So if our loop gets the desired answer whenever *Invariant* is true and *condition* false, our job is done!

Relationship to Recursion

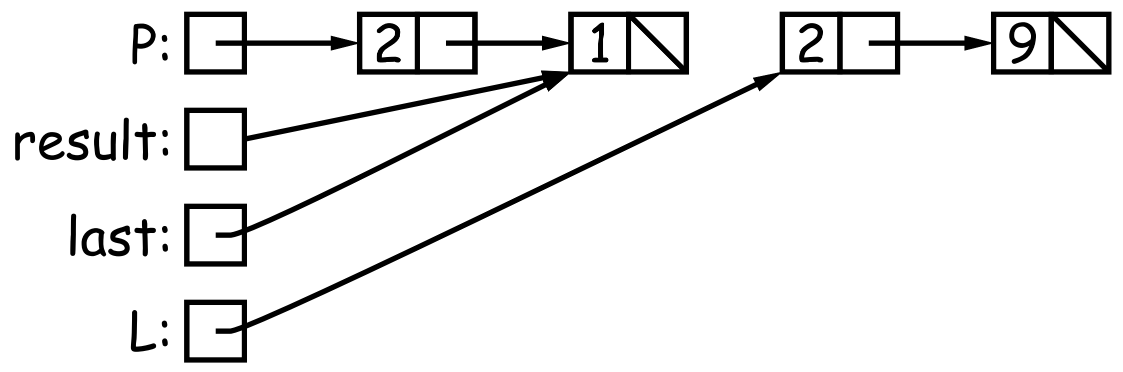
- Another way to see this is to consider an equivalent recursive procedure:

```
/** Assuming Invariant, produce a situation where Inveriant
 *  is true and condition is false. */
void loop() {
    // Invariant assumed true here.
    if (condition) {
        loop body
        // Invariant must be true here.
        loop()
        // Invariant true here and condition false.
    }
}
```

- Here, the invariant is the precondition of the function **loop**.
- The loop maintains the invariant while making the condition false.
- Idea is to arrange that our actual goal is implied by this post-condition.

Example: Loop Invariant for dremoveAll

```
/** The list resulting from removing all X's from L
 *  destructively. */
static IntList dremoveAll(IntList L, int x) {
  IntList result, last;
  result = last = null;
  while ** (L != null) {
    IntList next = L.tail;
    if (x != L.head) {
      if (last == null)
        result = last = L;
      else
        last = last.tail = L;
      L.tail = null;
    }
    L = next;
  }
  return result;
}
```



`P = dremoveAll (P, 2)`

** Invariant:

- `result` points to the list of items in the final result except for those from L onward.
- `L` points to an unchanged tail of the original list of items in `L`.
- `last` points to the last item in `result` or is null if `result` is null.

Recreation

What is the sum of the coefficients of

$$(1 - 3x + 3x^2)^{743}(1 + 3x - 3x^2)^{744}$$

after expanding and collecting terms?

CS61B Lecture #5: Arrays

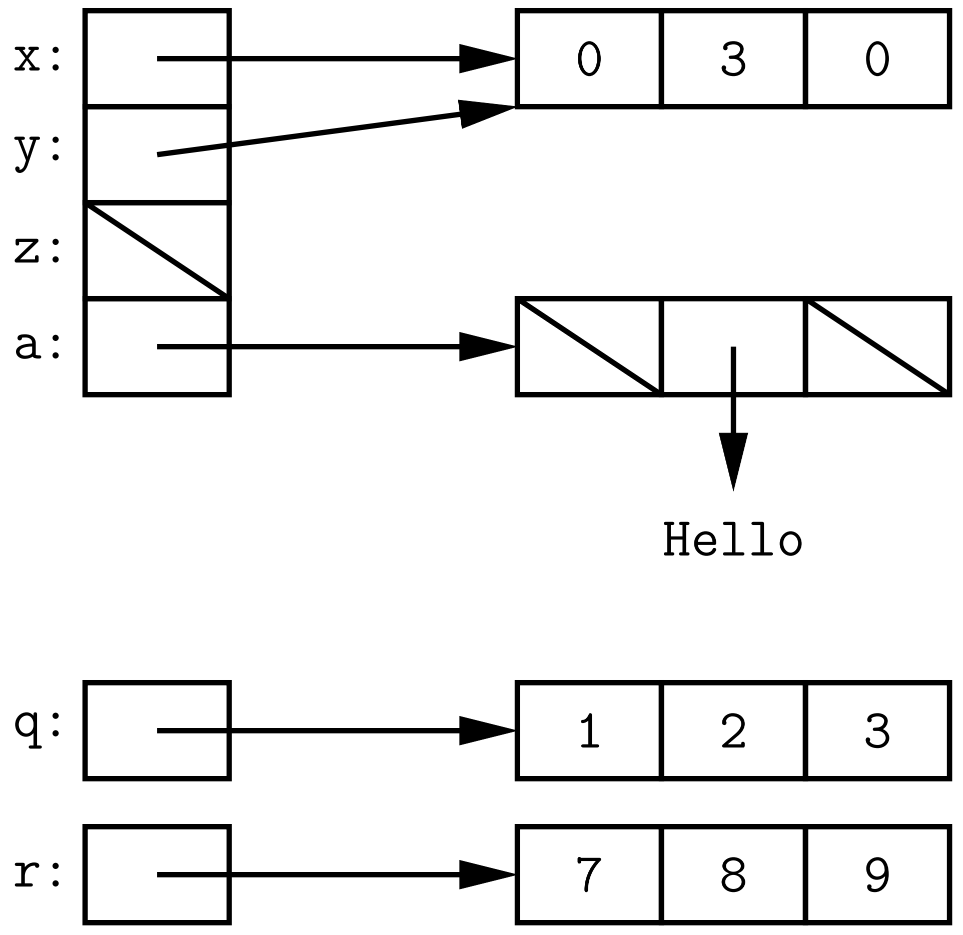
- An array is a structured container whose components are
  - **length**, a fixed integer.
  - a sequence of **length** simple containers of the same type, numbered from 0.
  - (.length field usually implicit in diagrams.)
- Arrays are anonymous, like other structured containers.
- Always referred to with pointers.
- For array pointed to by `A`,
  - Length is `A.length`
  - Numbered component i is `A[`i`]` (i is the *index*)
  - Important feature: index can be *any integer expression*.

A Few Samples

Java

```
int[] x, y, z;
String[] a;
x = new int[3];
y = x;
a = new String[3];
x[1] = 2;
y[1] = 3;
a[1] = "Hello";

int[] q;
q = new int[] { 1, 2, 3 };
// Short form for declarations:
int[] r = { 7, 8, 9 };
```

Results



Example: Accumulate Values

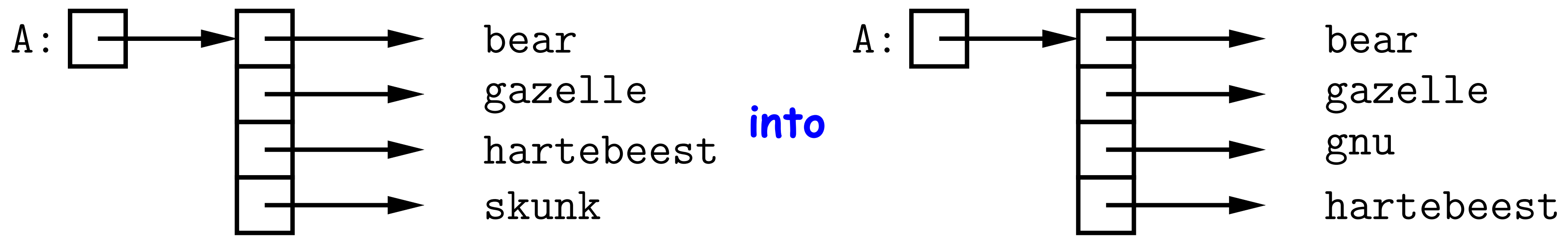
Problem: Sum up the elements of array *A*.

```
static int sum(int[] A) {
  int N;
  N = 0;                                            // New (1.5) syntax
  for (int i = 0; i < A.length; i += 1)             for (int x : A)
    N += A[i];                                         N += x;
  return N;
}
```

```
// For the hard-core: could have written

int N, i;
for (i=0, N=0; i<A.length; N += A[i], i += 1)
  { }   // or just ;

// But please don't: it's obscure.
```

Example: Insert into an Array

Problem: Want a call like `insert(A, 2, "gnu")` to convert (destructively)



```java
/** Insert X at location K in ARR, moving items K, K+1, ... to locations
 *  K+1, K+2, .... The last item in ARR is lost. */
static void insert (String[] arr, int k, String x) {
  for (int i = arr.length-1; i > k; i -= 1) // Why backwards?
    arr[i] = arr[i-1];
  /* Alternative to this loop:
        System.arraycopy(arr, k,  arr, k+1,  arr.length-k-1);*/
                           from       to        # to copy
  arr[k] = x;
}
```

(Aside) Java Shortcut

- **Useful tip:** Can write just '`arraycopy`' by including at the top of the source file:

  ```
  import static java.lang.System.arraycopy;
  ```

- This means "define the simple name `arraycopy` to be the equivalent of `java.lang.System.arraycopy` in the current source file."
- Can do the same for `out` so that you can write

  ```
  out.println(...);
  ```

  in place of

  ```
  System.out.println(...);
  ```

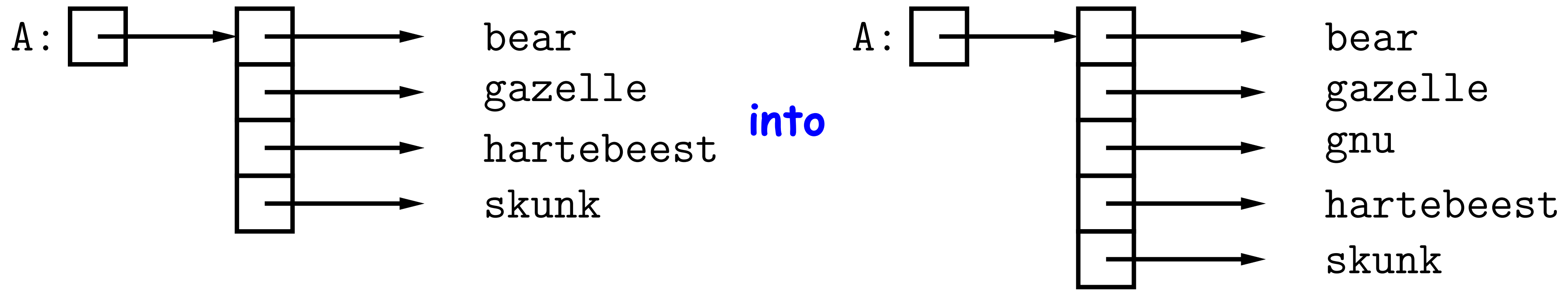
- Finally, a declaration like

  ```
  import static java.lang.Math.*;
  ```

  means "take all the (public) static definitions in `java.lang.Math` and make them available in this source file by their simple names (the name after the last dot)."
- Useful for functions like `sin`, `sqrt`, etc.

Growing an Array

Problem: Suppose that we want to change the description above, so that `A = insert2 (A, 2, "gnu")` does *not* shove "skunk" off the end, but instead "grows" the array.

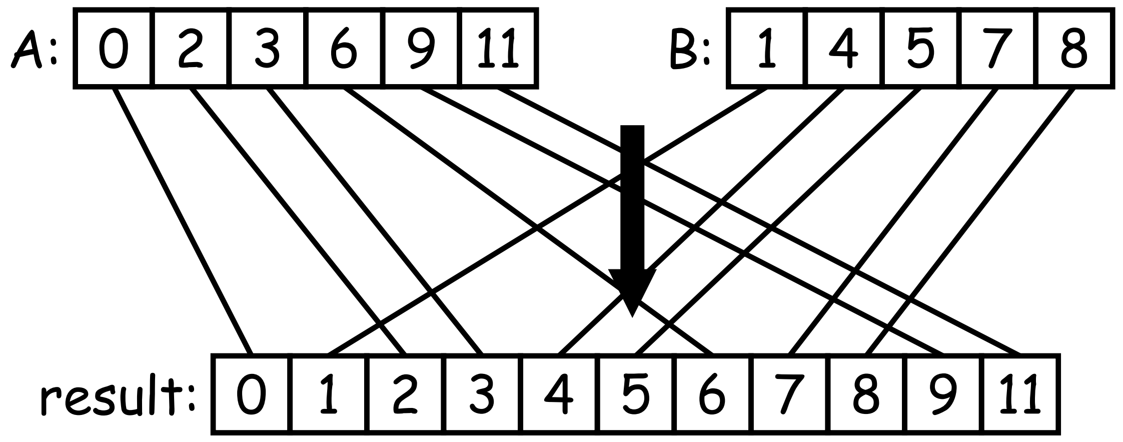


```
/** Return array, r, where r.length = ARR.length+1; r[0..K-1]
 *  the same as ARR[0..K-1], r[k] = x, r[K+1..] same as ARR[K..]. */
static String[] insert2(String[] arr, int k, String x) {
  String[] result = new String[arr.length + 1];
  arraycopy(arr, 0, result, 0, k);
  arraycopy(arr, k, result, k+1, arr.length-k);
  result[k] = x;
  return result;
}
```

Why do we need a different return type from `insert2`??

Example: Merging

Problem: Given two sorted arrays of ints, A and B, produce their *merge:* a sorted array containing all items from A and B.



Example: Merging Program

Problem: Given two sorted arrays of ints, A and B, produce their *merge:* a sorted array containing all from A and B.

Remark: In order to solve this recursively, it is useful to *generalize* the original function to allow merging *portions* of the arrays.

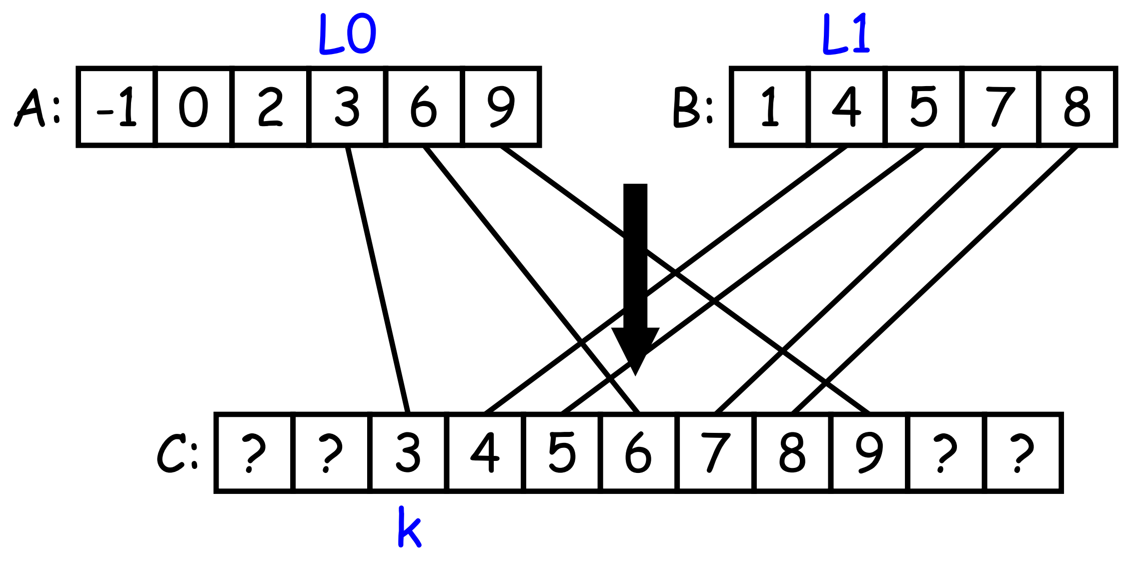
```
/** Assuming A and B are sorted, returns their merge. */
public static int[] merge(int[] A, int[] B) {
   return mergeTo(A, 0, B, 0);
}

/** The merge of A[L0..] and B[L1..] assuming A and B sorted. */
static int[] mergeTo(int[] A, int L0, int[] B, int L1) {
   int N = A.length - L0 + B.length - L1; int[] C = new int[N];
   if (L0 >= A.length) arraycopy(B, L1, C, 0, N);
   else if (L1 >= B.length) arraycopy(A, L0, C, 0, N);
   else if (A[L0] <= B[L1]) {
      C[0] = A[L0]; arraycopy(mergeTo(A, L0+1, B, L1), 0, C, 1, N-1);
   } else {
      C[0] = B[L1]; arraycopy(mergeTo(A, L0, B, L1+1), 0, C, 1, N-1);
   }
   return C;
}
```

What is wrong with this implementation?

A Tail-Recursive Strategy

```
public static int[] merge(int[] A, int[] B) {
   return mergeTo(A, 0, B, 0, new int[A.length+B.length], 0);
}

/** Merge A[L0..] and B[L1..] into C[K..], assuming A and B sorted. */
static int[] mergeTo(int[] A, int L0, int[] B, int L1, int[] C, int k){
   ...
}
```

This last method merges *part* of `A` with part of `B` into part of `C`. For example, consider a possible call `mergeTo(A, 3, B, 1, C, 2)`



A Tail-Recursive Solution

```
public static int[] merge(int[] A, int[] B) {
   return mergeTo(A, 0, B, 0, new int[A.length+B.length], 0);
}

/** Merge A[L0..] and B[L1..] into C[K..], assuming A and B sorted. */
static int[] mergeTo(int[] A, int L0, int[] B, int L1, int[] C, int k){
   if (??) {
      return C;
   } else if (??) {
      C[k] = A[L0];
      return mergeTo(A, ??, B, ??, C, ??)
   } else {
      C[k] = B[L1];
      return mergeTo(A, ??, B, ??, C, ??)
   }
}
```

A Tail-Recursive Solution

```
public static int[] merge(int[] A, int[] B) {
   return mergeTo(A, 0, B, 0, new int[A.length+B.length], 0);
}

/** Merge A[L0..] and B[L1..] into C[K..], assuming A and B sorted. */
static int[] mergeTo(int[] A, int L0, int[] B, int L1, int[] C, int k){
   if (L0 >= A.length && L1 >= B.length) {
      return C;
   } else if (??) {
      C[k] = A[L0];
      return mergeTo(A, ??, B, ??, C, ??)
   } else {
      C[k] = B[L1];
      return mergeTo(A, ??, B, ??, C, ??)
   }
}
```

A Tail-Recursive Solution

```
public static int[] merge(int[] A, int[] B) {
   return mergeTo(A, 0, B, 0, new int[A.length+B.length], 0);
}

/** Merge A[L0..] and B[L1..] into C[K..], assuming A and B sorted. */
static int[] mergeTo(int[] A, int L0, int[] B, int L1, int[] C, int k){
   if (L0 >= A.length && L1 >= B.length) {
      return C;
   } else if (L1 >= B.length || (L0 < A.length && A[L0] <= B[L1])) {
      C[k] = A[L0];
      return mergeTo(A, ??, B, ??, C, ??)
   } else {
      C[k] = B[L1];
      return mergeTo(A, ??, B, ??, C, ??)
   }
}
```

A Tail-Recursive Solution

```
public static int[] merge(int[] A, int[] B) {
   return mergeTo(A, 0, B, 0, new int[A.length+B.length], 0);
}

/** Merge A[L0..] and B[L1..] into C[K..], assuming A and B sorted. */
static int[] mergeTo(int[] A, int L0, int[] B, int L1, int[] C, int k){
   if (L0 >= A.length && L1 >= B.length) {
      return C;
   } else if (L1 >= B.length || (L0 < A.length && A[L0] <= B[L1])) {
      C[k] = A[L0];
      return mergeTo(A, L0 + 1, B, L1, C, k + 1);
   } else {
      C[k] = B[L1];
      return mergeTo(A, ??, B, ??, C, ??)
   }
}
```

A Tail-Recursive Solution

```
public static int[] merge(int[] A, int[] B) {
   return mergeTo(A, 0, B, 0, new int[A.length+B.length], 0);
}

/** Merge A[L0..] and B[L1..] into C[K..], assuming A and B sorted. */
static int[] mergeTo(int[] A, int L0, int[] B, int L1, int[] C, int k){
   if (L0 >= A.length && L1 >= B.length) {
      return C;
   } else if (L1 >= B.length || (L0 < A.length && A[L0] <= B[L1])) {
      C[k] = A[L0];
      return mergeTo(A, L0 + 1, B, L1, C, k + 1);
   } else {
      C[k] = B[L1];
      return mergeTo(A, L0, B, L1 + 1, C, k + 1);
   }
}
```

Iterative Solution

In general, we don't use either of the previous approaches in languages like C and Java. Array manipulation is most often iterative:

```
public static int[] merge(int[] A, int[] B) {
   int[] C = new int[A.length + B.length];
   // mergeTo(A, 0, B, 0, C, 0)
   int L0, L1, k;
   L0 = L1 = k = 0;

   while (??) {
       if (L1 >= B.length || (L0 < A.length && A[L0] <= B[L1])) {
           C[k] = A[L0];
           ??
       } else {
           C[k] = B[L1];
           ??
       }
   }
   return C;
}
```

Iterative Solution

In general, we don't use either of the previous approaches in languages like C and Java. Array manipulation is most often iterative:

```
public static int[] merge(int[] A, int[] B) {
   int[] C = new int[A.length + B.length];
   // mergeTo(A, 0, B, 0, C, 0)
   int L0, L1, k;
   L0 = L1 = k = 0;

   while (L0 < A.length || L1 < B.length) {
       if (L1 >= B.length || (L0 < A.length && A[L0] <= B[L1])) {
           C[k] = A[L0];
           ??
       } else {
           C[k] = B[L1];
           ??
       }
   }
   return C;
}
```

Iterative Solution

In general, we don't use either of the previous approaches in languages like C and Java. Array manipulation is most often iterative:

```
public static int[] merge(int[] A, int[] B) {
   int[] C = new int[A.length + B.length];
   // mergeTo(A, 0, B, 0, C, 0)
   int L0, L1, k;
   L0 = L1 = k = 0;

   while (L0 < A.length || L1 < B.length) {
       if (L1 >= B.length || (L0 < A.length && A[L0] <= B[L1])) {
           C[k] = A[L0];
           L0 += 1; k += 1;
       } else {
           C[k] = B[L1];
           L1 += 1; k += 1;
       }
   }
   return C;
}
```

Iterative Solution II

The same, with a **for** loop:

```
public static int[] merge(int[] A, int[] B) {
   int[] C = new int[A.length + B.length];
   int L0, L1;
   L0 = L1 = 0;
   for (int k = 0; k < C.length; k += 1) {
       if (L1 >= B.length || (L0 < A.length && A[L0] <= B[L1])) {
           C[k] = A[L0]; L0 += 1;
       } else {
           C[k] = B[L1]; L1 += 1;
       }
   }
   return C;
}
```

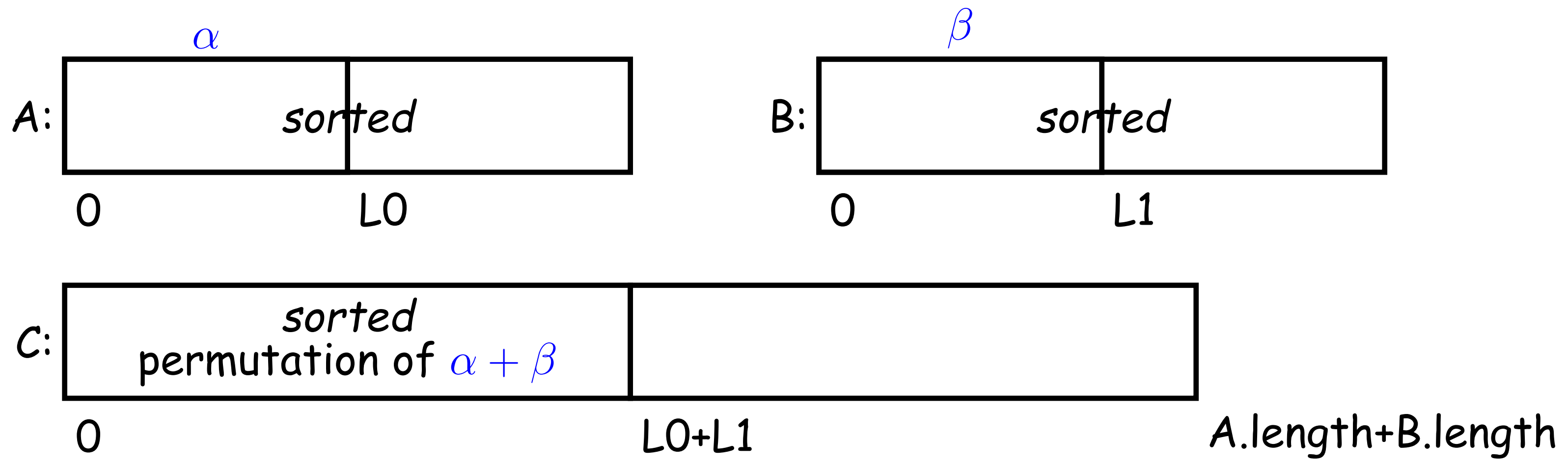
Invariant (true after `int k = 0`):

$0 \leq L0 < A.length \ \wedge \ 0 \leq L1 < B.length \wedge \ C.length = A.length + B.length \ \wedge \ k = L0 + L1$
$\wedge \ C[0:k]$ is a permutation of A[0:L0] + B[0:L1]
$\wedge \ C[0:k], A, B$ are sorted.

Alternative Solution: Removing k

Using previous invariant that k=L0+L1 simplifies things:

```
public static int[] merge(int[] A, int[] B) {
   int[] C = new int[A.length + B.length];
   int L0, L1;  L0 = L1 = 0;
   while (L0 + L1 < C.length) {
       if (L1 >= B.length || (L0 < A.length && A[L0] < B[L1])) {
           C[L0 + L1] = A[L0]; L0 += 1;
       } else {
           C[L0 + L1] = B[L1]; L1 += 1;
       }
   }
   return C;
}
```



Multidimensional Arrays

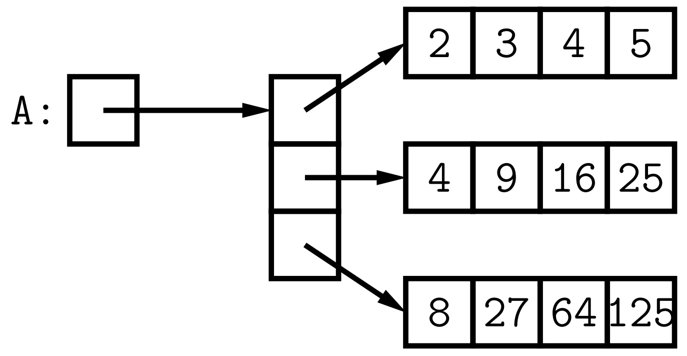
What about two- or higher-dimensional layouts, such as

$$A = \begin{array}{|r|r|r|r|}\hline 2 & 3 & 4 & 5 \\ \hline 4 & 9 & 16 & 25 \\ \hline 8 & 27 & 64 & 125 \\ \hline \end{array} \quad \textbf{?}$$

Multidimensional Arrays in Java

These are not primitive in Java, but we can build them as arrays of arrays:

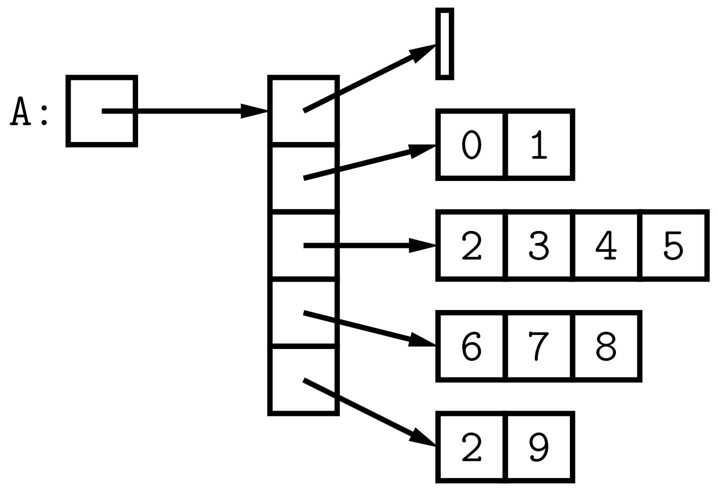
```
    int[][] A = new int[3][];
    A[0] = new int[] {2, 3, 4, 5};
    A[1] = new int[] {4, 9, 16, 25};
    A[2] = new int[] {8, 27, 64, 125};
  // or
    int[][] A;
    A = new int[][] { {2, 3, 4, 5},
                      {4, 9, 16, 25},
                      { 8, 27, 64, 125} };
  // or
    int[][] A = { {2, 3, 4, 5},
                  {4, 9, 16, 25},
                  {8, 27, 64, 125} };
  // or
    int[][] A = new A[3][4];
    for (int i = 0; i < 3; i += 1)
        for (int j = 0; j < 4; j += 1)
            A[i][j] = (int) Math.pow(j + 2, i + 1);
```



Exotic Multidimensional Arrays

- Since every element of an array is independent, there is no single "width" in general:

```
int[][] A = new int[5][];
A[0] = new int[] {};
A[1] = new int[] {0, 1};
A[2] = new int[] {2, 3, 4, 5};
A[3] = new int[] {6, 7, 8};
A[4] = new int[] {9};
```



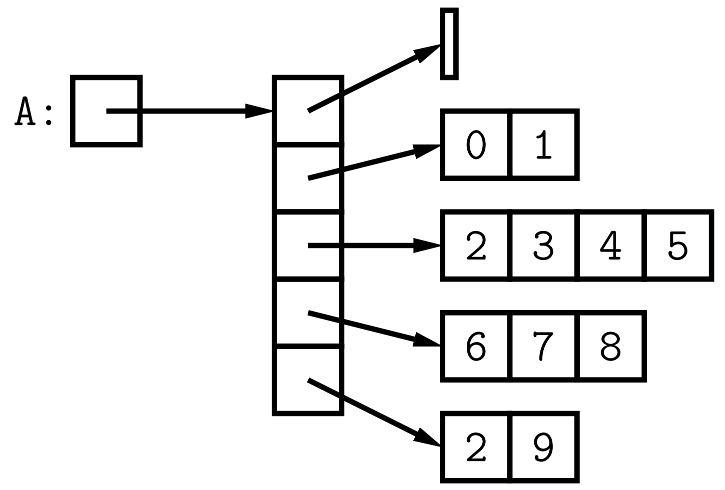
- What does this print?

```
int[][] ZERO = new int[3][];
ZERO[0] = ZERO[1] = ZERO[2] =
    new int[] {0, 0, 0};
ZERO[0][1] = 1;
System.out.println(ZERO[2][1]);
```

Exotic Multidimensional Arrays

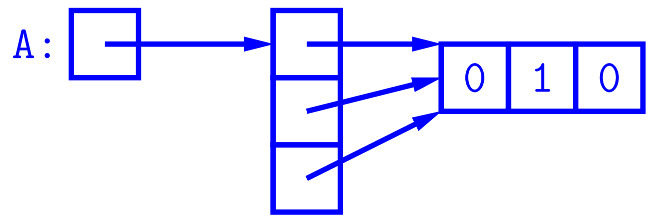
- Since every element of an array is independent, there is no single "width" in general:

```
int[][] A = new int[5][];
A[0] = new int[] {};
A[1] = new int[] {0, 1};
A[2] = new int[] {2, 3, 4, 5};
A[3] = new int[] {6, 7, 8};
A[4] = new int[] {9};
```



- What does this print?

```
int[][] ZERO = new int[3][];
ZERO[0] = ZERO[1] = ZERO[2] =
    new int[] {0, 0, 0};
ZERO[0][1] = 1;
System.out.println(ZERO[2][1]);
```



CS61B Lecture #6: More Iteration: Sort an Array

Problem. Print out the command-line arguments in lexicographic order:

```
% java sort the quick brown fox jumped over the lazy dog
brown dog fox jumped lazy over quick the the
```

Plan.

```
public class Sort {
  /** Sort and print WORDS lexicographically. */
  public static void main(String[] words) {
    sort(words, 0, words.length-1);
    print(words);
  }

  /** Sort items A[L..U], with all others unchanged. */
  static void sort(String[] A, int L, int U) { /* "TOMORROW" */ }

  /** Print A on one line, separated by blanks. */
  static void print(String[] A) { /* "TOMORROW" */ }
}
```

How do We Know If It Works?

- *Unit testing* refers to the testing of individual units (methods, classes) within a program, rather than the whole program.
- In this class, we mainly use the JUnit tool for unit testing.
- Example: `AGTestYear.java` in lab #1.
- *Integration testing* refers to the testing of entire (integrated) set of modules—the whole program.
- In this course, we'll look at various ways to run the program against prepared inputs and checking the output.
- *Regression testing* refers to testing with the specific goal of checking that fixes, enhancements, or other changes have not introduced faults (regressions).

Test-Driven Development

- Idea: write tests first.
- Implement unit at a time, run tests, fix and refactor until it works.
- We're not really going to push it in this course, but it is useful and has quite a following.

Testing sort

- This is pretty easy: just give a bunch of arrays to `sort` and then make sure they each get sorted properly.
- Have to make sure we cover the necessary cases:
  - *Corner cases.* E.g., empty array, one-element, all elements the same.
  - *Representative "middle" cases.* E.g., elements reversed, elements in order, one pair of elements reversed, ...

Simple JUnit

- The JUnit package provides some handy tools for unit testing.
- The Java annotation `@Test` on a method tells the JUnit machinery to call that method.
- (An *annotation* in Java provides information about a method, class, etc., that can be examined within Java itself.)
- A collection of methods with names beginning with `assert` then allow your test cases to check conditions and report failures.
- [See example.]

Selection Sort

```
/** Sort items A[L..U], with all others unchanged. */
static void sort(String[] A, int L, int U) {
  if (L < U) {
    int k = /*( Index s.t. A[k] is largest in A[L],...,A[U] )*/;
    /*{ swap A[k] with A[U] }*/;
    /*{ Sort items L to U-1 of A. }*/;
  }
}
```

And we're done! Well, OK, not quite.

Selection Sort

```
/** Sort items A[L..U], with all others unchanged. */
static void sort(String[] A, int L, int U) {
  if (L < U) {
    int k = indexOfLargest(A, L, U);
    /*{ swap A[k] with A[U] }*/;
    /*{ Sort items L to U-1 of A. }*/;
  }
}

/** Index k, I0<=k<=I1, such that V[k] is largest element among
 *  V[I0], ... V[I1]. Requires I0<=I1. */
static int indexOfLargest(String[] V, int i0, int i1) {
    ...
}
```

Selection Sort

```
/** Sort items A[L..U], with all others unchanged. */
static void sort(String[] A, int L, int U) {
  if (L < U) {
    int k = indexOfLargest(A, L, U);
    /*{ swap A[k] with A[U] }*/;
    sort(A, L, U-1);       // Sort items L to U-1 of A
  }
}

/** Index k, I0<=k<=I1, such that V[k] is largest element among
 *  V[I0], ... V[I1]. Requires I0<=I1. */
static int indexOfLargest(String[] V, int i0, int i1) {
    ...
}
```

Selection Sort

```
/** Sort items A[L..U], with all others unchanged. */
static void sort(String[] A, int L, int U) {
  if (L < U) {
    int k = indexOfLargest(A, L, U);
    String tmp = A[k];  A[k] = A[U]; A[U] = tmp;
    sort(A, L, U-1);      // Sort items L to U-1 of A
  }
}

/** Index k, I0<=k<=I1, such that V[k] is largest element among
 *  V[I0], ... V[I1]. Requires I0<=I1. */
static int indexOfLargest(String[] V, int i0, int i1) {
    ...
}
```

Selection Sort

```
/** Sort items A[L..U], with all others unchanged. */
static void sort(String[] A, int L, int U) {
  if (L < U) {
    int k = indexOfLargest(A, L, U);
    String tmp = A[k];  A[k] = A[U]; A[U] = tmp;
    sort(A, L, U-1);       // Sort items L to U-1 of A
  }
}
```

What would an iterative version look like?

```
  while (?) {
    ?
  }
```

Selection Sort

```
/** Sort items A[L..U], with all others unchanged. */
static void sort(String[] A, int L, int U) {
  if (L < U) {
    int k = indexOfLargest(A, L, U);
    String tmp = A[k];  A[k] = A[U]; A[U] = tmp;
    sort(A, L, U-1);      // Sort items L to U-1 of A
  }
}
```

Iterative version:

```
  while (L < U) {
    int k = indexOfLargest(A, L, U);
    String tmp = A[k];  A[k] = A[U]; A[U] = tmp;
    U -= 1;
  }
```

Find Largest

```
/** Index k, I0<=k<=I1, such that V[k] is largest element among
 *  V[I0], ... V[I1]. Requires I0<=I1. */
static int indexOfLargest(String[] V, int i0, int i1) {
  if (?)
    return i1;
  else  {


  }
}
```

Find Largest

```
/** Index k, I0<=k<=I1, such that V[k] is largest element among
 *  V[I0], ... V[I1]. Requires I0<=I1. */
static int indexOfLargest(String[] V, int i0, int i1) {
  if (i0 >= i1)
    return i1;
  else /* if (i0 < i1) */ {


  }
}
```

Find Largest

```
/** Index k, I0<=k<=I1, such that V[k] is largest element among
 *  V[I0], ... V[I1]. Requires I0<=I1. */
static int indexOfLargest(String[] V, int i0, int i1) {
  if (i0 >= i1)
    return i1;
  else /* if (i0 < i1) */ {
    int k = /*( index of largest value in V[i0 + 1..i1] )*/;
    return /*( whichever of i0 and k has larger value )*/;
  }
}
```

Find Largest

```
/** Index k, I0<=k<=I1, such that V[k] is largest element among
 *  V[I0], ... V[I1]. Requires I0<=I1. */
static int indexOfLargest(String[] V, int i0, int i1) {
  if (i0 >= i1)
    return i1;
  else /* if (i0 < i1) */ {
    int k = indexOfLargest(V, i0 + 1, i1);
    return /*( whichever of i0 and k has larger value )*/;
  }
}
```

Find Largest

```
/** Index k, I0<=k<=I1, such that V[k] is largest element among
 *  V[I0], ... V[I1]. Requires I0<=I1. */
static int indexOfLargest(String[] V, int i0, int i1) {
  if (i0 >= i1)
    return i1;
  else /* if (i0 < i1) */ {
    int k = indexOfLargest(V, i0 + 1, i1);
    return (V[i0].compareTo(V[k]) > 0) ? i0 : k;
    // if (V[i0].compareTo(V[k]) > 0) return i0; else return k;
  }
}
```

- Turning this into an iterative version is tricky: not tail recursive.
- What are the arguments to `compareTo` the first time it's called?

Iteratively Find Largest

```
/** Value k, I0<=k<=I1, such that V[k] is largest element among
 *  V[I0], ... V[I1]. Requires I0<=I1. */
static int indexOfLargest(String[] V, int i0, int i1) {
  if (i0 >= i1)
    return i1;
  else /* if (i0 < i1) */ {
    int k = indexOfLargest(V, i0 + 1, i1);
    return (V[i0].compareTo(V[k]) > 0) ? i0 : k;
    // if (V[i0].compareTo(V[k]) > 0) return i0; else return k;
  }
}
```

Iterative:

```
  int i, k;
  k = ?;     // Deepest iteration
  for (i = ?; ...?; i ...?)
    k = ?;
  return k;
```

Iteratively Find Largest

```
/** Value k, I0<=k<=I1, such that V[k] is largest element among
 *  V[I0], ... V[I1]. Requires I0<=I1. */
static int indexOfLargest(String[] V, int i0, int i1) {
  if (i0 >= i1)
    return i1;
  else /* if (i0 < i1) */ {
    int k = indexOfLargest(V, i0 + 1, i1);
    return (V[i0].compareTo(V[k]) > 0) ? i0 : k;
    // if (V[i0].compareTo(V[k]) > 0) return i0; else return k;
  }
}
```

Iterative:

```
  int i, k;
  k = i1;     // Deepest iteration
  for (i = ?; ...?; i ...?)
    k = ?;
  return k;
```

Iteratively Find Largest

```
/** Value k, I0<=k<=I1, such that V[k] is largest element among
 *  V[I0], ... V[I1]. Requires I0<=I1. */
static int indexOfLargest(String[] V, int i0, int i1) {
  if (i0 >= i1)
    return i1;
  else /* if (i0 < i1) */ {
    int k = indexOfLargest(V, i0 + 1, i1);
    return (V[i0].compareTo(V[k]) > 0) ? i0 : k;
    // if (V[i0].compareTo(V[k]) > 0) return i0; else return k;
  }
}
```

Iterative:

```
  int i, k;
  k = i1;     // Deepest iteration
  for (i = i1 - 1; i >= i0; i -= 1)
    k = ?;
  return k;
```

Iteratively Find Largest

```
/** Value k, I0<=k<=I1, such that V[k] is largest element among
 *  V[I0], ... V[I1]. Requires I0<=I1. */
static int indexOfLargest(String[] V, int i0, int i1) {
  if (i0 >= i1)
    return i1;
  else /* if (i0 < i1) */ {
    int k = indexOfLargest(V, i0 + 1, i1);
    return (V[i0].compareTo(V[k]) > 0) ? i0 : k;
    // if (V[i0].compareTo(V[k]) > 0) return i0; else return k;
  }
}
```

Iterative:

```
  int i, k;
  k = i1;     // Deepest iteration
  for (i = i1 - 1; i >= i0; i -= 1)
    k = (V[i].compareTo(V[k]) > 0) ? i : k;
  return k;
```

Finally, Printing

```
/** Print A on one line, separated by blanks. */
static void print(String[] A) {
  for (int i = 0; i < A.length; i += 1)
    System.out.print(A[i] + " ");
  System.out.println();
}

/* Java also provides a simple, specialized syntax for looping
 * through an entire array: */
  for (String s : A)
    System.out.print(s + " ");
```

Another Problem

Given an array of integers, `A`, of length $N > 0$, find the smallest index, k, such that all elements at indices $\geq k$ and $< N-1$ are greater than `A[`$N-1$`]`. Then rotate elements k to $N-1$ right by one. For example, if `A` starts out as

```
{ 1, 9, 4, 3, 0, 12, 11, 9, 15, 22, 12 }
```

then it ends up as

```
{ 1, 9, 4, 3, 0, 12, 11, 9, 12, 15, 22 }
```

As another example,

```
{ 1, 9, 4, 3, 0, 12, 11, 9, 15, 22, -2 }
```

would become

```
{ -2, 1, 9, 4, 3, 0, 12, 11, 9, 15, 22 }
```

What if A starts like this?

```
{ 1, 9, 4, 3, 0, 12, 11, 9, 12, 15, 22 }
```

Another Problem

Given an array of integers, `A`, of length $N > 0$, find the smallest index, k, such that all elements at indices $\geq k$ and $< N - 1$ are greater than `A`$[N-1]$. Then rotate elements k to $N-1$ right by one. For example, if `A` starts out as

```
{ 1, 9, 4, 3, 0, 12, 11, 9, 15, 22, 12 }
```

then it ends up as

```
{ 1, 9, 4, 3, 0, 12, 11, 9, 12, 15, 22 }
```

As another example,

```
{ 1, 9, 4, 3, 0, 12, 11, 9, 15, 22, -2 }
```

would become

```
{ -2, 1, 9, 4, 3, 0, 12, 11, 9, 15, 22 }
```

What if A starts like this?

```
{ 1, 9, 4, 3, 0, 12, 11, 9, 12, 15, 22 }
```

Answer: It's unchanged. (No, the spec is not ambiguous.)

Your turn

```
public class Shove {

    /** Rotate elements A[k] to A[A.length-1] one element to the
     *  right, where k is the smallest index such that elements
     *  k through A.length-2 are all larger than A[A.length-1].
     */
    static void moveOver(int[] A) {
       // FILL IN
    }

}
```

Recreation

Given that

$$\log(1+x) = x - \frac{1}{2}x^2 + \frac{1}{3}x^3 - \ldots$$

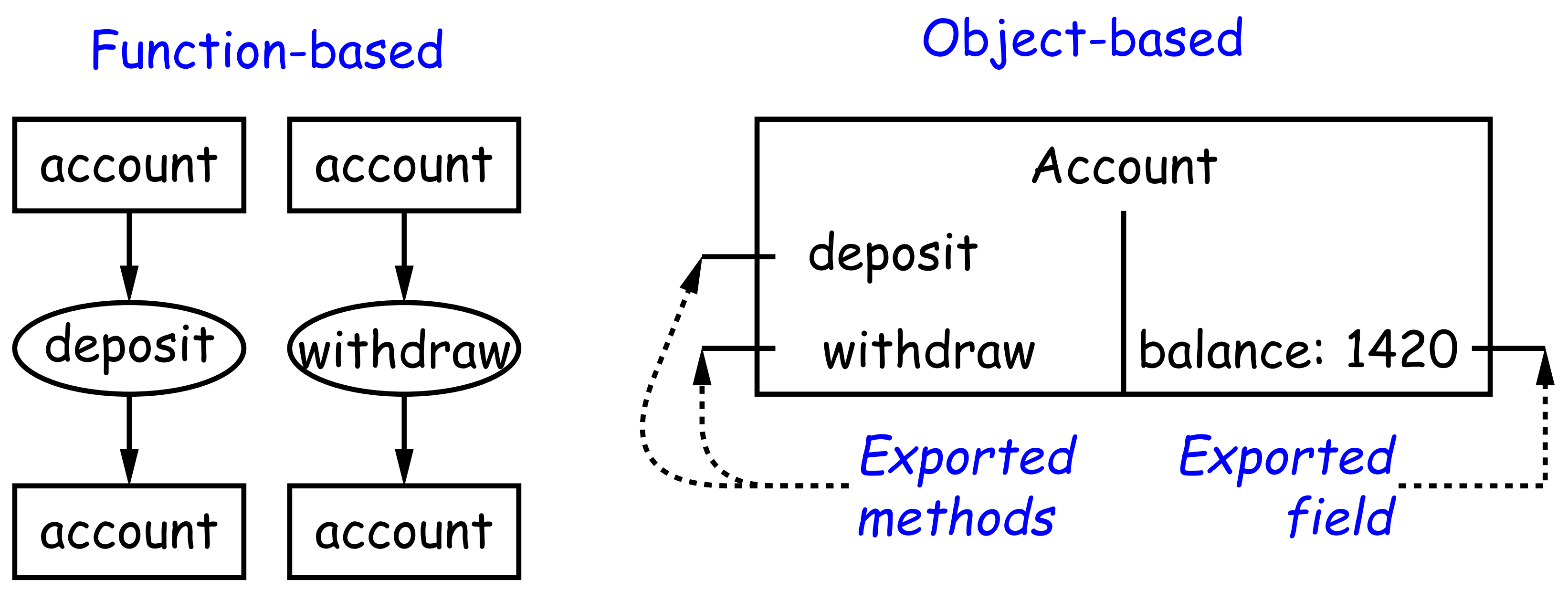
why is it not the case that

$$\begin{aligned}
\log 2 &= 1 - 1/2 + 1/3 - 1/4 + 1/5 - 1/6 + 1/7 - 1/8 + 1/9 - \ldots \\
&= (1 + 1/3 + 1/5 + 1/7 + 1/9 + \ldots) - (1/2 + 1/4 + 1/6 + 1/8 + \ldots) \\
&= (1 + 1/3 + 1/5 + 1/7 + 1/9 + \ldots) + (1/2 + 1/4 + 1/6 + 1/8 + \ldots) \\
&\quad -2(1/2 + 1/4 + 1/6 + 1/8 + \ldots) \\
&= (1 + 1/2 + 1/3 + 1/4 + \ldots) - (1 + 1/2 + 1/3 + 1/4 + \ldots) \\
&= 0?
\end{aligned}$$

CS61B Lecture #7: Object-Based Programming

Basic Idea.

- *Function-based programs* are organized primarily around the functions (methods, etc.) that do things. Data structures (objects) are considered separate.
- *Object-based programs* are organized around the *types of objects* that are used to represent data; methods are grouped by type of object.
- Simple banking-system example:



Philosophy

- Idea (from 1970s and before): An *abstract data type* is
  - a set of possible values (a *domain*), plus
  - a set of *operations* on those values (or their containers).
- In `IntList`, for example, the domain was a *set of pairs:* `(head,tail)`, where `head` is an `int` and `tail` is a pointer to an `IntList`.
- The `IntList` operations consisted only of assigning to and accessing the two fields (`head` and `tail`).
- In general, we prefer a purely *procedural interface,* where the functions (methods) do everything—no outside access to the internal representation (i.e., instance variables).
- That way, implementor of a class and its methods has complete control over behavior of instances.
- In Java, the preferred way to write the "operations of a type" is as *instance methods.*

You Saw It All (Maybe) in CS61A: The Account Class

```python
class Account:
    balance = 0
    def __init__(self, balance0):
        self.balance = balance0

    def deposit(self, amount):
        self.balance += amount
        return self.balance

    def withdraw(self, amount):
        if self.balance < amount:
            raise ValueError \
                ("Insufficient funds")
        else:
            self.balance -= amount
        return self.balance
```

```python
myAccount = Account(1000)
print(myAccount.balance)
myAccount.deposit(100)
myAccount.withdraw(500)
```

```java
public class Account {
  public int balance;
  public Account(int balance0) {
    this.balance = balance0;
  }
  public int deposit(int amount) {
    balance += amount; return balance;
  }
  public int withdraw(int amount) {
    if (balance < amount)
      throw new IllegalStateException
         ("Insufficient funds");
    else balance -= amount;
    return balance;
  }
}
```

```java
Account myAccount = new Account(1000);
print(myAccount.balance)
myAccount.deposit(100);
myAccount.withdraw(500);
```

You Also Saw It All in CS61AS

```
(define-class (account balance0)
  (instance-vars (balance 0))
  (initialize
    (set! balance balance0))

  (method (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (method (withdraw amount)
    (if (< balance amount)
      (error "Insufficient funds")
      (begin
        (set! balance (- balance amount))
        balance))) )
```

```
(define my-account
  (instantiate account 1000))
(ask my-account 'balance)
(ask my-account 'deposit 100)
(ask my-account 'withdraw 500)
```

```
public class Account {
  public int balance;
  public Account(int balance0) {
    balance = balance0;
  }
  public int deposit(int amount) {
    balance += amount; return balance;
  }
  public int withdraw(int amount) {
    if (balance < amount)
      throw new IllegalStateException
          ("Insufficient funds");
    else balance -= amount;
    return balance;
  }
}
```

```
Account myAccount = new Account(1000);
myAccount.balance
myAccount.deposit(100);
myAccount.withdraw(500);
```

The Pieces

- **Class declaration** defines a *new type of object,* i.e., new type of structured container.
- **Instance variables** such as `balance` are the simple containers within these objects (*fields* or *components*).
- **Instance methods**, such as `deposit` and `withdraw` are like ordinary (static) methods that take an invisible extra parameter (called **this**).
- The **new** operator creates (*instantiates*) new objects, and initializes them using constructors.
- **Constructors** such as the method-like declaration of `Account` are special methods that are used only to initialize new instances. They take their arguments from the **new** expression.
- **Method selection** picks methods to call. For example,

  ```
  myAccount.deposit(100)
  ```

  tells us to call the method named `deposit` that is defined for the object pointed to by `myAccount`.

Getter Methods

- Slight problem with Java version of `Account`: anyone can assign to the `balance` field
- This reduces the control that the implementor of Account has over possible values of the balance.
- Solution: allow public access only through methods:

```
public class Account {
  private int _balance;
  ...
  public int balance() { return _balance; }
  ...
}
```

- Now `Account._balance = 1000000` is an error outside `Account`.
- (I use the convention of putting '_' at the start of private instance variables to distinguish them from local variables and non-private variables. Could actually use `balance` for both the method and the variable, but please don't.)

Class Variables and Methods

- Suppose we want to keep track of the bank's total funds.
- This number is not associated with any particular Account, but is common to all—it is *class-wide.* In Java, "class-wide" ≡ `static`.

```
public class Account {
  ...
  private static int _funds = 0;
  public int deposit(int amount) {
    _balance += amount;
    _funds += amount;      // or this._funds or Account._funds
    return _balance;
  }
  public static int funds() {
    return _funds;         // or Account._funds
  }
  ...  // Also change withdraw.
}
```

- From outside, can refer to either `Account.funds()` or to `myAccount.funds()` (same thing).

Instance Methods

- Instance method such as

```
int deposit(int amount) {
  _balance += amount;
  _funds += amount;
  return balance;
}
```

behaves sort of like a static method with hidden argument:

```
static int deposit(final Account this, int amount) {
  this._balance += amount;
  _funds += amount;
  return this._balance;
}
```

- NOTE: Just explanatory: Not real Java (not allowed to declare 'this'). (`final` *is* real Java; means "can't change once initialized.")

Calling Instance Method

```
/** (Fictional) equivalent of deposit instance method. */
static int deposit(final Account this, int amount) {
  this._balance += amount;
  _funds += amount;
  return this._balance;
}
```

- Likewise, the instance-method call `myAccount.deposit(100)` is like a call on this fictional static method:

```
        Account.deposit(myAccount, 100);
```

- Inside a real instance method, as a convenient abbreviation, one can leave off the leading '`this.`' on field access or method call if not ambiguous. (Unlike Python)

‘Instance’ and ‘Static’ Don’t Mix

- Since real static methods don’t have the invisible `this` parameter, makes no sense to refer directly to instance variables in them:

```
public static int badBalance(Account A) {
    int x = A._balance;   // This is OK
                          // (A tells us whose balance)
    return _balance;      // WRONG! NONSENSE!
}
```

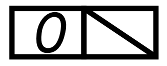
- Reference to `_balance` here equivalent to `this._balance`,
- But this is meaningless (*whose* balance?)
- However, it makes perfect sense to access a static (class-wide) field or method in an instance method or constructor, as happened with `_funds` in the `deposit` method.
- There’s only one of each static field, so don’t need to have a ‘this’ to get it. Can just name the class (or use no qualification inside the class, as we’be been doing).

Constructors

- To completely control objects of some class, you must be able to set their initial contents.
- A *constructor* is a kind of special instance method that is called by the **new** operator right after it creates a new object, as if

L = new IntList(1,null) $\Longrightarrow$ {

```
tmp = pointer to
```



```
tmp.IntList(1, null);
L = tmp;
```

Multiple Constructors and Default Constructors

- **All** classes have constructors. In the absence of any explicit constructor, get **default constructor**, as if you had written:

```
public class Foo {
    public Foo() {   }
}
```

- Multiple *overloaded* constructors possible, and they can use each other (although the syntax is odd):

```
public class IntList {
    public IntList(int head, IntList tail) {
        this.head = head; this.tail = tail;
    }

    public IntList(int head) {
        this(head, null);    // Calls first constructor.
    }
    ...
}
```

Constructors and Instance Variables

- Instance variables initializations are moved inside constructors that don't start with `this(...)`.

```
class Foo {
   int x = 5;

   Foo(int y) {
      DoStuff(y);
   }

   Foo() {
      this(42);
   }
}
```

$\Longleftrightarrow$

```
class Foo {
   int x;

   Foo(int y) {
      x = 5;
      DoStuff(y);
   }

   Foo() {
      this(42); // Assigns to x
   }
}
```

Summary: Java vs. Python

Java	Python
`class Foo {`	`class Foo: ...`
`    int x = ...;`	`    x = ...`
`    Foo(...)`	`    def __init__(self, ...):`
`       { ... }`	`        ...`
`    int f(...)`	`    def f(self, ...):`
`       {...}`	`        ...`
`    static int y = 21;`	`    y = 21    # Referred to as Foo.y`
`    static void g(...)`	`    @staticmethod`
`       {...}`	`    def g(...):`
`}`	`        ...`
`aFoo.f(...)`	`aFoo.f(...)`
`aFoo.x`	`aFoo.x`
`new Foo(...)`	`Foo(...)`
`this`	`self     # (typically)`

CS61B Lecture #8: Object-Oriented Mechanisms

Today:

- New in this lecture: the bare mechanics of "object-oriented programming."
- The general topic is: Writing software that operates on many kinds of data.

Overloading

Problem: How to get `System.out.print(x)` to print `x`, regardless of type of `x`?

- In Scheme or Python, one function can take an argument of any type, and then test the type (if needed).
- In Java, methods specify a single type of argument.
- Partial solution: *overloading*—multiple method definitions with the same name and different numbers or types of arguments.
- E.g., `System.out` has type `java.io.PrintStream`, which defines

```
void println()  Prints new line.
void println(String s)  Prints S.
void println(boolean b)  Prints "true" or "false"
void println(char c)  Prints single character
void println(int i)  Prints I in decimal
etc.
```

- Each of these is a different function. Compiler decides which to call on the basis of arguments' types.

Generic Data Structures

Problem: How to get a "list of anything" or "array of anything"?

- Again, no problem in Scheme or Python.
- But in Java, lists (such as `IntList`) and arrays have a single type of element.
- First, the short answer: any *reference* value can be converted to type `java.lang.Object` and back, so can use `Object` as the "generic (reference) type":

```
Object[] things = new Object[2];
things[0] = new IntList(3, null);
things[1] = "Stuff";
// Now ((IntList) things[0]).head == 3;
// and ((String) things[1]).startsWith("St") is true
// things[0].head                    Illegal
// things[1].startsWith("St")  Illegal
```

And Primitive Values?

- Primitive values (ints, longs, bytes, shorts, floats, doubles, chars, and booleans) are not really convertible to `Object`.
- Presents a problem for "list of anything."
- So Java introduced a set of *wrapper types*, one for each primitive type:

Prim.	Ref.	Prim.	Ref.	Prim.	Ref.
byte	Byte	short	Short	int	Integer
long	Long	char	Character	boolean	Boolean
float	Float	double	Double		

- One can create new wrapper objects for any value (*boxing*):

```
Integer Three = new Integer(3);
Object ThreeObj = Three;
```

and vice-versa (*unboxing*):

```
int three = Three.intValue();
```

Autoboxing

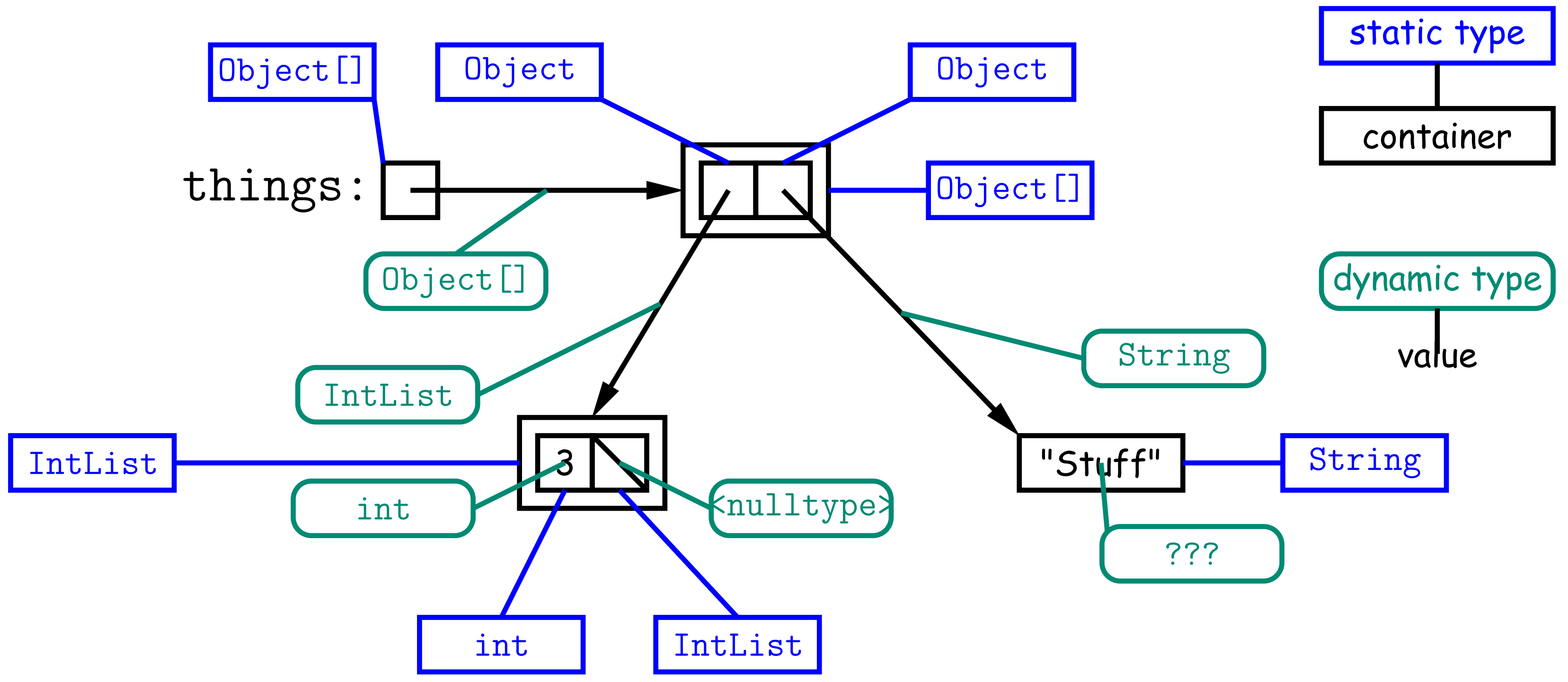
Boxing and unboxing are automatic (in many cases):

```
Integer Three = 3;
int three = Three;
int six = Three + 3;

Integer[] someInts = { 1, 2, 3 };
for (int x : someInts) {
    System.out.println(x);
}

System.out.println(someInts[0]);
    // Prints Integer 1, but NOT unboxed.
```

Dynamic vs. Static Types

- Every *value* has a type—its *dynamic type.*
- Every *container* (variable, component, parameter), literal, function call, and operator expression (e.g. x+y) has a type—its *static type*.
- Therefore, every *expression* has a static type.

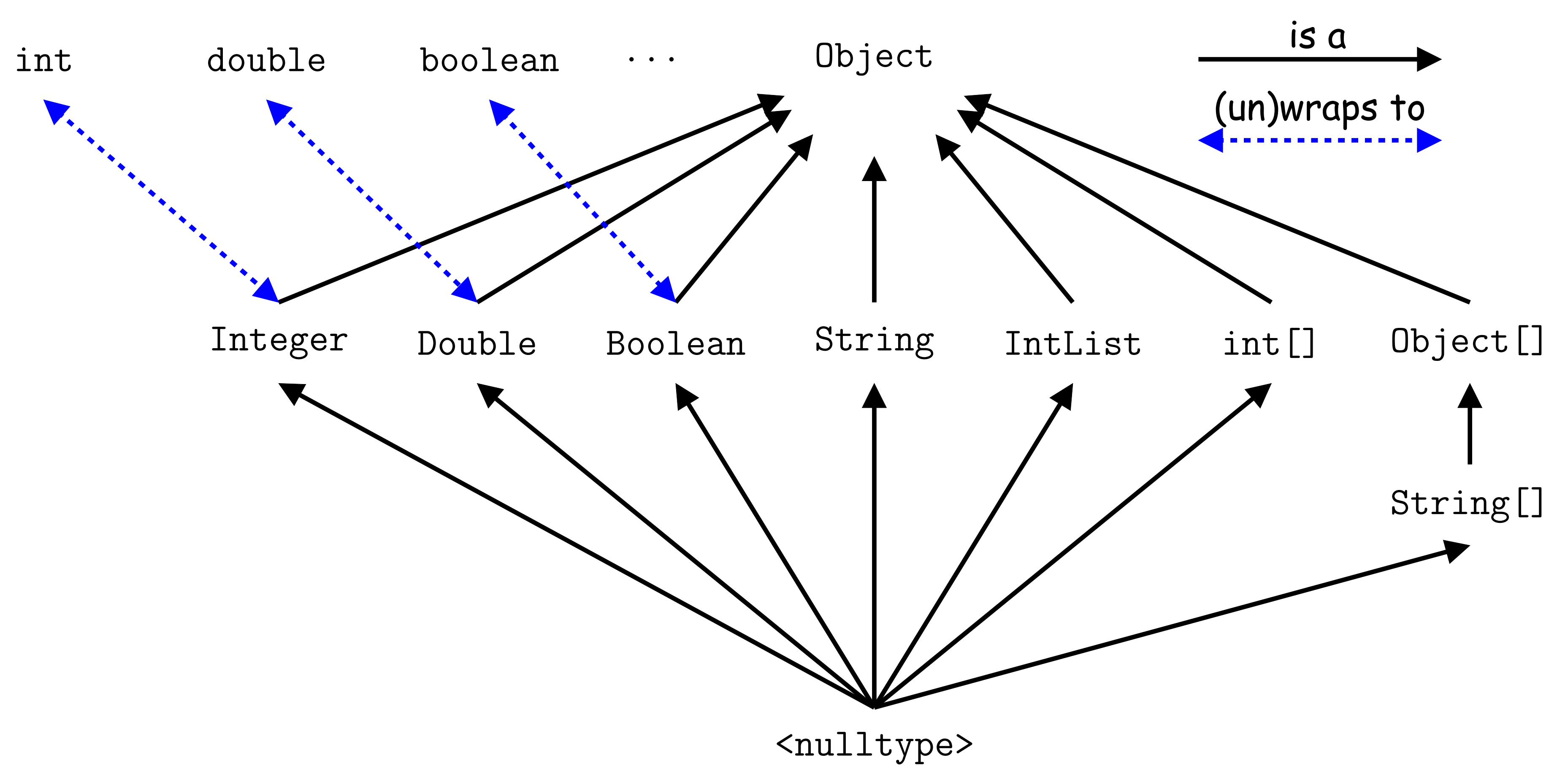
```
Object[] things = new Object[2];
things[0] = new IntList(3, null);
things[1] = "Stuff";
```



Type Hierarchies

- A container with (static) type T may contain a certain value only if that value "is a" T—that is, if the (dynamic) type of the value is a *subtype* of T. Likewise, a function with return type T may return only values that are subtypes of T.
- All types are subtypes of themselves (& that's all for primitive types)
- *Reference types* form a *type hierarchy;* some are subtypes of others. **null**'s type is a subtype of all reference types.
- All reference types are subtypes of `Object`.

Java Library Type Hierarchy (Partial)



The Basic Static Type Rule

- Java is designed so that any expression of (static) type T always yields a value that "is a" T.
- Static types are "known to the compiler," because you declare them, as in

  ```
  String x;              // Static type of field
  int f(Object s)  { // Static type of call to f, and of parameter
    int y;               // Static type of local variable
  ```

  or they are pre-declared by the language (like 3).
- Compiler insists that in an assignment, `L = E`, or function call, `f(E)`, where

  ```
      void f(SomeType L) { ... },
  ```

  E's static type must be subtype of L's static type.
- Similar rules apply to `E[i]` (static type of E must be an array) and other built-in operations.

Coercions

- The values of type `short`, for example, are a subset of those of `int` (`short`s are representable as 16-bit integers, `int`s as 32-bit integer)
- But we *don't* say that `short` is a subtype of `int`, because they don't quite behave the same.
- Instead, we say that values of type `short` can be *coerced* (converted) to a value of type `int`.
- Leads to a slight fudge: compiler will silently coerce "smaller" integer types to larger ones, `float` to `double`, and (as just seen) between primitive types and their wrapper types.
- So,

```
short x = 3002;
int y = x;
```

  works without complaint.

Consequences of Compiler's "Sanity Checks"

- This is a *conservative* rule. The last line of the following, which you might think is perfectly sensible, is illegal:

  ```
  int[] A = new int[2];
  Object x = A; // All references are Objects
  A[i] = 0;     // Static type of A is array...
  x[i+1] = 1;   // But not of x: ERROR
  ```

  Compiler figures that not every `Object` is an array.

- Q: Don't we *know* that `x` contains array value!?
- A: Yes, but still must tell the compiler, like this:

  ```
      ((int[]) x)[i+1] = 1;
  ```

- Defn: Static type of cast `(T) E` is T.
- Q: What if `x` *isn't* an array value, or is null?
- A: For that we have runtime errors—exceptions.

Overriding and Extension

- Notation so far is clumsy.
- Q: If I know `Object` variable `x` contains a `String`, why can't I write, `x.startsWith("this")`?
- A: `startsWith` is only defined on Strings, not on all `Objects`, so the compiler isn't sure it makes sense, unless you cast.
- But, if an operation *were* defined on all `Objects`, then you *wouldn't* need clumsy casting.
- Example: `.toString()` is defined on all `Object`s. You can always say `x.toString()` if `x` has a reference type.
- The default `.toString()` function is not very useful; on an `IntList`, would produce string like `"IntList@2f6684"`
- But for any subtype of Object, you may *override* the default definition.

Overriding toString

- For example, if `s` is a String, `s.toString()` is the identity function (fortunately).
- For any type you define, you may supply your own definition. For example, in `IntList`, could add

```
public String toString() {
    StringBuffer b = new StringBuffer();
    b.append("[");
    for (IntList L = this; L != null; L = L.tail)
        b.append(" " + L.head);
    b.append("]");
    return b.toString();
}
```

- If `x = new IntList(3, new IntList(4, null))`, then `x.toString()` is `"[3 4]"`.
- Conveniently, the "+" operator on Strings calls `.toString` when asked to append an Object, and so does the `"%s"` formatter for `printf`.
- With this trick, you can supply an output function for any type you define.

Extending a Class

- To say that class `B` is a direct subtype of class `A` (or `A` is a *direct superclass* of `B`), write

```
    class B extends A { ... }
```

- By default, `class ... extends java.lang.Object`.
- The subtype *inherits* all fields and methods of its direct superclass (and passes them along to any of its subtypes).
- In class B, you may *override* an instance method (*not* a static method), by providing a new definition with same *signature* (name, return type, argument types).
- I'll say that a method and all its overridings form a *dynamic method set*.
- **The Point:** If `f(...)` is an instance method, then the call `x.f(...)` calls whatever overriding of `f` applies to the *dynamic type* of `x`, *regardless* of the static type of `x`.

Illustration

```
class Worker {
  void work() {
    collectPay();
  }
}
```

```
class Prof extends Worker {
  // Inherits work()
}
```

```
class TA extends Worker {
  void work() {
    while (true) {
      doLab(); discuss(); officeHour();
    }
  }
}
```

```
Prof paul = new Prof();       | paul.work()   ==> collectPay();
TA daniel = new TA();         | daniel.work()  ==> doLab(); discuss(); ...
Worker wPaul = paul,          | wPaul.work() ==> collectPay();
       wDaniel = daniel;      | wDaniel.work() ==> doLab(); discuss(); ...
```

Lesson: For instance methods (only), select method based on *dynamic type*. Simple to state, but we'll see it has profound consequences.

What About Fields and Static Methods?

```
class Parent {                                  class Child extends Parent {
  int x = 0;                                      String x = "no";
  static int y = 1;                               static String y = "way";
  static void f() {                               static void f() {
     System.out.printf("Ahem!%n");                   System.out.printf("I wanna!%n");
  }                                               }
  static int f(int x) {                         }
    return x+1;
  }
}
```

```
  Child  tom = new Child();  | tom.x    ==> no       pTom.x    ==> 0
  Parent pTom = tom;         | tom.y    ==> way      pTom.y    ==> 1
                             | tom.f()  ==> I wanna! pTom.f()  ==> Ahem!
                             | tom.f(1) ==> 2        pTom.f(1) ==> 2
```

Lesson: Fields *hide* inherited fields of same name; static methods *hide* methods of the same signature.
Real Lesson: Hiding causes confusion; so understand it, but don't do it!

What's the Point?

- The mechanism described here allows us to define a kind of *generic* method.
- A superclass can define a set of operations (methods) that are common to many different classes.
- Subclasses can then provide different implementations of these common methods, each specialized in some way.
- All subclasses will have at least the methods listed by the superclass.
- So when we write methods that operate on the superclass, they will automatically work for all subclasses with no extra work.

CS61B Lecture #9: Interfaces and Abstract Classes

Recreation

Show that for any polynomial with a leading coefficient of 1 and integral coefficients, all rational roots are integers.

Reminder:

The four projects are individual efforts in this class (no partnerships). Feel free to discuss projects or pieces of them before doing the work. But you must complete each project yourself. That is, feel free to discuss projects with each other, but be aware that we expect your work to be substantially different from that of all your classmates (in this or any other semester). You will find a more detailed account of our policy in under the “Course Info” tab on the course website.

Abstract Methods and Classes

- Instance method can be *abstract:* No body given; must be supplied in subtypes.
- One good use is in specifying a pure interface to a family of types:

```
/** A drawable object. */
public abstract class Drawable {
    // "abstract class" = "can't say new Drawable"
    /** Expand THIS by a factor of XSIZE in the X direction,
     *  and YSIZE in the Y direction. */
    public abstract void scale(double xsize, double ysize);

    /** Draw THIS on the standard output. */
    public abstract void draw();
}
```

- Now a `Drawable` is something that has *at least* the operations `scale` and `draw` on it.
- Can't create a `Drawable` because it's abstract.
- In fact, in this case, it wouldn't make any sense to create one, because it has two methods without any implementation.

Methods on Drawables

```
/** A drawable object. */
public abstract class Drawable {
    /** Expand THIS by a factor of SIZE */
    public abstract void scale(double xsize, double ysize);
    /** Draw THIS on the standard output. */
    public abstract void draw();
}
```

- Can't write new Drawable(), *BUT*, we can write methods that operate on `Drawables` in `Drawable` or in other classes:

```
void drawAll(Drawable[] thingsToDraw) {
   for (Drawable thing : thingsToDraw)
       thing.draw();
}
```

- But `draw` has no implementation! How can this work?

Concrete Subclasses

- Regular classes can extend abstract ones to make them "less abstract" by overriding their abstract methods.
- Can define kinds of `Drawable`s that are *concrete*, in that all methods have implementations and one can use **new** on them:

Concrete Subclass Examples

```
public class Rectangle extends Drawable {
    public Rectangle(double w, double h) { this.w = w; this.h = h; }
    public void scale(double xsize, double ysize) {
        w *= xsize; h *= ysize;
    }
    public void draw() { draw a w x h rectangle }
    private double w,h;
}
```

Any Oval or Rectangle is a Drawable.

```
public class Oval extends Drawable {
    public Oval(double xrad, double yrad) {
        this.xrad = xrad; this.yrad = yrad;
    }
    public void scale(double xsize, double ysize) {
        xrad *= xsize; yrad *= ysize;
    }
    public void draw() { draw an oval with axes xrad and yrad }
    private double xrad, yrad;
}
```

Using Concrete Classes

- We *can* create new `Rectangles` and `Ovals`.
- Since these classes are subtypes of `Drawable`, we can put them in any container whose static type is `Drawable`, ...
- ...and therefore can pass them to any method that expects `Drawable` parameters:
- Thus, writing

```
Drawable[] things = {
    new Rectangle(3, 4), new Oval(2, 2)
};
drawAll(things);
```

draws a 3×4 rectangle and a circle with radius 2.

Aside: Documentation

- Our style checker would insist on comments for all the methods, constructors, and fields of the concrete subtypes.
- But we already have comments for `draw` and `scale` in the class `Drawable`, and the whole idea of object-oriented programming is that the subtypes conform to the supertype both in syntax and behavior (all `scale` methods scale their figure), so comments are generally not helpful on overriding methods. Still, the reader would like to know that a given method *does* override something.
- Hence, the `@Override` annotation. We can write:

```
@Override
public void scale(double xsize, double ysize) {
    xrad *= xsize; yrad *= ysize;
}
@Override
public void draw() { draw a circle with radius rad }
```

- The compiler will check that these method headers are proper overridings of the parent's methods, and our style checker won't complain about the lack of comments.

Interfaces

- In generic English usage, an *interface* is a "point where interaction occurs between two systems, processes, subjects, etc." (*Concise Oxford Dictionary*).
- In programming, often use the term to mean a *description* of this generic interaction, specifically, a description of the functions or variables by which two things interact.
- Java uses the term to refer to a slight variant of an abstract class that (until Java 1.7) contains only abstract methods (and static constants), like this:

```
public interface Drawable {
  void scale(double xsize, double ysize);   // Automatically public.
  void draw();
}
```

- Interfaces are automatically abstract: can't say `new Drawable()`; can say `new Rectangle(...)`.

Implementing Interfaces

- Idea is to treat Java interfaces as the public specifications of data types, and classes as their implementations:

  ```
  public class Rectangle implements Drawable { ... }
  ```

  (We *extend* ordinary classes and *implement* interfaces, hence the change in keyword.)

- Can use the interface as for abstract classes:

  ```
  void drawAll(Drawable[] thingsToDraw) {
      for (Drawable thing : thingsToDraw)
          thing.draw();
  }
  ```

- Again, this works for `Rectangles` and any other implementation of `Drawable`.

Multiple Inheritance

- Can *extend* one class, but *implement* any number of interfaces.
- Contrived Example:

```
interface Readable {
  Object get();
}

interface Writable {
  void put(Object x);
}

class Source implements Readable {
  public Object get() { ... }
}
```

```
void copy(Readable r,
          Writable w) {
  w.put(r.get());
}


class Sink implements Writable {
  public void put(Object x) { ... }
}
```

```
class Variable implements Readable, Writable {
  public Object get() { ... }
  public void put(Object x) { ... }
}
```

- The first argument of `copy` can be a `Source` or a `Variable`. The second can be a `Sink` or a `Variable`.

Review: Higher-Order Functions

- In Python, you had *higher-order functions* like this:

  ```
  def map(proc,      items):
     #  function      list
     if items is None:
         return None
     else:
         return IntList(proc(items.head), map(proc, items.tail))
  ```

  and you could write

  ```
      map(abs, makeList(-10, 2, -11, 17))
        ====> makeList(10, 2, 11, 17)
      map(lambda x: x * x, makeList(1, 2, 3, 4))
        ====> makeList(t(1, 4, 9, 16)
  ```

- Java does not have these directly, but can use abstract classes or interfaces and subtyping to get the same effect (with more writing)

Map in Java

```
/** Function with one integer argument */

public interface IntUnaryFunction {
  int apply(int x);
}
```

```
IntList map(IntUnaryFunction proc,
            IntList items) {
  if (items == null)
    return null;
  else return new IntList(
      proc.apply(items.head),
      map(proc, items.tail)
    );
}
```

- It's the use of this function that's clumsy. First, define class for absolute value function; then create an instance:

```
class Abs implements IntUnaryFunction {
  public int apply(int x) { return Math.abs(x); }
}
-----------------------------------------------
R = map(new Abs(), some list);
```

Lambda Expressions

- Since Java 7, one can create classes likes `Abs` on the fly with *anonymous classes*:

```
R = map(new IntUnaryFunction() {
           public int apply(int x) { return Math.abs(x); }
        }, some list);
```

- This is sort of like declaring

```
    class Anonymous implements IntUnaryFunction {
        public int apply(int x) { return Math.abs(x); }
    }
```

  and then writing

```
    R = map(new Anonymous(), some list);
```

Lambda in Java 8

- In Java 8, lambda expressions are even more succinct:

  ```
  R = map((int x) -> Math.abs(x), some list);
          or even better, when the function already exists:
  R = map(Math::abs, some list);
  ```

- These figure out you need an anonymous `IntUnaryFunction` and create one.

- You can see examples in `signpost.GUI`:

  ```
      addMenuButton("Game->New", this::newGame);
  ```

  Here, the second parameter of `ucb.gui2.TopLevel.addMenuButton` is a *call-back function.*

- It has the Java library type `java.util.function.Consumer`, which has a one-argument method, like `IntUnaryFunction`,

Inheriting Headers vs. Method Bodies

- One can implement multiple interfaces, but extend only one class: *multiple interface inheritance*, but *single body inheritance*.
- This scheme is simple, and pretty easy for language implementors to implement.
- However, there are cases where it would be nice to be able to "mix in" implementations from a number of sources.

Extending Supertypes, Default Implementations

- As indicated above, before Java 8, interfaces contained just static constants and abstract methods.
- Java 8 introduced static methods into interfaces and also *default methods*, which are essentially instance methods and are used whenever a method of a class implementing the interface would otherwise be abstract.
- Suppose I want to add a new one-parameter `scale` method to all concrete subclasses of the interface `Drawable`. Normally, that would involve adding an implementation of that method to all concrete classes.
- We could instead make `Drawable` an abstract class again, but in the general case that can have its own problems.

Default Methods in Interfaces

- So Java 8 introduced default methods:

```
public interface Drawable {
  void scale(double xsize, double ysize);
  void draw();

  /** Scale by SIZE in the X and Y dimensions. */
  default void scale(double size) {
      scale(size, size);
  }
}
```

- Useful feature, but, as in other languages with full multiple inheritance (like C++ and Python), it can lead to confusing programs. I suggest you use them sparingly.

CS61B Lecture #10: OOP mechanism and Class Design

Review: A Puzzle

```
class A {                                        class B extends A {
  void f() {                                       void f() {
       System.out.println("A.f");                    System.out.println("B.f");
  }                                                }
  void g() { f(); /* or this.f()  */ }
}                                                }
```

```
class C {
  static void main(String[] args) {
    B aB = new B();
    h(aB);
  }

  static void h(A x) { x.g(); }
}
```

1. What is printed?
2. If we made `g` static?
3. If we made `f` static?
4. If we overrode `g` in `B`?
5. If `f` not defined in A?

Choices

a. `A.f`
b. `B.f`
c. Some kind of error

Review: A Puzzle

```
class A {
  void f() {
       System.out.println("A.f");
  }
  void g() { f(); /* or this.f()  */ }
}
```

```
class B extends A {
  void f() {
    System.out.println("B.f");
  }

}
```

```
class C {
  static void main(String[] args) {
    B aB = new B();
    h(aB);
  }

  static void h(A x) { x.g(); }
}
```

1. What is printed?
2. If we made `g` static?
3. If we made `f` static?
4. If we overrode `g` in `B`?
5. If `f` not defined in A?

Choices

a. `A.f`
b. `B.f`
c. Some kind of error

Review: A Puzzle

```
class A {                                   class B extends A {
  void f() {                                  void f() {
       System.out.println("A.f");               System.out.println("B.f");
  }                                           }
  static void g(A y) { y.f(); }
}                                           }
```

```
class C {
  static void main(String[] args) {
    B aB = new B();
    h(aB);
  }

  static void h(A x) { A.g(x); } // x.g(x) also legal here
}
```

1. What is printed?
2. If we made `g` static?
3. If we made `f` static?
4. If we overrode `g` in `B`?
5. If `f` not defined in A?

Choices

a. `A.f`
b. `B.f`
c. Some kind of error

Review: A Puzzle

```
class A {                                  |  class B extends A {
  void f() {                               |    void f() {
       System.out.println("A.f");          |      System.out.println("B.f");
  }                                        |    }
  static void g(A y) { y.f(); }            |
}                                          |  }
```

```
class C {
  static void main(String[] args) {
    B aB = new B();
    h(aB);
  }

  static void h(A x) { A.g(x); } // x.g(x) also legal here
}
```

1. What is printed?
2. If we made `g` static?
3. If we made `f` static?
4. If we overrode `g` in `B`?
5. If `f` not defined in A?

Choices

a. `A.f`

b. `B.f`

c. Some kind of error

Review: A Puzzle

```
class A {
  static void f() {
       System.out.println("A.f");
  }
  void g() { f(); /* or this.f()  */ }
}
```

```
class B extends A {
  static void f() {
    System.out.println("B.f");
  }

}
```

```
class C {
  static void main(String[] args) {
    B aB = new B();
    h(aB);
  }

  static void h(A x) { x.g(); }
}
```

1. What is printed?
2. If we made `g` static?
3. If we made `f` static?
4. If we overrode `g` in `B`?
5. If `f` not defined in A?

Choices

a. `A.f`
b. `B.f`
c. Some kind of error

Review: A Puzzle

```
class A {                                    |  class B extends A {
  static void f() {                          |    static void f() {
       System.out.println("A.f");            |      System.out.println("B.f");
  }                                          |    }
  void g() { f(); /* or this.f()  */ }       |
}                                            |  }
```

```
class C {
  static void main(String[] args) {
    B aB = new B();
    h(aB);
  }

  static void h(A x) { x.g(); }
}
```

1. What is printed?
2. If we made `g` static?
3. If we made `f` static?
4. If we overrode `g` in `B`?
5. If `f` not defined in A?

Choices

a. `A.f`
b. `B.f`
c. Some kind of error

Review: A Puzzle

```
class A {                                        class B extends A {
  void f() {                                       void f() {
      System.out.println("A.f");                     System.out.println("B.f");
  }                                                }
  void g() { f(); /* or this.f()  */ }             void g() { f(); }
}                                                }
```

```
class C {
  static void main(String[] args) {
    B aB = new B();
    h(aB);
  }

  static void h(A x) { x.g(); }
}
```

1. What is printed?
2. If we made g static?
3. If we made f static?
4. If we overrode g in B?
5. If f not defined in A?

Choices

a. `A.f`
b. `B.f`
c. Some kind of error

Review: A Puzzle

```
class A {                                          class B extends A {
  void f() {                                         void f() {
       System.out.println("A.f");                      System.out.println("B.f");
  }                                                  }
  void g() { f(); /* or this.f()  */ }               void g() { f(); }
}                                                  }
```

```
class C {
  static void main(String[] args) {
    B aB = new B();
    h(aB);
  }

  static void h(A x) { x.g(); }
}
```

1. What is printed?
2. If we made `g` static?
3. If we made `f` static?
4. If we overrode `g` in `B`?
5. If `f` not defined in A?

Choices

a. `A.f`
b. `B.f`
c. Some kind of error

Review: A Puzzle

```
class A {                                        | class B extends A {
                                                 |   void f() {
  void g() { f(); /* or this.f()  */ }           |     System.out.println("B.f");
}                                                |   }
                                                 |
                                                 | }
```

```
class C {
  static void main(String[] args) {
    B aB = new B();
    h(aB);
  }

  static void h(A x) { x.g(); }
}
```

1. What is printed?
2. If we made `g` static?
3. If we made `f` static?
4. If we overrode `g` in `B`?
5. If `f` not defined in A?

Choices

a. `A.f`

b. `B.f`

c. Some kind of error

Review: A Puzzle

```
class A {

  void g() { f(); /* or this.f()  */ }
}
```

```
class B extends A {
  void f() {
    System.out.println("B.f");
  }

}
```

```
class C {
  static void main(String[] args) {
    B aB = new B();
    h(aB);
  }

  static void h(A x) { x.g(); }
}
```

1. What is printed?
2. If we made g static?
3. If we made f static?
4. If we overrode g in B?
5. If f not defined in A?

Choices

a. A.f
b. B.f
c. Some kind of error

Answer to Puzzle

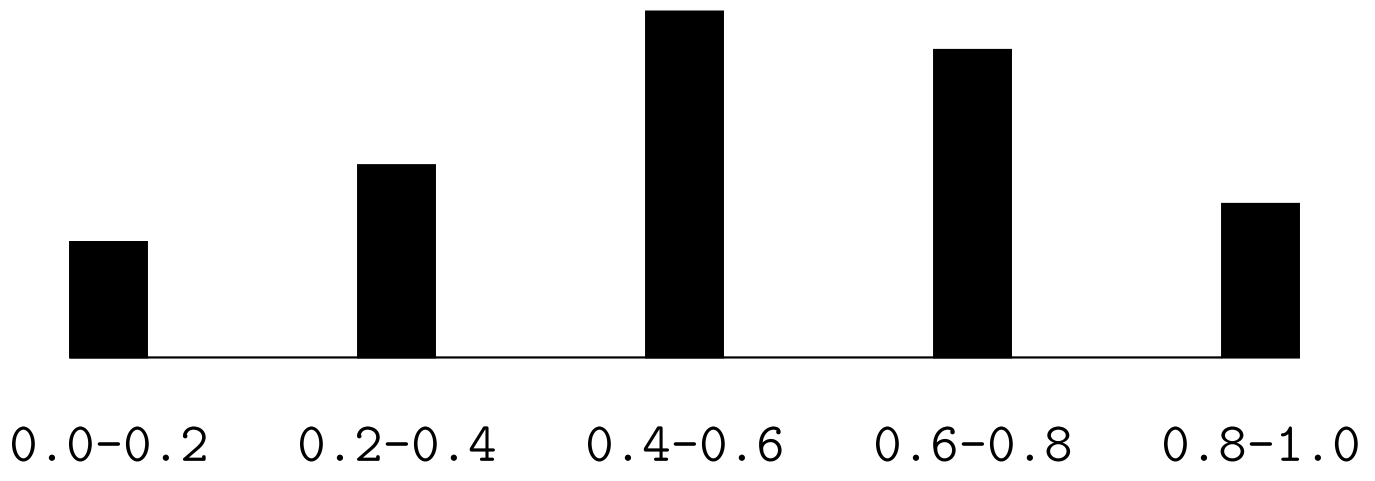
1. Executing `java C` prints ______, because
   A. `C.main` calls `h` and passes it `aB`, whose dynamic type is `B`.
   B. `h` calls `x.g()`. Since `g` is inherited by `B`, we execute the code for `g` in class `A`.
   C. `g` calls `this.f()`. Now `this` contains the value of `h`'s argument, whose dynamic type is `B`. Therefore, we execute the definition of `f` that is in `B`.
   D. In calls to `f`, in other words, static type is ignored in figuring out what method to call.
2. If `g` were static, we see ______; selection of `f` still depends on dynamic type of `this`. Same for overriding `g` in `B`.
3. If `f` were static, would print ______ because then selection of `f` would depend on static type of `this`, which is `A`.
4. If `f` were not defined in `A`, we'd see ______

Answer to Puzzle

1. Executing `java C` prints __B.f__, because
   - A. `C.main` calls `h` and passes it `aB`, whose dynamic type is `B`.
   - B. `h` calls `x.g()`. Since `g` is inherited by `B`, we execute the code for `g` in class `A`.
   - C. `g` calls `this.f()`. Now `this` contains the value of `h`'s argument, whose dynamic type is `B`. Therefore, we execute the definition of `f` that is in `B`.
   - D. In calls to `f`, in other words, static type is ignored in figuring out what method to call.
2. If `g` were static, we see __B.f__; selection of `f` still depends on dynamic type of `this`. Same for overriding `g` in `B`.
3. If `f` were static, would print __A.f__ because then selection of `f` would depend on static type of `this`, which is `A`.
4. If `f` were not defined in `A`, we'd see __a compile-time error__

Example: Designing a Class

Problem: Want a class that represents histograms, like this one:



Analysis: What do we need from it? At least:

- Specify buckets and limits.
- Accumulate counts of values.
- Retrieve counts of values.
- Retrieve numbers of buckets and other initial parameters.

Specification Seen by Clients

- The *clients* of a module (class, program, etc.) are the programs or methods that *use* that module's exported definitions.
- In Java, intention is that exported definitions are designated **public**.
- Clients are intended to rely on *specifications,* (*aka* APIs) not code.
- *Syntactic specification:* method and constructor headers—syntax needed to use.
- *Semantic specification:* what they do. No formal notation, so use comments.
  - Semantic specification is a *contract.*
  - Conditions client must satisfy (*preconditions*, marked "Pre:" in examples below).
  - Promised results (*postconditions*).
  - Design these to be *all the client needs!*
  - Exceptions communicate errors, specifically failure to meet preconditions.

Histogram Specification and Use

```java
/** A histogram of floating-point values */
public interface Histogram {
  /** The number of buckets in THIS. */
  int size();

  /** Lower bound of bucket #K. Pre: 0<=K<size(). */
  double low(int k);

  /** # of values in bucket #K. Pre: 0<=K<size(). */
  int count(int k);

  /** Add VAL to the histogram. */
  void add(double val);
}
```

Sample output:

```
>=  0.00 |   10
>= 10.25 |   80
>= 20.50 |  120
>= 30.75 |   50
```

```java
void fillHistogram(Histogram H,
                   Scanner in)
{
    while (in.hasNextDouble())
       H.add(in.nextDouble());
}
```

```java
void printHistogram(Histogram H) {
    for (int i = 0; i < H.size(); i += 1)
       System.out.printf
          (">=%5.2f | %4d%n",
           H.low(i), H.count(i));
}
```

An Implementation

```
public class FixedHistogram implements Histogram {
  private double low, high; /* From constructor*/
  private int[] count; /* Value counts */

  /** A new histogram with SIZE buckets of values >= LOW and < HIGH. */
  public FixedHistogram(int size, double low, double high)
  {
    if (low >= high || size <= 0) throw new IllegalArgumentException();
    this.low = low; this.high = high;
    this.count = new int[size];
  }

  public int size() { return count.length; }
  public double low(int k) { return low + k * (high-low)/count.length; }

  public int count(int k) { return count[k]; }

  public void add(double val) {
     if (val >= low && val < high)
          count[(int) ((val-low)/(high-low) * count.length)] += 1;
  }
}
```

Let's Make a Tiny Change

Don't require *a priori* bounds:

```
class FlexHistogram implements Histogram {
  /** A new histogram with SIZE buckets. */
  public FlexHistogram(int size) {
    ?
  }
  // What needs to change?
}
```

- How would you do this? Profoundly changes implementation.
- But *clients* (like `printHistogram` and `fillHistogram`) still work with no changes.
- Illustrates the power of *separation of concerns*.

Implementing the Tiny Change

- Pointless to pre-allocate the `count` array.
- Don't know bounds, so must save arguments to `add`.
- Then recompute `count` array "lazily" when `count(···)` called.
- Invalidate `count` array whenever histogram changes.

```
class FlexHistogram implements Histogram {
   private ArrayList<Double> values = new ArrayList<>();
   int size;
   private int[] count;

   public FlexHistogram(int size) { this.size = size; this.count = null;
}

   public void add(double x) { count = null; values.add(x); }

   public int count(int k) {
     if (count == null) { compute count from values here. }
     return count[k];
   }
}
```

Advantages of Procedural Interface over Visible Fields

By using public method for `count` instead of making the array `count` visible, the "tiny change" is transparent to clients:

- If client had to write `myHist.count[k]`, it would mean

  "The number of items currently in the k^{th} bucket of histogram `myHist` (which, by the way, is stored in an array called `count` in `myHist` that always holds the up-to-date count)."

- Parenthetical comment *worse than useless* to the client.
- If `count` array had been visible, after "tiny change," *every use* of `count` in client program would have to change.
- So using a method for the public `count` method decreases what client *has to* know, and (therefore) has to change.

Recreation

An integer is divided by 9 when a certain one of its digits is deleted, and the resulting number is again divisible by 9.

a. Prove that actually dividing the resulting number by 9 results in deleting another digit.

b. Find all integers satisfying the conditions of this problem.

Announcements

- Project 0 autograder has been running. Check the Scores tab for results.
- Yes, you can resubmit. See the Course Info tab.
- In particular, *many* people need to do style fixes! Use `make style` or `style61b signpost/*.java` to check before submission.

Project Ethics

Basic Rules:

1. By You Alone: All major submitted non-skeleton code should be written by you alone.
2. Do Not Possess or Share Code: Before a project deadline, you should never be in possession of solution code that you did not write, nor distribute your own code to others in the class.
3. Cite Your Sources: When you receive significant assistance on a project from someone else (other than the staff), cite that assistance somewhere in your source code.

Ethical Collaboration

Unprojectematic

- Discussion of approaches for solving a problem.
- Giving away or receiving significant ideas towards a problem solution, if cited.
- Discussion of specific syntax issues and bugs in your code.
- Using small snippets of code that you find online for solving tiny problems (e.g. googling "uppercase string java" may lead you to some sample code that you copy and paste. Cite these.

Requiring Great Caution:

- Looking at someone else's project code to assist with debugging.
- Looking at someone else's project code to understand a particular idea or part of a project. Generally unwise though, due to the danger of plagiarism.

Unethical Collaborations

- Possessing another student's project code in any form before a final deadline, or distributing your own.
- Possessing project solution code that you did not write yourself before a final deadline (e.g., from github, or from staff solution code found somewhere). Likewise, distributing such code.

CS61B Lecture #11: Examples: Comparable & Reader

Comparable

- Java library provides an interface to describe Objects that have a *natural order* on them, such as String, Integer, BigInteger and BigDecimal:

```
public interface Comparable { // For now, the Java 1.4 version
  /** Returns value <0, == 0, or > 0 depending on whether THIS is
   *  <, ==, or > OBJ.  Exception if OBJ not of compatible type. */
  int compareTo(Object obj);
}
```

- Might use in a general-purpose max function:

```
/** The largest value in array A, or null if A empty. */
public static Comparable max(Comparable[] A) {
  if (A.length == 0) return null;
  Comparable result; result = A[0];
  for (int i = 1; i < A.length; i += 1)
    if (result.compareTo(A[i]) < 0) result = A[i];
  return result;
}
```

- Now `max(S)` will return maximum value in `S` if `S` is an array of Strings, or any other kind of Object that implements Comparable.

Examples: Implementing Comparable

```java
/** A class representing a sequence of ints. */
class IntSequence implements Comparable {
    private int[] myValues;
    private int myCount;
    ...
    public int get(int k) { return myValues[k]; }

    @Override
    public int compareTo(Object obj) {
       IntSequence x = (IntSequence) obj; // Blows up if obj not an IntSequence
       for (int i = 0; i < myCount && i < x.myCount; i += 1) {
           if (myValues[i] < x.myValues[i]) {
              return -1;
           } else if (myValues[i] > x.myValues[i]) {
              return 1;
       }
       return myCount - x.myCount;  // <0 iff myCount < x.myCount
    }
}
```

Implementing Comparable II

- Also possible to add an interface retroactively.
- If IntSequence did not implement Comparable, but did implement compareTo (without @Override), we could write

  ```
  class ComparableIntSequence extends IntSequence implements Comparable {

  }
  ```
- Java would then "match up" the compareTo in IntSequence with that in Comparable.

Java Generics (I)

- We've shown you the old Java 1.4 `Comparable`. The current version uses a newer feature: Java generic types:

```
public interface Comparable<T> {
    int compareTo(T x);
}
```

- Here, `T` is like a formal parameter in a method, except that its "value" is a *type*.
- Revised `IntSequence` (no casting needed):

```
class IntSequence implements Comparable<IntSequence> {
    ...
    @Override
    public int compareTo(IntSequence x) {
       for (int i = 0; i < myCount && i < x.myCount; i += 1) {
           if (myValues[i] < x.myValues[i]) ...

       return myCount - x.myCount;
    }
}
```

Example: Readers

- Java class `java.io.Reader` abstracts *sources of characters*.
- Here, we present a revisionist version (not the real thing):

```
public interface Reader {  // Real java.io.Reader is abstract class
  /** Release this stream: further reads are illegal */
  void close();

  /** Read as many characters as possible, up to LEN,
   *  into BUF[OFF], BUF[OFF+1],..., and return the
   *  number read, or -1 if at end-of-stream. */
  int read(char[] buf, int off, int len);

  /** Short for read(BUF, 0, BUF.length). */
  int read(char[] buf);

  /** Read and return single character, or -1 at end-of-stream. */
  int read();
}
```

- Can't write `new Reader()`; it's abstract. So what good is it?

Generic Partial Implementation

- According to their specifications, some of `Reader`'s methods are related.
- Can express this with a *partial implementation,* which leaves key methods unimplemented and provides default bodies for others.
- Result still abstract: can't use **new** on it.

```
/** A partial implementation of Reader. Concrete
 *  implementations MUST override close and read(,,).
 *  They MAY override the other read methods for speed. */
public abstract class AbstractReader implements Reader {
  // Next two lines are redundant.
  public abstract void close();
  public abstract int read(char[] buf, int off, int len);

  public int read(char[] buf) { return read(buf,0,buf.length); }

  public int read() { return (read(buf1) == -1) ? -1 : buf1[0]; }

  private char[] buf1 = new char[1];
}
```

Implementation of Reader: StringReader

The class StringReader reads characters from a String:

```
public class StringReader extends AbstractReader {
  private String str;
  private int k;
  /** A Reader that delivers the characters in STR. */
  public StringReader(String s) {
      str = s; k = 0;
  }

  public void close() {
      str = null;
  }

  public int read(char[] buf, int off, int len) {
      if (k == str.length())
          return -1;
      len = Math.min(len, str.length() - k);
      str.getChars(k, k+len, buf, off);
      k += len;
      return len;
  }
}
```

Using Reader

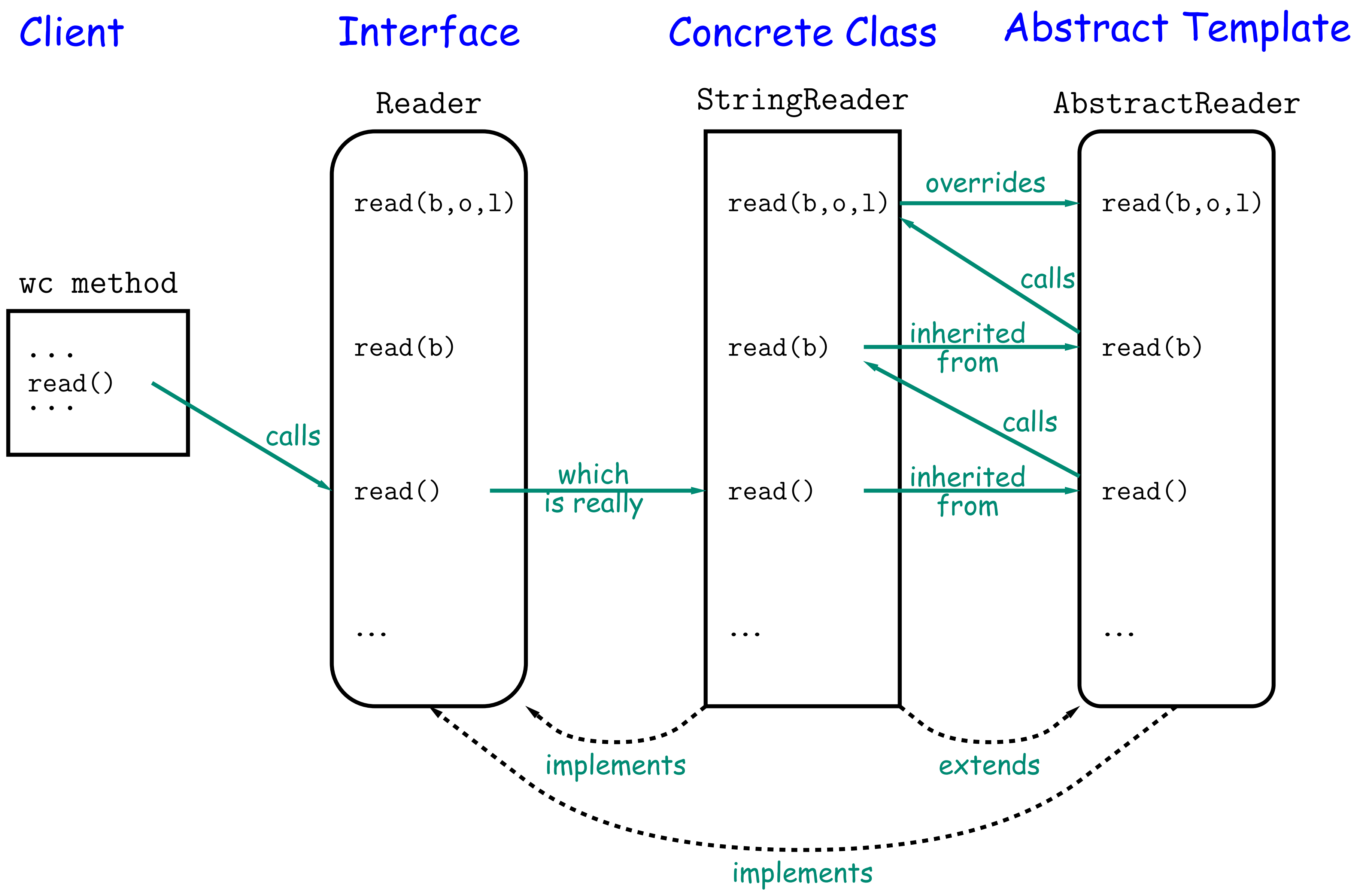
Consider this method, which counts words:

```
/** The total number of words in R, where a "word" is
 *  a maximal sequence of non-whitespace characters. */
int wc(Reader r) {
  int c0, count;
  c0 = ' '; count = 0;
  while (true) {
      int c = r.read();
      if (c == -1) return count;
      if (Character.isWhitespace((char) c0)
          && !Character.isWhitespace((char) c))
          count += 1;
      c0 = c;
  }
}
```

This method works for *any* `Reader`:

```
wc(new StringReader(someText))          // # words in someText
wc(new InputStreamReader(System.in))    // # words in standard input
wc(new FileReader("foo.txt"))           // # words in file foo.txt.
```

How It Fits Together



Lessons

- The `Reader` interface class served as a *specification* for a whole set of readers.
- Ideally, most client methods that deal with `Readers`, like `wc`, will specify type `Reader` for the formal parameters, not a specific kind of `Reader`, thus assuming as little as possible.
- And only when a client creates a new `Reader` will it get specific about what subtype of `Reader` it needs.
- That way, client's methods are as *widely applicable* as possible.
- Finally, `AbstractReader` is a tool for implementors of non-abstract `Reader` classes, and not used by clients.
- Alas, Java library is not pure. E.g., `AbstractReader` is really just called `Reader` and there is no interface. In this example, we saw what they *should* have done!
- The `Comparable` interface allows definition of functions that depend only on a limited subset of the properties (methods) of their arguments (such as "must have a `compareTo` method").

To Think About

- A student adds a JUnit test:

```
@Test
public void mogrifyTest() {
    assertEquals("mogrify fails",
                 new int[] { 2, 4, 8, 12 },
                 MyClass.mogrify(new int[] { 1, 2, 4, 6 }));
}
```

  The test always seems to fail, no matter what `mogrify` does. Why?

- A student sees this in an autograder log:

```
Fatal: no proj0/signpost directory.
```

  What is likely to be the problem?

- A student does not see his proj0 submission under the Scores tab. What can be the problem?

CS61B Lecture #12: Additional OOP Details, Exceptions

Parent Constructors

- In lecture notes #5, talked about how Java allows implementer of a class to control all manipulation of objects of that class.
- In particular, this means that Java gives the constructor of a class the first shot at each new object.
- When one class extends another, there are two constructors—one for the parent type and one for the new (child) type.
- In this case, Java guarantees that one of the parent's constructors is called first. In effect, there is a call to a parent constructor at the beginning of every one of the child's constructors.
- You can call the parent's constructor yourself. By default, Java calls the "default" (parameterless) constructor.

```
class Figure {
  public Figure(int sides) {
      ...
  }...
}
```

```
class Rectangle extends Figure {
   public Rectangle() {
       super(4);
   }...
}
```

Using an Overridden Method

- Suppose that you wish to *add* to the action defined by a superclass's method, rather than to completely override it.
- The overriding method can refer to overridden methods by using the special prefix super.
- For example, you have a class with expensive functions, and you'd like a memoizing version of the class.

```
class ComputeHard {
  int cogitate(String x, int y) { ... }
}

class ComputeLazily extends ComputeHard {
  int cogitate(String x, int y) {
    if (don't already have answer for this x and y) {
       int result = super.cogitate(x, y);  // <<< Calls overridden function
       memoize (save) result;
       return result;
    }
     return memoized result;
  }
}
```

Trick: Delegation and Wrappers

- Not always appropriate to use inheritance to extend something.
- Homework gives example of a `TrReader`, which *contains* another `Reader`, to which it *delegates* the task of actually going out and reading characters.
- Another example: a class that instruments objects:

```
interface Storage {
  void put(Object x);
  Object get();
}
```

```
class Monitor implements Storage {
  int gets, puts;
  private Storage store;
  Monitor(Storage x) { store = x; gets = puts = 0; }
  public void put(Object x) { puts += 1; store.put(x); }
  public Object get() { gets += 1; return store.get(); }
}
```

```
// ORIGINAL
Storage S = something;
f(S);
```

```
// INSTRUMENTED
Monitor S = new Monitor(something);
f(S);
System.out.println(S.gets + " gets");
```

`Monitor` is called a *wrapper class*.

What to do About Errors?

- Large amount of any production program devoted to detecting and responding to errors.
- Some errors are external (bad input, network failures); others are internal errors in programs.
- When method has stated precondition, it's the client's job to comply.
- Still, it's nice to detect and report client's errors.
- In Java, we *throw exception objects,* typically:

  `throw new` *SomeException* `(`*optional description*`);`

- Exceptions are objects. By convention, they are given two constructors: one with no arguments, and one with a descriptive string argument (which the exception stores).
- Java system throws some exceptions implicitly, as when you dereference a null pointer, or exceed an array bound.

Catching Exceptions

- A **throw** causes each active method call to *terminate abruptly,* until (and unless) we come to a **try** block.
- Catch exceptions and do something corrective with **try**:

```
try {
    Stuff that might throw exception;
} catch (SomeException e) {
    Do something reasonable;
} catch (SomeOtherException e) {
    Do something else reasonable;
}
Go on with life;
```

- When *SomeException* exception occurs during "Stuff..." and is not handled there, we immediately "do something reasonable" and then "go on with life."
- Descriptive string (if any) available as `e.getMessage()` for error messages and the like.

Catching Exceptions, II

- Using a supertype as the parameter type in a **catch** clause will catch any subtype of that exception as well:

```
try {
    Code that might throw a FileNotFoundException or a
        MalformedURLException ;
catch (IOException ex) {
    Handle any kind of IOException;
}
```

- Since `FileNotFoundException` and `MalformedURLException` both inherit from *IOException*, the **catch** handles both cases.
- Subtyping means that multiple **catch** clauses can apply; Java takes the first.
- Stylistically, it's nice to be more (concrete) about exception types where possible.
- In particular, our style checker will therefore balk at the use of `Exception`, `RuntimeException`, `Error`, and `Throwable` as exception supertypes.

Catching Exceptions, III

- There's a relatively new shorthand for handling multiple exceptions the same way:

```
try {
    Code that might throw IllegalArgumentException
        or IllegalStateException;
catch (IllegalArgumentException|IllegalStateException ex) {
    Handle exception;
}
```

Exceptions: Checked vs. Unchecked

- The object thrown by **throw** command must be a subtype of `Throwable` (in `java.lang`).
- Java pre-declares several such subtypes, among them
  - `Error`, used for serious, unrecoverable errors;
  - `Exception`, intended for all other exceptions;
  - `RuntimeException`, a subtype of `Exception` intended mostly for programming errors too common to be worth declaring.
- Pre-declared exceptions are all subtypes of one of these.
- Any subtype of `Error` or `RuntimeException` is said to be *unchecked.*
- All other exception types are *checked.*

Unchecked Exceptions

- Intended for
  - Programmer errors: many library functions throw `IllegalArgumentException` when one fails to meet a precondition.
  - Errors detected by the basic Java system: e.g.,
    - Executing `x.y` when `x` is null,
    - Executing `A[i]` when `i` is out of bounds,
    - Executing `(String) x` when `x` turns out not to point to a `String`.
  - Certain catastrophic failures, such as running out of memory.
- May be thrown anywhere at any time with no special preparation.

Checked Exceptions

- Intended to indicate exceptional circumstances that are not necessarily programmer errors. Examples:
  - Attempting to open a file that does not exist.
  - Input or output errors on a file.
  - Receiving an interrupt.
- Every checked exception that can occur inside a method must either be handled by a `try` statement, or reported in the method's declaration.
- For example,

  ```
  void myRead() throws IOException, InterruptedException { ... }
  ```

  means that myRead (or something it calls) *might* throw `IOException` or `InterruptedException`.
- Language Design: Why did Java make the following illegal?

  ```
  class Parent {                class Child extends Parent {
     void f() { ... }               void f () throws IOException { ... }
  }                             }
  ```

Good Practice

- Throw exceptions rather than using print statements and System.exit everywhere,
- ...because response to a problem may depend on the *caller*, not just method where problem arises.
- Nice to throw an exception when programmer violates preconditions.
- Particularly good idea to throw an exception rather than let bad input corrupt a data structure.
- Good idea to document when methods throw exceptions.
- To convey information about the cause of exceptional condition, put it into the exception rather than into some global variable:

```
class MyBad extends Exception {
   public IntList errs;
   MyBad(IntList nums) { errs=nums; }
}
```

```
try {...
} catch (MyBad e) {
 ... e.errs ...
}
```

CS61B Lecture #13: Packages, Access, Loose Ends

- Modularization facilities in Java.
- Importing
- Nested classes.
- Using overridden method.
- Parent constructors.
- Type testing.

Package Mechanics

- Classes correspond to things being modeled (represented) in one's program.
- Packages are collections of "related" classes and other packages.
- Java puts standard libraries and packages in package `java` and `javax`.
- By default, a class resides in the *anonymous package.*
- To put it elsewhere, use a `package` declaration at start of file, as in

  `package database;` *or* `package ucb.util;`

- Oracle's `javac` uses convention that class `C` in package `P1.P2` goes in subdirectory `P1/P2` of any other directory in the *class path*.
- Unix example:

```
$ export CLASSPATH=.:$HOME/java-utils:$MASTERDIR/lib/classes/junit.jar
$ java junit.textui.TestRunner MyTests
```

Searches for TestRunner.class in ./junit/textui, ~/java-utils/junit/textui and finally looks for junit/textui/TestRunner.class in the junit.jar file (which is a single file that is a special compressed archive of an entire directory of files).

Access Modifiers

- Access modifiers (**private, public, protected**) do not add anything to the power of Java.
- Basically allow a programmer to declare which classes are supposed to need to access (“know about”) what declarations.
- In Java, are also part of security—prevent programmers from accessing things that would “break” the runtime system.
- Accessibility always determined by static types.
  - To determine correctness of writing `x.f()`, look at the definition of `f` in the *static type* of `x`.
  - Why the static type? Because the rules are supposed to be enforced by the compiler, which only knows static types of things (static types don’t depend on what happens at execution time).

The Access Rules: Public

- Accessibility of a member depends on (1) how the member's declaration is qualified and (2) where it is being accessed.
- C1, C2, C3, and C4 are distinct classes.
- Class C2a is either class C2 itself or a subtype of C2.

```
package P1;
public class C1 ... {
  // M is a method, field,...
  public int M ...
  void h(C1 x)
    { ... x.M ... } // OK.
}
------------------------------
package P1;
public class C4 ... {
  void p(C1 x)
    { ... x.M ... } // OK.
}
```

```
package P2;
class C2 extends C3 {
  void f(P1.C1 x) {... x.M ...} // OK
  void g(C2a y) {... y.M ...  } // OK
}
```

Public members are available evrywhere.

The Access Rules: Private

- C1, C2, and C4 are distinct classes.
- Class C2a is either class C2 itself or a subtype of C2.

```
package P1;
public class C1 ... {
  // M is a method, field,...
  private int M ...
  void h(C1 x)
    { ... x.M ... } // OK.
}
------------------------------
package P1;
public class C4 ... {
  void p(C1 x)
    { ... x.M ... } // ERROR.
}
```

```
package P2;
class C2 extends C1 {
  void f(P1.C1 x) {... x.M ...} // ERROR
  void g(C2a y) {... y.M ...  } // ERROR
}
```

Private members are available only within the text of the same class, even for subtypes.

The Access Rules: Package Private

- C1, C2, and C4 are distinct classes.
- Class C2a is either class C2 itself or a subtype of C2.

```
package P1;
public class C1 ... {
  // M is a method, field,...
  int M ...
  void h(C1 x)
    { ... x.M ... } // OK.
}
------------------------------
package P1;
public class C4 ... {
  void p(C1 x)
    { ... x.M ... } // OK.
}
```

```
package P2;
class C2 extends C1 {
  void f(P1.C1 x) {... x.M ...} // ERROR
  void g(C2a y) {... y.M ...  } // ERROR
}
```

Package Private members are available only within the same package (even for subtypes).

The Access Rules: Protected

- C1, C2, and C4 are distinct classes.
- Class C2a is either class C2 itself or a subtype of C2.

```
package P1;
public class C1 ... {
  // M is a method, field,...
  protected int M ...
  void h(C1 x)
    { ... x.M ... } // OK.
}
------------------------------
package P1;
public class C4 ... {
  void p(C1 x)
    { ... x.M ... } // OK.
}
```

```
package P2;
class C2 extends C1 {
  void f(P1.C1 x) {... x.M ...} // ERROR
       // (x's type is not subtype of C2.)
  void g(C2a y) {... y.M ...  } // OK
  void g2()     {... M ... } // OK (this.M)
}
```

Protected members of C1 are available within P1, as for package private. Outside P1, they are available within subtypes of C1 such as C2, but only if accessed from expressions whose static types are subtypes of C2.

What May be Controlled

- Classes and interfaces that are not nested may be public or package private (we haven't talked explicitly about nested types yet).
- Members—fields, methods, constructors, and (later) nested types—may have any of the four access levels.
- May *override* a method only with one that has *at least* as permissive an access level. Reason: avoid inconsistency:

```
package P1;
public class C1 {
  public int f() { ... }
}

public class C2 extends C1 {
  // Actually a compiler error; pretend
  // it's not and see what happens
  int f() { ... }
}
```

```
package P2;
class C3 {
  void g(C2 y2) {
    C1 y1 = y2
    y2.f(); // Bad???
    y1.f(); // OK??!!?
  }
}
```

That is, there's no point in restricting C2.f, because access control depends on static types, and C1.f is public.

Intentions of this Design

- `public` declarations represent *specifications*—what clients of a package are supposed to rely on.
- `package private` declarations are part of the *implementation* of a class that must be known to other classes that assist in the implementation.
- `protected` declarations are part of the implementation that subtypes may need, but that clients of the subtypes generally won't.
- `private` declarations are part of the implementation of a class that only that class needs.

Quick Quiz

```
package SomePack;
public class A1 {
  int f1() {
    A1 a = ...
    a.x1 = 3; // OK?
  }
  protected int y1;
  private int x1;
}
```

```
// Anonymous package

class A2 {
  void g(SomePack.A1 x) {
    x.f1();  // OK?
    x.y1 = 3; // OK?
  }
}

class B2 extends SomePack.A1 {
  void h(SomePack.A1 x) {
    x.f1();  // OK?
    x.y1 = 3; // OK?
    f1();     // OK?
    y1 = 3;   // OK?
    x1 = 3;   // OK?
  }
}
```

- **Note:** Last three lines of `h` have implicit **this**.'s in front. Static type of **this** is `B2`.

Quick Quiz

```
package SomePack;
public class A1 {
  int f1() {
    A1 a = ...
    a.x1 = 3; // OK
  }
  protected int y1;
  private int x1;
}
```

```
// Anonymous package

class A2 {
  void g(SomePack.A1 x) {
    x.f1();  // OK?
    x.y1 = 3; // OK?
  }
}

class B2 extends SomePack.A1 {
  void h(SomePack.A1 x) {
    x.f1();  // OK?
    x.y1 = 3; // OK?
    f1();     // OK?
    y1 = 3;   // OK?
    x1 = 3;   // OK?
  }
}
```

- **Note:** Last three lines of `h` have implicit **this**.'s in front. Static type of **this** is `B2`.

Quick Quiz

```
package SomePack;
public class A1 {
  int f1() {
    A1 a = ...
    a.x1 = 3; // OK
  }
  protected int y1;
  private int x1;
}
```

```
// Anonymous package

class A2 {
  void g(SomePack.A1 x) {
    x.f1();  // ERROR
    x.y1 = 3; // OK?
  }
}

class B2 extends SomePack.A1 {
  void h(SomePack.A1 x) {
    x.f1();  // OK?
    x.y1 = 3; // OK?
    f1();     // OK?
    y1 = 3;   // OK?
    x1 = 3;   // OK?
  }
}
```

- **Note:** Last three lines of `h` have implicit **this**.'s in front. Static type of **this** is `B2`.

Quick Quiz

```
package SomePack;
public class A1 {
  int f1() {
    A1 a = ...
    a.x1 = 3; // OK
  }
  protected int y1;
  private int x1;
}
```

```
// Anonymous package

class A2 {
  void g(SomePack.A1 x) {
    x.f1();  // ERROR
    x.y1 = 3; // ERROR
  }
}

class B2 extends SomePack.A1 {
  void h(SomePack.A1 x) {
    x.f1();  // OK?
    x.y1 = 3; // OK?
    f1();     // OK?
    y1 = 3;   // OK?
    x1 = 3;   // OK?
  }
}
```

- **Note:** Last three lines of `h` have implicit **this**.'s in front. Static type of **this** is `B2`.

Quick Quiz

```
package SomePack;
public class A1 {
  int f1() {
    A1 a = ...
    a.x1 = 3; // OK
  }
  protected int y1;
  private int x1;
}
```

```
// Anonymous package

class A2 {
  void g(SomePack.A1 x) {
    x.f1();  // ERROR
    x.y1 = 3; // ERROR
  }
}

class B2 extends SomePack.A1 {
  void h(SomePack.A1 x) {
    x.f1();  // ERROR
    x.y1 = 3; // OK?
    f1();     // OK?
    y1 = 3;   // OK?
    x1 = 3;   // OK?
  }
}
```

- **Note:** Last three lines of `h` have implicit **this.**'s in front. Static type of **this** is `B2`.

Quick Quiz

```
package SomePack;
public class A1 {
  int f1() {
    A1 a = ...
    a.x1 = 3; // OK
  }
  protected int y1;
  private int x1;
}
```

```
// Anonymous package

class A2 {
  void g(SomePack.A1 x) {
    x.f1();  // ERROR
    x.y1 = 3; // ERROR
  }
}

class B2 extends SomePack.A1 {
  void h(SomePack.A1 x) {
    x.f1();  // ERROR
    x.y1 = 3; // OK?
    f1();     // ERROR
    y1 = 3;   // OK?
    x1 = 3;   // OK?
  }
}
```

- **Note:** Last three lines of `h` have implicit **this**.'s in front. Static type of **this** is `B2`.

Quick Quiz

```
package SomePack;
public class A1 {
  int f1() {
    A1 a = ...
    a.x1 = 3; // OK
  }
  protected int y1;
  private int x1;
}
```

```
// Anonymous package

class A2 {
  void g(SomePack.A1 x) {
    x.f1();  // ERROR
    x.y1 = 3; // ERROR
  }
}

class B2 extends SomePack.A1 {
  void h(SomePack.A1 x) {
    x.f1();  // ERROR
    x.y1 = 3; // OK?
    f1();     // ERROR
    y1 = 3;   // OK
    x1 = 3;   // OK?
  }
}
```

- **Note:** Last three lines of `h` have implicit **this.**'s in front. Static type of **this** is `B2`.

Quick Quiz

```
package SomePack;
public class A1 {
  int f1() {
    A1 a = ...
    a.x1 = 3; // OK
  }
  protected int y1;
  private int x1;
}
```

```
// Anonymous package

class A2 {
  void g(SomePack.A1 x) {
    x.f1();  // ERROR
    x.y1 = 3; // ERROR
  }
}

class B2 extends SomePack.A1 {
  void h(SomePack.A1 x) {
    x.f1();  // ERROR
    x.y1 = 3; // OK?
    f1();     // ERROR
    y1 = 3;   // OK
    x1 = 3;   // ERROR
  }
}
```

- **Note:** Last three lines of `h` have implicit **this**.'s in front. Static type of **this** is `B2`.

Quick Quiz

```
package SomePack;
public class A1 {
  int f1() {
    A1 a = ...
    a.x1 = 3; // OK
  }
  protected int y1;
  private int x1;
}
```

```
// Anonymous package

class A2 {
  void g(SomePack.A1 x) {
    x.f1();  // ERROR
    x.y1 = 3; // ERROR
  }
}

class B2 extends SomePack.A1 {
  void h(SomePack.A1 x) {
    x.f1();  // ERROR
    x.y1 = 3; // ERROR
    f1();     // ERROR
    y1 = 3;   // OK
    x1 = 3;   // ERROR
  }
}
```

- **Note:** Last three lines of `h` have implicit **this**.'s in front. Static type of **this** is `B2`.

Access Control Static Only

"Public" and "private" don't apply to dynamic types; it is possible to call methods in objects of types you can't name:

```
package utils;                                   | package mystuff;
/** A Set of things. */                          |
public interface Collector {                     | class User {
  void add(Object x);                            |    utils.Collector c =
}                                                |      utils.Utils.concat();
----------------------------                     |
package utils;                                   |    c.add("foo");  // OK
public class Utils {                             |    ... c.value(); // ERROR
  public static Collector concat() {             |    ((utils.Concatenator) c).value()
    return new Concatenator();                   |                     // ERROR
  }                                              |
}                                                ----------------------------------

/** NON-PUBLIC class that collects strings. */
class Concatenater implements Collector {
  StringBuffer stuff = new StringBuffer();
  int n = 0;
  public void add(Object x) { stuff.append(x); n += 1; }
  public Object value() { return stuff.toString(); }
}
```

Loose End #1: Importing

- Writing `java.util.List` every time you mean `List` or `java.lang.regex.Pattern` every time you mean `Pattern` is annoying.
- The purpose of the **import** clause at the beginning of a source file is to define abbreviations:
  - `import java.util.List;` means "within this file, you can use `List` as an abbreviation for `java.util.List`.
  - `import java.util.*;` means "within this file, you can use *any* class name in the package `java.util` without mentioning the package."
- Importing does *not* grant any special access; it *only* allows abbreviation.
- In effect, your program always contains `import java.lang.*;`

Loose End #2: Static importing

- One can easily get tired of writing `System.out` and `Math.sqrt`. Do you really need to be reminded with each use that `out` is in the `java.lang.System` package and that `sqrt` is in the Math package (duh)?
- Both examples are of *static* members. New feature of Java allows you to abbreviate such references:
  - `import static java.lang.System.out;` means "within this file, you can use `out` as an abbreviation for `System.out`.
  - `import static java.lang.System.*;` means "within this file, you can use *any* static member name in `System` without mentioning the package.
- Again, this is *only* an abbreviation. No special access.
- Alas, you can't do this for classes in the anonymous package.

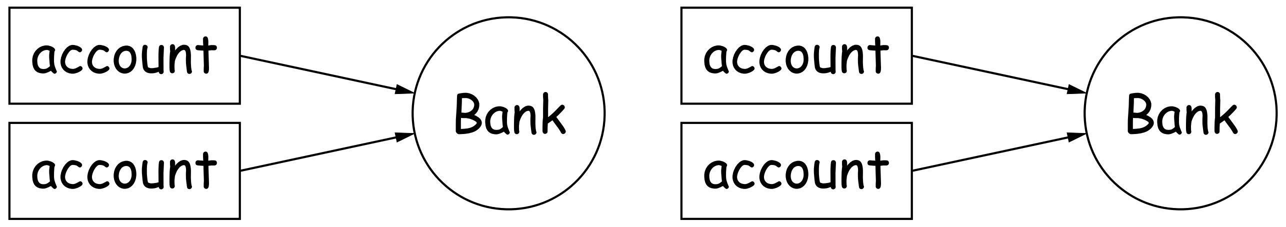
Loose End #3: Nesting Classes

- Sometimes, it makes sense to *nest* one class in another. The nested class might
  - be used only in the implementation of the other, or
  - be conceptually "subservient" to the other
- Nesting such classes can help avoid name clashes or "pollution of the name space" with names that will never be used anywhere else.
- Example: Polynomials can be thought of as sequences of terms. Terms aren't meaningful outside of Polynomials, so you might define a class to represent a term *inside* the Polynomial class:

```
class Polynomial {

  methods on polynomials

  private Term[] terms;
  private static class Term {
    ...
  }
}
```

Inner Classes

- Last slide showed a static nested class. Static nested classes are just like any other, except that they can be private or protected, and they can see private variables of the enclosing class.
- Non-static nested classes are called *inner classes.*
- Somewhat rare (and syntax is odd); used when each instance of the nested class is created by and naturally associated with an instance of the containing class, like Banks and Accounts:



```
class Bank {                                      | Bank e = new Bank(...);
  private void connectTo(...) {...}               | Bank.Account p0 =
  public class Account {                          |     e.new Account(...);
    public void call(int number) {                | Bank.Account p1 =
       Bank.this.connectTo(...); ...              |     e.new Account(...);
    } // Bank.this means "the bank that           |
  }   // created me"                              |
 }                                                |
```

Loose End #4: instanceof

- It is possible to ask about the dynamic type of something:

```
void typeChecker(Reader r) {
  if (r instanceof TrReader)
    System.out.print("Translated characters: ");
  else
    System.out.print("Characters: ");
  ...
}
```

- However, this is seldom what you want to do. Why do this:

```
if (x instanceof StringReader)
  read from  (StringReader) x;
else if (x instanceof FileReader)
  read from  (FileReader) x;
...
```

  when you can just call `x.read()`?!

- In general, use instance methods rather than **instanceof**.

CS61B Lecture #14: Integers

Integer Types and Literals

Type	Bits	Signed?	Literals
byte	8	Yes	Cast from **int**: (byte) 3
short	16	Yes	None. Cast from **int**: (short) 4096

char	16	No	`'a' // (char) 97`
			`'\n'  // newline ((char) 10)`
			`'\t'  // tab ((char) 8)`
			`'\\'  // backslash`
			`'A', '\101', '\u0041'  // == (char) 65`

int	32	Yes	`123`
			`0100  // Octal for 64`
			`0x3f, 0xffffffff // Hexadecimal 63, -1 (!)`

long	64	Yes	`123L, 01000L, 0x3fL`
			`1234567891011L`

- Negative numerals are just negated (positive) literals.
- "N bits" means that there are 2^N integers in the domain of the type:
  - If signed, range of values is $-2^{N-1} .. 2^{N-1} - 1$.
  - If unsigned, only non-negative numbers, and range is $0..2^N - 1$.

Overflow

- **Problem:** How do we handle overflow, such as occurs in `10000*10000*10000`?
- Some languages throw an exception (Ada), some give undefined results (C, C++)
- Java *defines* the result of any arithmetic operation or conversion on integer types to "wrap around"—*modular arithmetic.*
- That is, the "next number" after the largest in an integer type is the smallest (like "clock arithmetic").
- E.g., `(byte) 128 == (byte) (127+1) == (byte) -128`
- In general,
  - If the result of some arithmetic subexpression is supposed to have type T, an n-bit integer type,
  - then we compute the real (mathematical) value, x,
  - and yield a number, x', that is in the range of T, and that is equivalent to x modulo 2^n.
  - (That means that $x - x'$ is a multiple of 2^n.)

Modular Arithmetic

- Define $a \equiv b \pmod{n}$ to mean that $a - b = kn$ for some integer k.
- Define the binary operation $a \bmod n$ as the value b such that $a \equiv b \pmod{n}$ and $0 \le b < n$ for $n > 0$. (Can be extended to $n \le 0$ as well, but we won't bother with that here.) This is *not* the same as Java's % operation.
- Various facts: (Here, let a' denote $a \bmod n$).

$$\begin{aligned}
a'' &= a' \\
a' + b'' &= (a' + b)' = a + b' \\
(a' - b')' &= (a' + (-b)')' = (a - b)' \\
(a' \cdot b')' &= a' \cdot b' = a \cdot b' \\
(a^k)' &= ((a')^k)' = (a \cdot (a^{k-1})')', \text{ for } k > 0.
\end{aligned}$$

Modular Arithmetic: Examples

- `(byte) (64*8)` yields 0, since $512 - 0 = 2 \times 2^8$.
- `(byte) (64*2)` and `(byte) (127+1)` yield -128, since $128 - (-128) = 1 \times 2^8$.
- `(byte) (101*99)` yields 15, since $9999 - 15 = 39 \times \cdot 2^8$.
- `(byte) (-30*13)` yields 122, since $-390 - 122 = -2 \times 2^8$.
- `(char) (-1)` yields $2^{16} - 1$, since $-1 - (2^{16} - 1) = -1 \times 2^{16}$.

Modular Arithmetic and Bits

- Why wrap around?
- Java's definition is the natural one for a machine that uses binary arithmetic.
- For example, consider bytes (8 bits):

Decimal	Binary
101	1100101
$\times$99	1100011
9999	100111|00001111
$-$ 9984	100111|00000000
15	00001111

- In general, bit n, counting from 0 at the right, corresponds to 2^n.
- The bits to the left of the vertical bars therefore represent multiples of $2^8 = 256$.
- So throwing them away is the same as arithmetic modulo 256.

Negative numbers

- Why this representation for -1?

$$\begin{array}{rr|l} & 1 & 00000001_2 \\ + & -1 & 11111111_2 \\ = & 0 & 1|00000000_2 \end{array}$$

  Only 8 bits in a byte, so bit 8 falls off, leaving 0.

- The truncated bit is in the 2^8 place, so throwing it away gives an equal number modulo 2^8. All bits to the left of it are also divisible by 2^8.

- On unsigned types (**char**), arithmetic is the same, but we choose to represent only non-negative numbers modulo 2^{16}:

$$\begin{array}{rr|l} & 1 & 0000000000000001_2 \\ + & 2^{16}-1 & 1111111111111111_2 \\ = & 2^{16}+0 & 1|0000000000000000_2 \end{array}$$

Conversion

- In general Java will silently convert from one type to another if this makes sense and no information is lost from value.
- Otherwise, cast explicitly, as in `(byte) x`.
- Hence, given

```
byte aByte; char aChar; short aShort; int anInt; long aLong;

// OK:
aShort = aByte; anInt = aByte; anInt = aShort;
anInt = aChar; aLong = anInt;

// Not OK, might lose information:
anInt = aLong; aByte = anInt; aChar = anInt; aShort = anInt;
aShort = aChar; aChar = aShort; aChar = aByte;

// OK by special dispensation:
aByte = 13;    // 13 is compile-time constant
aByte = 12+100 // 112 is compile-time constant
```

Promotion

- Arithmetic operations (+, *, . . .) *promote* operands as needed.
- Promotion is just implicit conversion.
- For integer operations,
  - if any operand is **long**, promote both to **long**.
  - otherwise promote both to **int**.
- So,

```
aByte + 3 == (int) aByte + 3    // Type int
aLong + 3 == aLong + (long) 3   // Type long
'A' + 2 == (int) 'A' + 2        // Type int
aByte = aByte + 1               // ILLEGAL (why?)
```

- But fortunately,

```
aByte += 1;      // Defined as aByte = (byte) (aByte+1)
```

- Common example:

```
// Assume aChar is an upper-case letter
char lowerCaseChar = (char) ('a' + aChar - 'A'); // why cast?
```

Bit twiddling

- Java (and C, C++) allow for handling integer types as sequences of bits. No “conversion to bits” needed: they already are.
- Operations and their uses:

Mask	Set	Flip	Flip all
00101100	00101100	00101100	
& 10100111	| 10100111	^ 10100111	~ 10100111
00100100	10101111	10001011	01011000

- Shifting:

Left	Arithmetic Right	Logical Right
10101101 << 3	10101101 >> 3	10101100 >>> 3
01101000	11110101	00010101

- What is:
  - (-1) >>> 29?
  - x << n?
  - x >> n?
  - (x >>> 3) & ((1<<5)-1)?

Bit twiddling

- Java (and C, C++) allow for handling integer types as sequences of bits. No "conversion to bits" needed: they already are.
- Operations and their uses:

Mask	Set	Flip	Flip all
00101100	00101100	00101100	
& 10100111	| 10100111	^ 10100111	~ 10100111
00100100	10101111	10001011	01011000

- Shifting:

Left	Arithmetic Right	Logical Right
10101101 << 3	10101101 >> 3	10101100 >>> 3
01101000	11110101	00010101

- What is:

(-1) >>> 29?	$= 7$.
x << n?	
x >> n?	
(x >>> 3) & ((1<<5)-1)?	

Bit twiddling

- Java (and C, C++) allow for handling integer types as sequences of bits. No "conversion to bits" needed: they already are.
- Operations and their uses:

Mask	Set	Flip	Flip all
`  00101100`	`  00101100`	`  00101100`	
`& 10100111`	`| 10100111`	`^ 10100111`	`~ 10100111`
`  00100100`	`  10101111`	`  10001011`	`  01011000`

- Shifting:

Left	Arithmetic Right	Logical Right
`10101101 << 3`	`10101101 >>  3`	`10101100 >>> 3`
`01101000`	`11110101`	`00010101`

- What is:

`(-1) >>> 29`?	$= 7.$
x `<<` n?	$= x \cdot 2^n.$
x `>>` n?	
(x `>>> 3) & ((1<<5)-1)`?	

Bit twiddling

- Java (and C, C++) allow for handling integer types as sequences of bits. No "conversion to bits" needed: they already are.
- Operations and their uses:

Mask	Set	Flip	Flip all
00101100	00101100	00101100	
& 10100111	| 10100111	^ 10100111	~ 10100111
00100100	10101111	10001011	01011000

- Shifting:

Left	Arithmetic Right	Logical Right
10101101 << 3	10101101 >> 3	10101100 >>> 3
01101000	11110101	00010101

- What is:

`(-1) >>> 29`?	$= 7.$
x `<<` n?	$= x \cdot 2^n.$
x `>>` n?	$= \lfloor x/2^n \rfloor$ (i.e., rounded down).
(x `>>> 3) & ((1<<5)-1)`?	

Bit twiddling

- Java (and C, C++) allow for handling integer types as sequences of bits. No "conversion to bits" needed: they already are.
- Operations and their uses:

Mask	**Set**	**Flip**	**Flip all**
`00101100`	`00101100`	`00101100`	
`& 10100111`	`| 10100111`	`^ 10100111`	`~ 10100111`
`00100100`	`10101111`	`10001011`	`01011000`

- Shifting:

Left	**Arithmetic Right**	**Logical Right**
`10101101 << 3`	`10101101 >>  3`	`10101100 >>> 3`
`01101000`	`11110101`	`00010101`

- What is:

`(-1) >>> 29`?	$= 7$.
x `<<` n?	$= x \cdot 2^n$.
x `>>` n?	$= \lfloor x/2^n \rfloor$ (i.e., rounded down).
`(`x` >>> 3) & ((1<<5)-1)`?	5-bit integer, bits 3–7 of x.

CS61B Lecture #16: Complexity

What Are the Questions?

- Cost is a principal concern throughout engineering:

  "An engineer is someone who can do for a dime what any fool can do for a dollar."

- Cost can mean
  - *Operational cost* (for programs, time to run, space requirements).
  - *Development costs*: How much engineering time? When delivered?
  - *Maintenance costs*: Upgrades, bug fixes.
  - *Costs of failure*: How robust? How safe?
- Is this program *fast enough*? Depends on:
  - *For what purpose;*
  - *For what input data.*
- How much *space* (memory, disk space)?
  - Again depends on what input data.
- How will it *scale*, as input gets big?

Enlightening Example

Problem: Scan a text corpus (say 10^8 bytes or so), and find and print the 20 most frequently used words, together with counts of how often they occur.

- Solution 1 (Knuth): Heavy-Duty data structures
  - Hash Trie implementation, randomized placement, pointers galore, several pages long.
- Solution 2 (Doug McIlroy): UNIX shell script:

```
tr -c -s '[:alpha:]' '[\n*]' < FILE | \
sort | \
uniq -c | \
sort -n -r -k 1,1 | \
sed 20q
```

- Which is better?
  - #1 is much faster,
  - but #2 took 5 minutes to write and processes 100MB in ≈ 50 sec.
  - I pick #2.
- In very many cases, almost anything will do: Keep It Simple.

Cost Measures (Time)

- *Wall-clock or execution* time
  - You can do this at home:

    ```
    time java FindPrimes 1000
    ```

  - Advantages: easy to measure, meaning is obvious.
  - Appropriate where time is critical (real-time systems, e.g.).
  - Disadvantages: applies only to specific data set, compiler, machine, etc.
- *Dynamic statement counts* of # of times statements are executed:
  - Advantages: more general (not sensitive to speed of machine).
  - Disadvantages: doesn't tell you actual time, still applies only to specific data sets.
- *Symbolic execution times:*
  - That is, *formulas* for execution times as functions of input size.
  - Advantages: applies to all inputs, makes scaling clear.
  - Disadvantage: practical formula must be approximate, may tell very little about actual time.

Asymptotic Cost

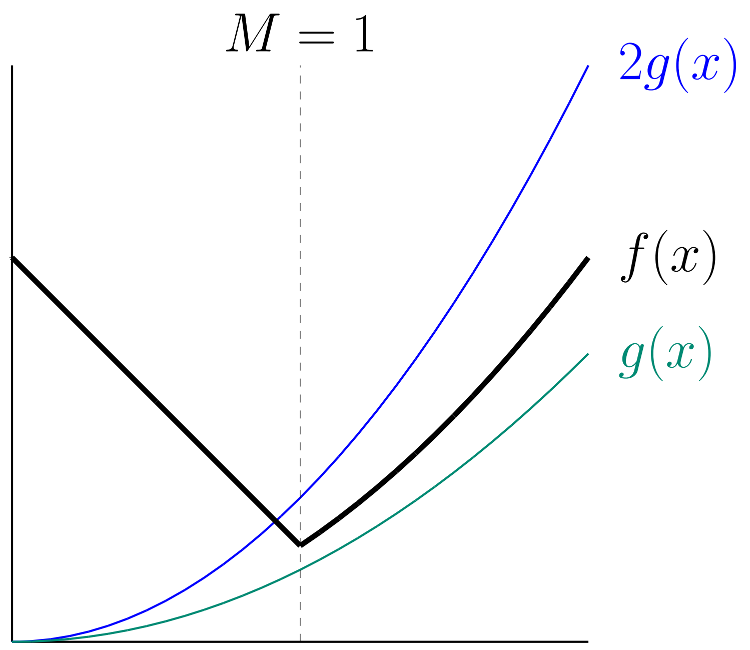
- Symbolic execution time lets us see *shape* of the cost function.
- Since we are approximating anyway, pointless to be precise about certain things:
  - *Behavior on small inputs*:
    - Can always pre-calculate some results.
    - Times for small inputs not usually important.
    - Often more interested in *asymptotic behavior* as input size becomes very large.
  - *Constant factors* (as in "off by factor of 2"):
    - Just changing machines causes constant-factor change.
- How to abstract away from (i.e., ignore) these things?

Handy Tool: Order Notation

- Idea: Don't try to produce specific functions that specify size, but rather *families of functions with similarly behaved magnitudes*.
- Then say something like "f is bounded by g if it is in g's family."
- For any function $g(x)$, the functions $2g(x)$, $0.5g(x)$, or for any $K > 0$, $K \cdot g(x)$, all have the same "shape". So put all of them into g's family.
- Any function $h(x)$ such that $h(x) = K \cdot g(x)$ for $x > M$ (for some constant M) has g's shape "except for small values." So put all of these in g's family.
- For upper limits, throw in all functions whose absolute value is everywhere $\leq$ some member of g's family. Call this set $O(g)$ or $O(g(n))$.
- Or, for lower limits, throw in all functions whose absolute values is everywhere $\geq$ some member of g's family. Call this set $\Omega(g)$.
- Finally, define $\Theta(g) = O(g) \cap \Omega(g)$—the set of functions *bracketed in magnitude by* two members of g's family.

Big Oh

- Goal: Specify bounding from above.



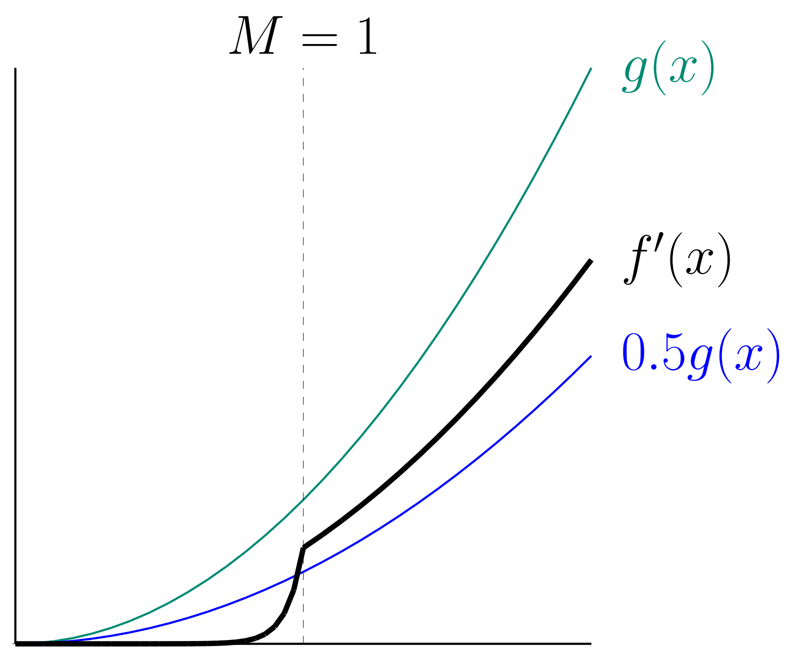
- Here, $f(x) \le 2g(x)$ as long as $x > 1$,
- So $f(x)$ is in g's "bounded-above family," written

$$f(x) \in O(g(x)),$$

- ...even though (in this case) $f(x) > g(x)$ everywhere.

Big Omega

- Goal: Specify bounding from below:



- Here, $f'(x) \geq \frac{1}{2}g(x)$ as long as $x > 1$,
- So $f'(x)$ is in g's "bounded-below family," written

$$f'(x) \in \Omega(g(x)),$$

- . . . even though $f(x) < g(x)$ everywhere.

Big Theta

- In the two previous slides, we not only have $f(x) \in O(g(x))$ and $f'(x) \in \Omega(g(x))$,...
- ...but also $f(x) \in \Omega(g(x))$ and $f'(x) \in O(g(x))$.
- We can summarize this all by saying $f(x) \in \Theta(g(x))$ and $f'(x) \in \Theta(g(x))$.

Aside: Various Mathematical Pedantry

- Technically, if I am going to talk about $O(\cdot)$, $\Omega(\cdot)$ and $\Theta(\cdot)$ as sets of functions, I really should write, for example,

$$f \in O(g) \quad \text{instead of} \quad f(x) \in O(g(x))$$

- In effect, $f(x) \in O(g(x))$ is short for $\lambda\, x.\; f(x) \in O(\lambda\, x.\; g(x))$.
- The standard notation outside this course, in fact, is $f(x) = O(g(x))$, but personally, I think that's a serious abuse of notation.

How We Use Order Notation

- Elsewhere in mathematics, you'll see $O(\dots)$, etc., used generally to specify bounds on functions.
- For example,

$$\pi(N) = \Theta(\frac{N}{\ln N})$$

which I would prefer to write

$$\pi(N) \in \Theta(\frac{N}{\ln N})$$

(Here, $\pi(N)$ is the number of primes less than or equal to N.)

- Also, you'll see things like

$$f(x) = x^3 + x^2 + O(x) \qquad \text{(or } f(x) \in x^4 + x^2 + O(x)\text{)},$$

meaning that $f(x) = x^3 + x^2 + g(x)$ where $g(x) \in O(x)$.

- For our purposes, the functions we will be bounding will be *cost functions:* functions that measure the amount of execution time or the amount of space required by a program or algorithm.

Why It Matters

- Computer scientists often talk as if constant factors didn't matter at all, only the difference of $\Theta(N)$ vs. $\Theta(N^2)$.
- In reality they do matter, but at some point, constants always get swamped.

n	$16\lg n$	$\sqrt{n}$	n	$n\lg n$	n^2	n^3	2^n
2	16	1.4	2	2	4	8	4
4	32	2	4	8	16	64	16
8	48	2.8	8	24	64	512	256
16	64	4	16	64	256	$4,096$	$65,636$
32	80	5.7	32	160	1024	$32,768$	4.2×10^9
64	96	8	64	384	$4,096$	$262,144$	1.8×10^{19}
128	112	11	128	896	$16,384$	2.1×10^9	3.4×10^{38}
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
$1,024$	160	32	$1,024$	$10,240$	1.0×10^6	1.1×10^9	1.8×10^{308}
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
2^{20}	320	1024	1.0×10^6	2.1×10^7	1.1×10^{12}	1.2×10^{18}	$6.7\times 10^{315,652}$

Some Intuition on Meaning of Growth

- How big a problem can you solve in a given time?
- In the following table, left column shows time in microseconds to solve a given problem as a function of problem size N.
- Entries show the *size of problem* that can be solved in a second, hour, month (31 days), and century, for various relationships between time required and problem size.
- $N =$ problem size.

Time (μsec) for	**Max N Possible in**			
problem size N	1 second	1 hour	1 month	1 century
$\lg N$	10^{300000}	$10^{1000000000}$	$10^{8\cdot 10^{11}}$	$10^{10^{14}}$
N	10^6	$3.6 \cdot 10^9$	$2.7 \cdot 10^{12}$	$3.2 \cdot 10^{15}$
$N \lg N$	63000	$1.3 \cdot 10^8$	$7.4 \cdot 10^{10}$	$6.9 \cdot 10^{13}$
N^2	1000	60000	$1.6 \cdot 10^6$	$5.6 \cdot 10^7$
N^3	100	1500	14000	150000
2^N	20	32	41	51

Using the Notation

- Can use this order notation for any kind of real-valued function.
- We will use them to describe cost functions. Example:

```
/** Find position of X in list L, or -1 if not found. */
int find(List L, Object X) {
   int c;
   for (c = 0; L != null; L = L.next, c += 1)
      if (X.equals(L.head)) return c;
   return -1;
}
```

- Choose representative operation: number of `.equals` tests.
- If N is length of L, then loop does *at most* N tests: *worst-case time* is N tests.
- In fact, total # of instructions executed is roughly proportional to N in the worst case, so can also say worst-case time is $O(N)$, regardless of units used to measure.
- Use $N > M$ provision (in defn. of $O(\cdot)$) to ignore empty list.

Be Careful

- It's also true that the worst-case time is $O(N^2)$, since $N \in O(N^2)$ also: Big-Oh bounds are loose.
- The worst-case time is $\Omega(N)$, since $N \in \Omega(N)$, but that does *not* mean that the loop *always* takes time N, or even $K \cdot N$ for some K.
- Instead, we are just saying something about the *function* that maps N into the *largest possible* time required to process any array of length N.
- To say as much as possible about our worst-case time, we should try to give a Θ bound: in this case, we can: $\Theta(N)$.
- But again, that still tells us nothing about *best-case* time, which happens when we find `X` at the beginning of the loop. Best-case time is $\Theta(1)$.

Effect of Nested Loops

- Nested loops often lead to polynomial bounds:

```
for (int i = 0; i < A.length; i += 1)
   for (int j = 0; j < A.length; j += 1)
      if (i != j && A[i] == A[j])
         return true;
return false;
```

- Clearly, time is $O(N^2)$, where $N =$ `A.length`. *Worst-case time* is $\Theta(N^2)$.
- Loop is inefficient though:

```
for (int i = 0; i < A.length; i += 1)
   for (int j = i+1; j < A.length; j += 1)
      if (A[i] == A[j]) return true;
return false;
```

- Now worst-case time is proportional to

$$N - 1 + N - 2 + \ldots + 1 = N(N-1)/2 \in \Theta(N^2)$$

(so asymptotic time unchanged by the constant factor).

Recursion and Recurrences: Fast Growth

- Silly example of recursion. In the worst case, both recursive calls happen:

```
/** True iff X is a substring of S */
boolean occurs(String S, String X) {
  if (S.equals(X)) return true;
  if (S.length() <= X.length()) return false;
  return
    occurs(S.substring(1), X) ||
    occurs(S.substring(0, S.length()-1), X);
}
```

- Define $C(N)$ to be the worst-case cost of `occurs(S,X)` for `S` of length N, `X` of fixed size N_0, measured in # of calls to `occurs`. Then

$$C(N) = \begin{cases} 1, & \text{if } N \leq N_0, \\ 2C(N-1)+1 & \text{if } N > N_0 \end{cases}$$

- So $C(N)$ grows exponentially:

$$\begin{aligned} C(N) &= 2C(N-1)+1 = 2(2C(N-2)+1)+1 = \ldots = \underbrace{2(\cdots 2}_{N-N_0}\cdot 1+1)+\ldots+1 \\ &= 2^{N-N_0}+2^{N-N_0-1}+2^{N-N_0-2}+\ldots+1 = 2^{N-N_0+1}-1 \in \Theta(2^N) \end{aligned}$$

Binary Search: Slow Growth

```
/** True X iff is an element of S[L .. U]. Assumes
 *  S in ascending order, 0 <= L <= U-1 < S.length. */
boolean isIn(String X, String[] S, int L, int U) {
  if (L > U) return false;
  int M = (L+U)/2;
  int direct = X.compareTo(S[M]);
  if (direct < 0) return isIn(X, S, L, M-1);
  else if (direct > 0) return isIn(X, S, M+1, U);
  else return true;
}
```

- Here, worst-case time, $C(D)$, (as measured by # of calls to `.compareTo`), depends on size $D = U - L + 1$.
- We eliminate `S[M]` from consideration each time and look at half the rest. Assume $D = 2^k - 1$ for simplicity, so:

$$\begin{aligned} C(D) &= \begin{cases} 0, & \text{if } D \leq 0, \\ 1 + C((D-1)/2), & \text{if } D > 0. \end{cases} \\ &= \underbrace{1 + 1 + \ldots + 1}_{k} + 0 \\ &= k = \lg(D+1) \in \Theta(\lg D) \end{aligned}$$

Another Typical Pattern: Merge Sort

```
List sort(List L) {
  if (L.length() < 2) return L;
  Split L into L0 and L1 of about equal size;
  L0 = sort(L0);  L1 = sort(L1);
  return Merge of L0 and L1
}
```

Merge ("combine into a single ordered list") takes time proportional to size of its result.

- Assuming that size of L is $N = 2^k$, worst-case cost function, $C(N)$, counting just merge time (which is proportional to # items merged):

$$\begin{aligned} C(N) &= \begin{cases} 0, & \text{if } N < 2; \\ 2C(N/2) + N, & \text{if } N \geq 2. \end{cases} \\ &= 2(2C(N/4) + N/2) + N \\ &= 4C(N/4) + N + N \\ &= 8C(N/8) + N + N + N \\ &= N \cdot 0 + \underbrace{N + N + \ldots + N}_{k = \lg N} \\ &= N \lg N \end{aligned}$$

- In general, can say it's $\Theta(N \lg N)$ for arbitrary N (not just 2^k).

CS61B Lecture #17

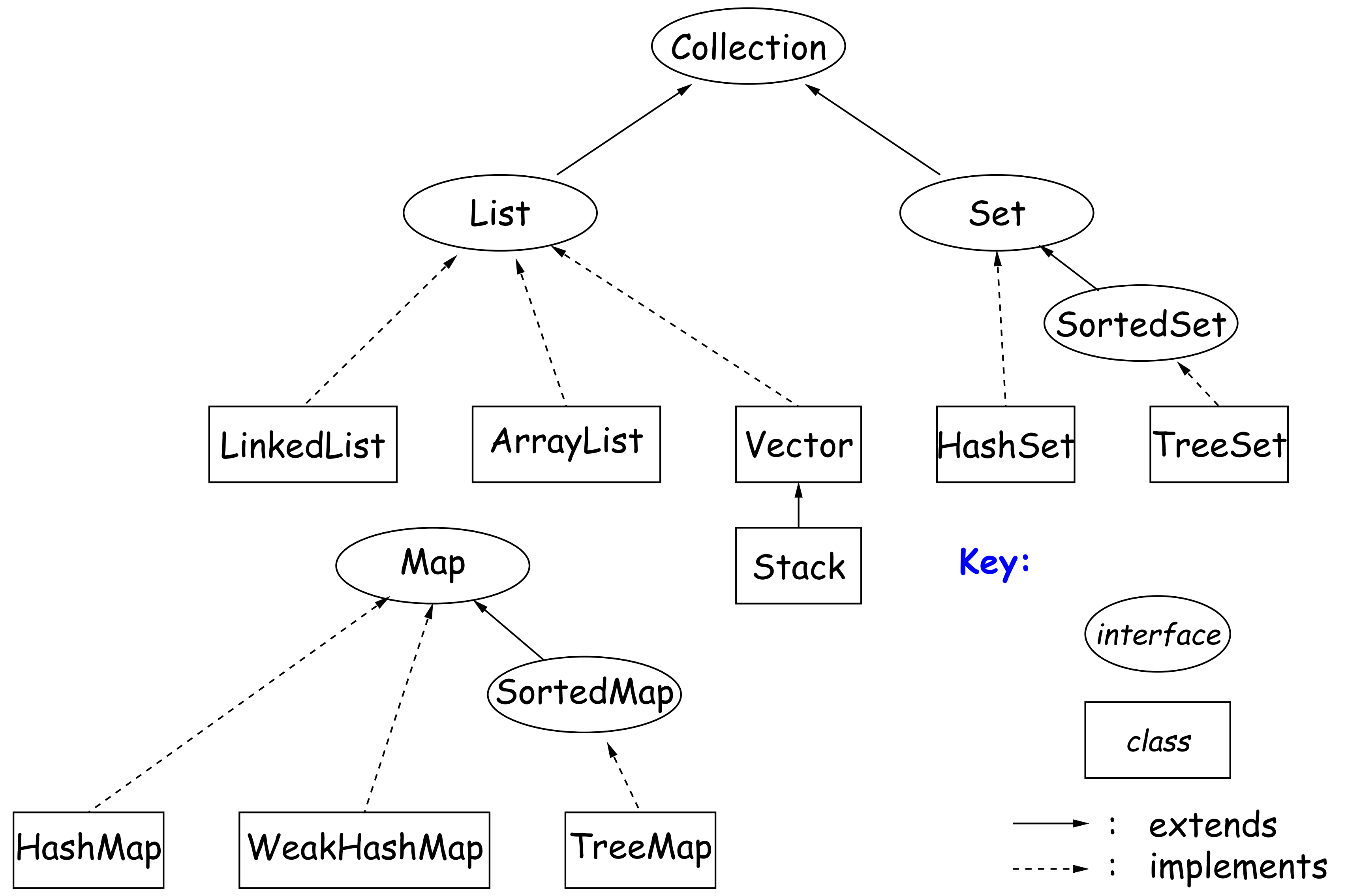
Topics

- Overview of standard Java Collections classes.
  - Iterators, ListIterators
  - Containers and maps in the abstract
- Amortized analysis of implementing lists with arrays.

Data Types in the Abstract

- Most of the time, should *not* worry about implementation of data structures, search, etc.
- What they do for us—their specification—is important.
- Java has several standard types (in `java.util`) to represent collections of objects
  - Six interfaces:
    - `Collection`: General collections of items.
    - `List`: Indexed sequences with duplication
    - `Set`, `SortedSet`: Collections without duplication
    - `Map`, `SortedMap`: Dictionaries (key $\mapsto$ value)
  - Concrete classes that provide actual instances: `LinkedList`, `ArrayList`, `HashSet`, `TreeSet`.
  - To make change easier, purists would use the concrete types only for **new**, interfaces for parameter types, local variables.

Collection Structures in java.util



The Collection Interface

- Collection interface. Main functions promised:
  - Membership tests: `contains` ($\in$), `containsAll` ($\subseteq$)
  - Other queries: `size`, `isEmpty`
  - Retrieval: `iterator`, `toArray`
  - *Optional* modifiers: `add`, `addAll`, `clear`, `remove`, `removeAll` (set difference), `retainAll` (intersect)

Side Trip about Library Design: Optional Operations

- Not all `Collections` need to be modifiable; often makes sense just to get things from them.
- So some operations are optional (`add`, `addAll`, `clear`, `remove`, `removeAll`, `retainAll`)
- The library developers decided to have *all* Collections implement these, but allowed implementations to throw an exception:

  `UnsupportedOperationException`

- An alternative design would have created separate interfaces:

```
interface Collection { contains, containsAll, size, iterator, ... }
interface Expandable extends Collection { add, addAll }
interface Shrinkable extends Collection { remove, removeAll, ... }
interface ModifiableCollection
   extends Collection, Expandable, Shrinkable { }
```

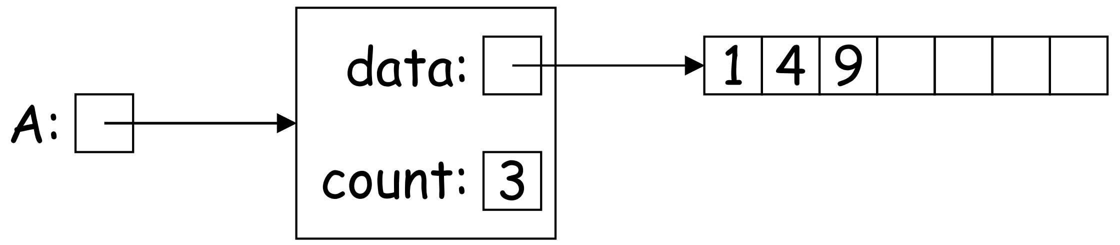
- You'd soon have lots of interfaces. Perhaps that's why they didn't do it that way.

The List Interface

- Extends `Collection`
- Intended to represent *indexed sequences* (generalized arrays)
- Adds new methods to those of `Collection`:
  - Membership tests: `indexOf`, `lastIndexOf`.
  - Retrieval: `get(`i`)`, `listIterator()`, `sublist(`B, E`)`.
  - Modifiers: `add` and `addAll` with additional index to say *where* to add. Likewise for removal operations. `set` operation to go with `get`.
- Type `ListIterator<Item>` extends `Iterator<Item>`:
  - Adds `previous` and `hasPrevious`.
  - `add`, `remove`, and `set` allow one to iterate through a list, inserting, removing, or changing as you go.
  - **Important Question:** What advantage is there to saying `List L` rather than `LinkedList L` or `ArrayList L`?

Implementing Lists (I): ArrayLists

- The main concrete types in Java library for interface `List` are `ArrayList` and `LinkedList`:
- As you might expect, an `ArrayList`, `A`, uses an array to hold data. For example, a list containing the three items 1, 4, and 9 might be represented like this:



- After adding four more items to `A`, its `data` array will be full, and the value of `data` will have to be replaced with a pointer to a new, bigger array that starts with a copy of its previous values.
- Question: For best performance, how big should this new array be?
- If we increase the size by 1 each time it gets full (or by any constant value), the cost of N additions will scale as $\Theta(N^2)$, which makes `ArrayList` look much worse than `LinkedList` (which uses an `IntList`-like implementation.)

Expanding Vectors Efficiently

- When using array for expanding sequence, best to *double* the size of array to grow it. Here's why.
- If array is size s, doubling its size and moving s elements to the new array takes time proportional to $2s$.
- In all cases, there is an additional $\Theta(1)$ cost for each addition to account for actually assigning the new value into the array.
- When you add up these costs for inserting a sequence of N items, the *total* cost turns out to be proportional to N, as if each addition took constant time, even though some of the additions actually take time proportional to N all by themselves!

Amortized Time

- Suppose that the actual costs of a sequence of N operations are $c_0, c_1, \ldots, c_{N-1}$, which may differ from each other by arbitrary amounts and where $c_i \in O(f(i))$.
- Consider another sequence $a_0, a_1, \ldots, a_{N-1}$, where $a_i \in O(g(i))$.
- If
  $$\sum_{0 \le i < k} a_i \ge \sum_{0 \le i < k} c_i \textit{ for all } k,$$
  we say that the operations all run in $O(g(i))$ *amortized time*.
- That is, the actual cost of a given operation, c_i, may be arbitrarily larger than the amortized time, a_i, as long as the *total* amortized time is always greater than or equal to the total actual time, no matter where the sequence of operations stops—i.e., no matter what k is.
- In cases of interest, the amortized time bounds are much less than the actual individual time bounds: $g(i) \ll f(i)$.
- E.g., for the case of insertion with array doubling, $f(i) \in O(N)$ and $g(i) \in O(1)$.

Amortization: Expanding Vectors (II)

To Insert Item #	Resizing Cost	Cumulative Cost	Resizing Cost per Item	Array Size After Insertions
0	0	0	0	1
1	2	2	1	2
2	4	6	2	4
3	0	6	1.5	4
4	8	14	2.8	8
5	0	14	2.33	8
⋮	⋮	⋮	⋮	⋮
7	0	14	1.75	8
8	16	30	3.33	16
⋮	⋮	⋮	⋮	⋮
15	0	30	1.88	16
⋮	⋮	⋮	⋮	⋮
2^m+1 to $2^{m+1}-1$	0	$2^{m+2}-2$	≈ 2	2^{m+1}
2^{m+1}	2^{m+2}	$2^{m+3}-2$	≈ 4	2^{m+2}

- If we spread out (*amortize*) the cost of resizing, we average at most about 4 time units for resizing on each item: "amortized resizing time is 4 units." Time to add N elements is $\Theta(N)$, *not* $\Theta(N^2)$.

Demonstrating Amortized Time: Potential Method

- To formalize the argument, associate a *potential, $\Phi_i \geq 0$,* to the ith operation that keeps track of "saved up" time from cheap operations that we can "spend" on later expensive ones. Start with $\Phi_0 = 0$.
- Now we pretend that the cost of the ith operation is actually a_i, the *amortized cost*, defined

$$a_i = c_i + \Phi_{i+1} - \Phi_i,$$

  where c_i is the real cost of the operation. Or, looking at potential:

$$\Phi_{i+1} = \Phi_i + (a_i - c_i)$$

- On cheap operations, we artificially set $a_i > c_i$ so that we can increase Φ ($\Phi_{i+1} > \Phi_i$).
- On expensive ones, we typically have $a_i \ll c_i$ and greatly decrease Φ (but don't let it go negative—may not be "overdrawn").
- We try to do all this so that a_i remains as we desired (e.g., $O(1)$ for expanding array), without allowing $\Phi_i < 0$.
- Requires that we choose a_i so that Φ_i always stays ahead of c_i.

Application to Expanding Arrays

- When adding to our array, the cost, c_i, of adding element #i when the array already has space for it is 1 unit.
- The array does not initially have space when adding items 1, 2, 4, 8, 16,...—in other words at item 2^n for all $n \geq 0$. So,
  - $c_i = 1$ if $i \geq 0$ and is not a power of 2; and
  - $c_i = 2i + 1$ when i is a power of 2 (copy i items, clear another i items, and then add item #i).
- So on each operation #2^n we're going to need to have saved up at least $2{\cdot}2^n = 2^{n+1}$ units of potential to cover the expense of expanding the array, and we have this operation and the preceding $2^{n-1} - 1$ operations in which to save up this much potential (everything since the preceding doubling operation).
- So choose a_0 = 1 and $a_i = 5$ for $i > 0$. Apply $\Phi_{i+1} = \Phi_i + (a_i - c_i)$, and here is what happens:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
c_i	1	3	5	1	9	1	1	1	17	1	1	1	1	1	1	1	33	1
a_i	1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
Φ_i	0	0	2	2	6	2	6	10	14	2	6	10	14	18	22	26	30	2

Pretending each cost is 5 never underestimates true cumulative time.

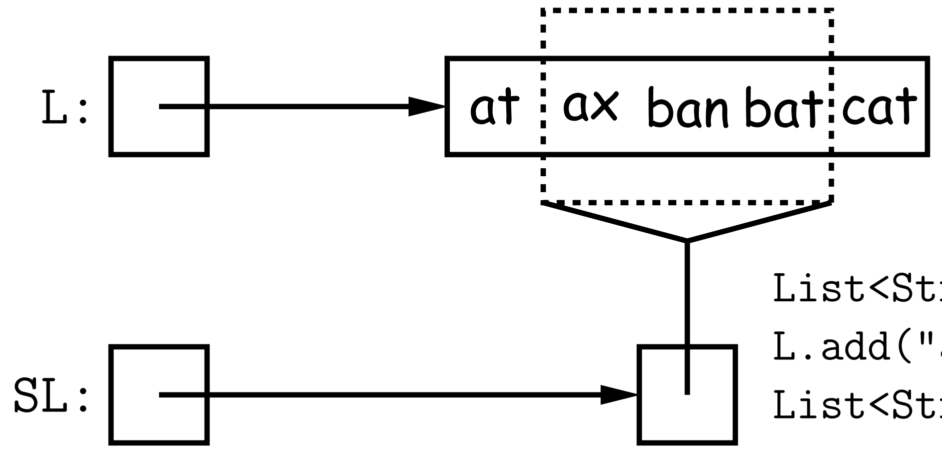
CS61B Lecture #18: Assorted Topics

- Views
- Maps
- More partial implementations
- Array vs. linked: tradeoffs
- Sentinels
- Specialized sequences: stacks, queues, deques
- Circular buffering
- Recursion and stacks
- Adapters

Views

New Concept: A *view* is an alternative presentation of (interface to) an existing object.

- For example, the `sublist` method is supposed to yield a "view of" part of an existing list:



```
List<String> L = new ArrayList<String>();
L.add("at"); L.add("ax"); ...
List<String> SL = L.sublist(1,4);
```

- Example: after `L.set(2, "bag")`, value of `SL.get(1)` is `"bag"`, and after `SL.set(1,"bad")`, value of `L.get(2)` is `"bad"`.
- Example: after `SL.clear()`, `L` will contain only `"at"` and `"cat"`.
- Small challenge: "How do they *do* that?!"

Maps

- A `Map` is a kind of "modifiable function:"

```
package java.util;
public interface Map<Key,Value> {
  Value get(Object key);            // Value at KEY.
  Object put(Key key, Value value); // Set get(KEY) -> VALUE
  ...
}
------------------------------------------------------
Map<String,String> f = new TreeMap<String,String>();
f.put("Paul", "George"); f.put("George", "Martin");
f.put("Dana", "John");
// Now f.get("Paul").equals("George")
//     f.get("Dana").equals("John")
//     f.get("Tom") == null
```

Map Views

```
public interface Map<Key,Value> { // Continuation

           /* Views of Maps */

  /** The set of all keys. */
  Set<Key> keySet();

  /** The multiset of all values that can be returned by get.
   *  (A multiset is a collection that may have duplicates). */
  Collection<Value> values();

  /** The set of all(key, value) pairs */
  Set<Map.Entry<Key,Value>> entrySet();
}
```

View Examples

Using example from a previous slide:

```
Map<String,String> f = new TreeMap<String,String>();
f.put("Paul", "George"); f.put("George", "Martin");
f.put("Dana", "John");
```

we can take various views of `f`:

```
for (Iterator<String> i = f.keySet().iterator(); i.hasNext();)
   i.next()  ===>  Dana, George, Paul
// or, more succinctly:
for (String name : f.keySet())
   name  ===>  Dana, George, Paul

for (String parent : f.values())
   parent  ===>   John, Martin, George

for (Map.Entry<String,String> pair : f.entrySet())
   pair   ===>   (Dana,John), (George,Martin), (Paul,George)

f.keySet().remove("Dana");   // Now f.get("Dana") == null
```

Simple Banking I: Accounts

Problem: Want a simple banking system. Can look up accounts by name or number, deposit or withdraw, print.

Account Structure

```
class Account {
  Account(String name, String number, int init) {
     this.name = name; this.number = number;
     this.balance = init;
  }
  /** Account-holder's name */
  final String name;
  /** Account number */
  final String number;
  /** Current balance */
  int balance;

  /** Print THIS on STR in some useful format. */
  void print(PrintStream str) { ... }
}
```

Simple Banking II: Banks

```
class Bank {
  /* These variables maintain mappings of String -> Account.  They keep
   * the set of keys (Strings) in "compareTo" order, and the set of
   * values (Accounts) is ordered according to the corresponding keys. */
  SortedMap<String,Account> accounts = new TreeMap<String,Account>();
  SortedMap<String,Account> names = new TreeMap<String,Account>();

  void openAccount(String name, int initBalance) {
     Account acc =
       new Account(name, chooseNumber(), initBalance);
     accounts.put(acc.number, acc);
     names.put(name, acc);
  }

  void deposit(String number, int amount) {
    Account acc = accounts.get(number);
    if (acc == null) ERROR(...);
    acc.balance += amount;
  }
  // Likewise for withdraw.
```

Banks (continued): Iterating

Printing out Account Data

```
/** Print out all accounts sorted by number on STR. */
void printByAccount(PrintStream str) {
   // accounts.values() is the set of mapped-to values.  Its
   // iterator produces elements in order of the corresponding keys.
   for (Account account : accounts.values())
     account.print(str);
}

/** Print out all bank accounts sorted by name on STR. */
void printByName(PrintStream str) {
   for (Account account : names.values())
     account.print(str);
}
```

A Design Question: What would be an appropriate representation for keeping a record of all transactions (deposits and withdrawals) against each account?

Partial Implementations

- Besides interfaces (like `List`) and concrete types (like `LinkedList`), Java library provides abstract classes such as `AbstractList`.
- Idea is to take advantage of the fact that operations are related to each other.
- Example: once you know how to do `get(k)` and `size()` for an implementation of `List`, you can implement all the other methods needed for a *read-only* list (and its iterators).
- Now throw in `add(k,x)` and you have all you need for the additional operations of a growable list.
- Add `set(k,x)` and `remove(k)` and you can implement everything else.

Example: The java.util.AbstractList helper class

```
public abstract class AbstractList<Item> implements List<Item>
{
   /** Inherited from List */
   // public abstract int size();
   // public abstract Item get(int k);
   public boolean contains(Object x) {
      for (int i = 0; i < size(); i += 1) {
        if ((x == null && get(i) == null) ||
            (x != null && x.equals(get(i))))
          return true;
      }
      return false;
   }
   /* OPTIONAL: Throws exception; override to do more. */
   void add(int k, Item x) {
     throw new UnsupportedOperationException();
   }
```
Likewise for remove, set

Example, continued: AListIterator

```
// Continuing abstract class AbstractList<Item>:
public Iterator<Item> iterator() { return listIterator(); }
public ListIterator<Item> listIterator() {
   return new AListIterator(this);
}

private static class AListIterator implements ListIterator<Item> {
   AbstractList<Item> myList;
   AListIterator(AbstractList<Item> L) { myList = L; }
   /** Current position in our list. */
   int where = 0;

   public boolean hasNext() { return where < myList.size(); }
   public Item next() { where += 1; return myList.get(where-1); }
   public void add(Item x) { myList.add(where, x); where += 1; }
   ... previous, remove, set, etc.
}
...
```

Aside: Another way to do AListIterator

It's also possible to make the nested class non-static:

```
public Iterator<Item> iterator() { return listIterator(); }
public ListIterator<Item> listIterator() { return this.new AListIterator(); }

private class AListIterator implements ListIterator<Item> {
  /** Current position in our list. */
  int where = 0;

  public boolean hasNext() { return where < AbstractList.this.size(); }
  public Item next() { where += 1; return AbstractList.this.get(where-1); }
  public void add(Item x) { AbstractList.this.add(where, x); where += 1; }
  ... previous, remove, set, etc.
}
...
```

- Here, `AbstractList.this` means "the `AbstractList` I am attached to" and X`.new AListIterator` means "create a new `AListIterator` that is attached to X."
- In this case you can abbreviate `this.new` as `new` and can leave off some `AbstractList.this` parts, since meaning is unambiguous.

Example: Using AbstractList

Problem: Want to create a *reversed view* of an existing `List` (same elements in reverse order). Operations on the original list affect the view, and vice-versa.

```
public class ReverseList<Item> extends AbstractList<Item> {
  private final List<Item> L;

  public ReverseList(List<Item> L) { this.L = L; }

  public int size() { return L.size(); }

  public Item get(int k) { return L.get(L.size()-k-1); }

  public void add(int k, Item x) { L.add(L.size()-k, x); }

  public Item set(int k, Item x) { return L.set(L.size()-k-1, x); }

  public Item remove(int k) { return L.remove(L.size() - k - 1); }
}
```

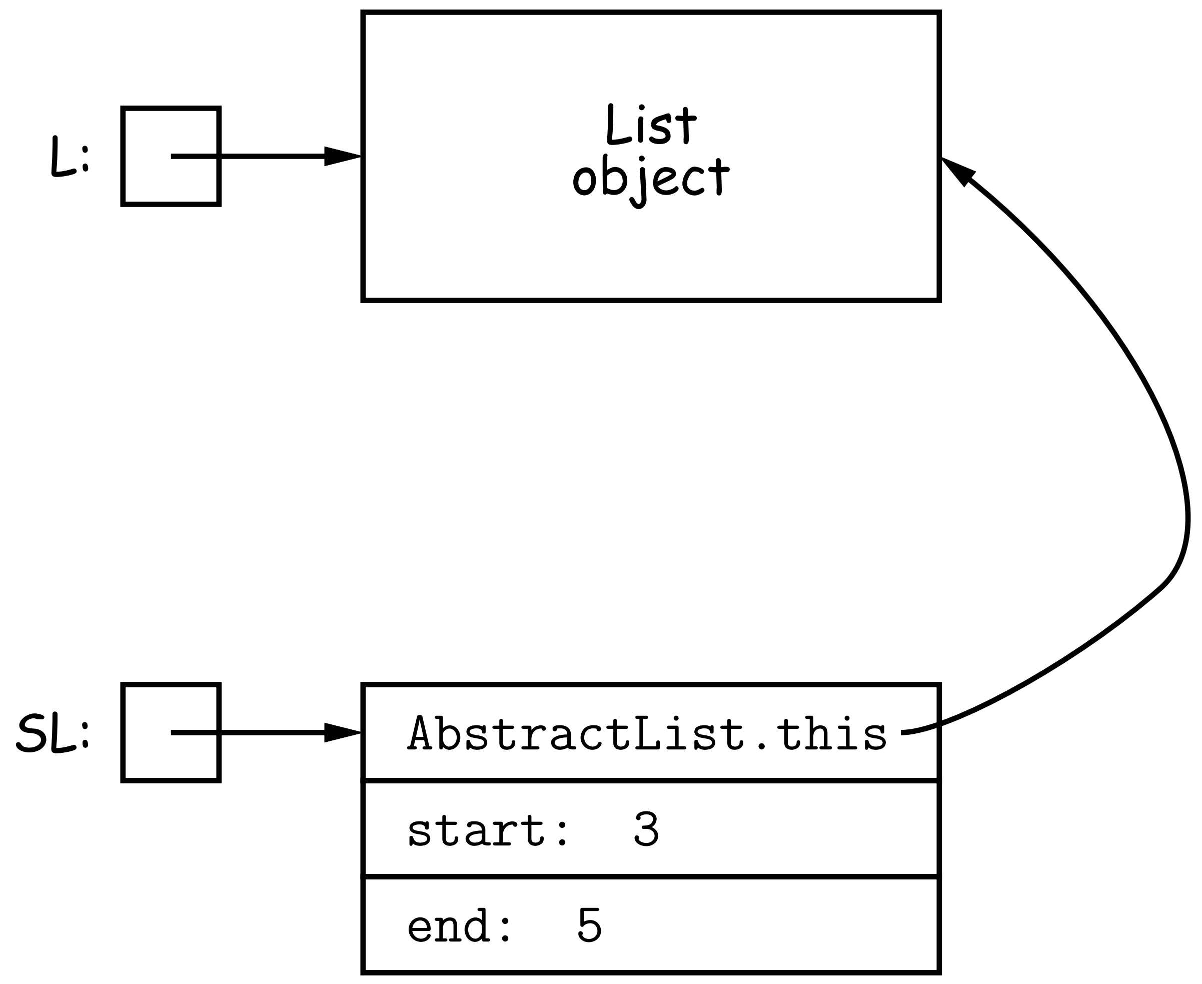
Getting a View: Sublists

Problem: `L.sublist(start, end)` is a `List` that gives a view of part of an existing list. Changes in one must affect the other. How?

```
// Continuation of class AbstractList.  Error checks not shown.
List<Item> sublist(int start, int end) {
  return this.new Sublist(start, end);
}

private class Sublist extends AbstractList<Item> {
  private int start, end;
  Sublist(int start, int end) { obvious }

  public int size() { return end-start; }
  public Item get(int k) { return AbstractList.this.get(start+k); }

  public void add(int k, Item x)
    { AbstractList.this.add(start+k, x); end += 1; }
  ...
}
```

What Does a Sublist Look Like?

- Consider `SL = L.sublist(3, 5);`



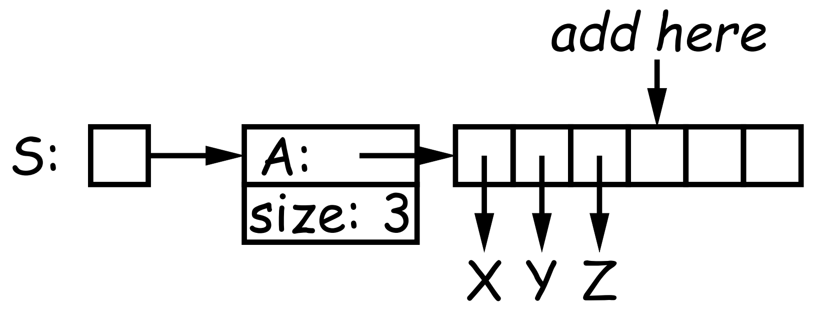
Arrays and Links

- Two main ways to represent a sequence: array and linked list
- In Java Library: `ArrayList` and `Vector` vs. `LinkedList`.
- Array:
  - Advantages: compact, fast ($\Theta(1)$) *random access* (indexing).
  - Disadvantages: insertion, deletion can be slow ($\Theta(N)$)
- Linked list:
  - Advantages: insertion, deletion fast once position found.
  - Disadvantages: space (link overhead), random access slow.

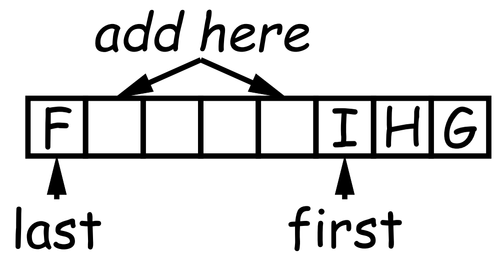
Implementing with Arrays

- Biggest problem using arrays is insertion/deletion in the *middle* of a list (must shove things over).
- Adding/deleting from ends can be made fast:
  - Double array size to grow; amortized cost constant (Lecture #15).
  - Growth at one end really easy; classical stack implementation:

```
S.push("X");
S.push("Y");
S.push("Z");
```



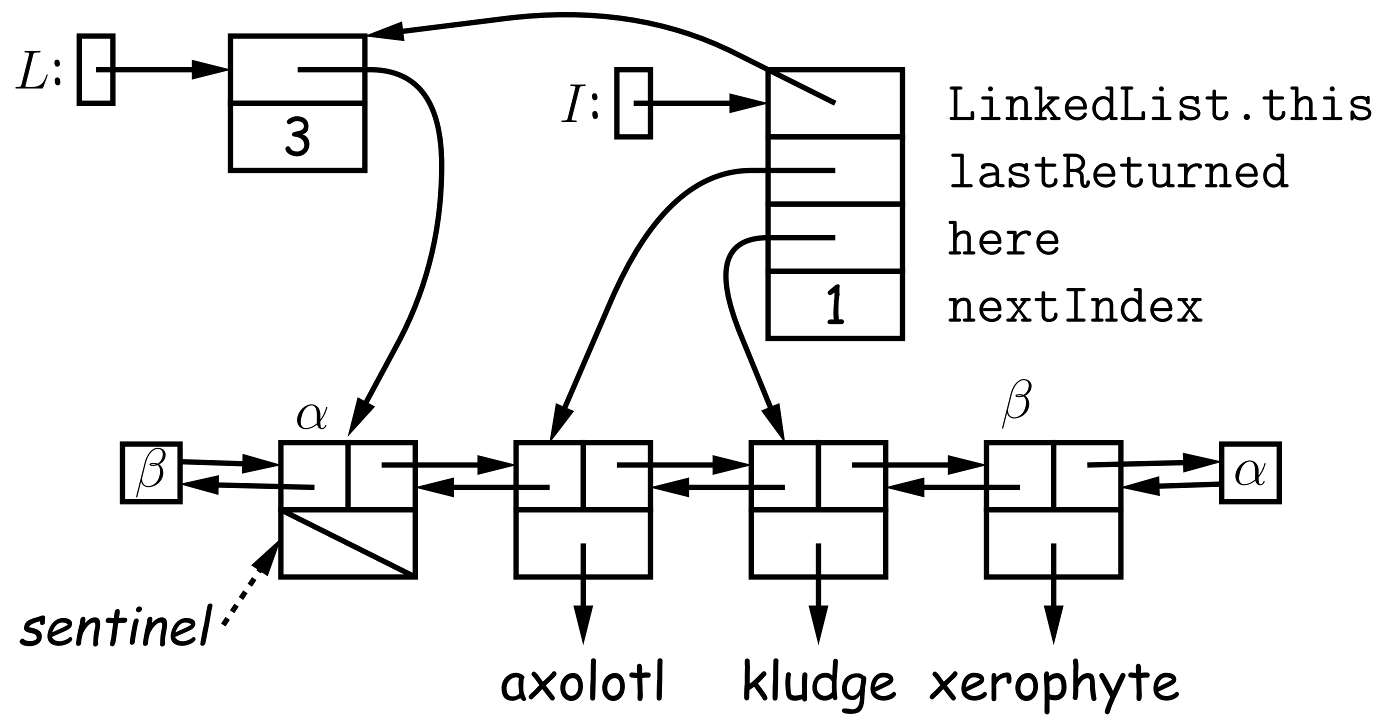
  - To allow growth at either end, use *circular buffering:*



  - Random access still fast.

Linking

- Essentials of linking should now be familiar
- Used in Java `LinkedList`. One possible representation for linked list and an iterator object over it:

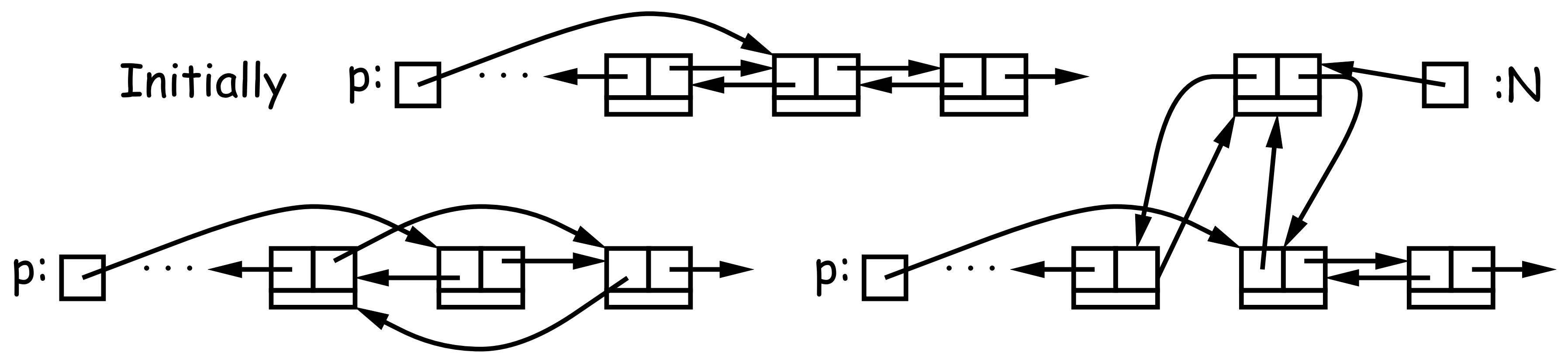


```
L = new LinkedList<String>();        I = L.listIterator();
L.add("axolotl");                    I.next();
L.add("kludge");
L.add("xerophyte");
```

Clever trick: Sentinels

- A *sentinel* is a dummy object containing no useful data except links.
- Used to eliminate special cases and to provide a fixed object to point to in order to access a data structure.
- Avoids special cases ('**if**' statements) by ensuring that the first and last item of a list always have (non-null) nodes—possibly sentinels—before and after them:

-
  ```
  // To delete list node at p:      // To add new node N before p:
  p.next.prev = p.prev;             N.prev = p.prev; N.next = p;
  p.prev.next = p.next;             p.prev.next = N;
                                    p.prev = N;
  ```



Specialization

- Traditional special cases of general list:
  - **Stack**: Add and delete from one end (LIFO).
  - **Queue**: Add at end, delete from front (FIFO).
  - **Dequeue**: Add or delete at either end.
- All of these easily representable by either array (with circular buffering for queue or deque) or linked list.
- Java has the `List` types, which can act like any of these (although with non-traditional names for some of the operations).
- Also has `java.util.Stack`, a subtype of `List`, which gives traditional names ("push", "pop") to its operations. There is, however, no "stack" interface.

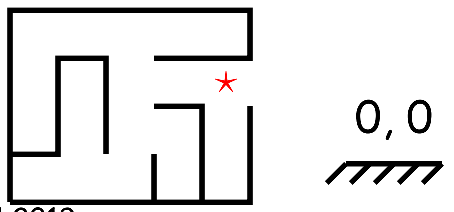
Stacks and Recursion

- Stacks related to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance benefit):
  - Calls become "push current variables and parameters, set parameters to new values, and loop."
  - Return becomes "pop to restore variables and parameters."

```
findExit(start):
  if isExit(start)
    FOUND
  else if (!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x) && !isCrumb(x)
          findExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legal(start,x) && !isCrumb(x)
            push x on S
```

Call: findExit((0,0))
Exit: (4, 2)



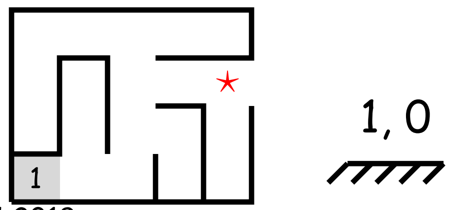
Stacks and Recursion

- Stacks related to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance benefit):
  - Calls become "push current variables and parameters, set parameters to new values, and loop."
  - Return becomes "pop to restore variables and parameters."

```
findExit(start):
  if isExit(start)
    FOUND
  else if (!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x) && !isCrumb(x)
          findExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legal(start,x) && !isCrumb(x)
            push x on S
```

Call: findExit((0,0))
Exit: (4, 2)



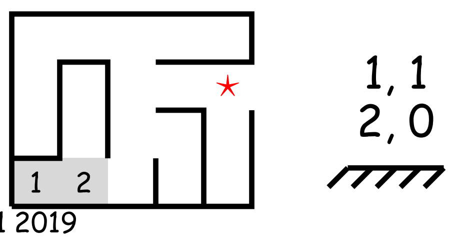
Stacks and Recursion

- Stacks related to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance benefit):
  - Calls become "push current variables and parameters, set parameters to new values, and loop."
  - Return becomes "pop to restore variables and parameters."

```
findExit(start):
  if isExit(start)
    FOUND
  else if (!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x) && !isCrumb(x)
          findExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legal(start,x) && !isCrumb(x)
            push x on S
```

Call: findExit((0,0))
Exit: (4, 2)



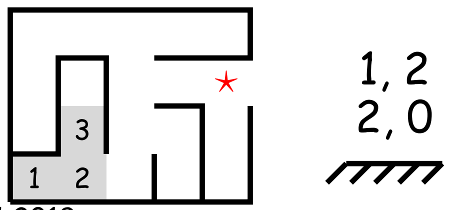
Stacks and Recursion

- Stacks related to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance benefit):
  - Calls become "push current variables and parameters, set parameters to new values, and loop."
  - Return becomes "pop to restore variables and parameters."

```
findExit(start):
  if isExit(start)
    FOUND
  else if (!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x) && !isCrumb(x)
          findExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legal(start,x) && !isCrumb(x)
            push x on S
```

Call: findExit((0,0))
Exit: (4, 2)



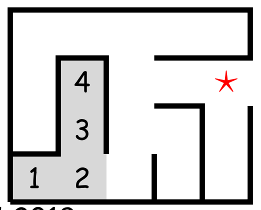
Stacks and Recursion

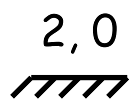
- Stacks related to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance benefit):
  - Calls become "push current variables and parameters, set parameters to new values, and loop."
  - Return becomes "pop to restore variables and parameters."

```
findExit(start):
  if isExit(start)
    FOUND
  else if (!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x) && !isCrumb(x)
          findExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legal(start,x) && !isCrumb(x)
            push x on S
```

Call: findExit((0,0))
Exit: (4, 2)





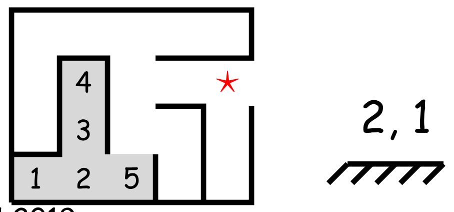
Stacks and Recursion

- Stacks related to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance benefit):
  - Calls become "push current variables and parameters, set parameters to new values, and loop."
  - Return becomes "pop to restore variables and parameters."

```
findExit(start):
  if isExit(start)
    FOUND
  else if (!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x) && !isCrumb(x)
          findExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legal(start,x) && !isCrumb(x)
            push x on S
```

Call: findExit((0,0))
Exit: (4, 2)



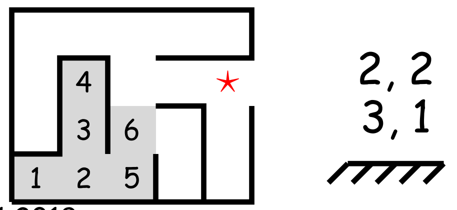
Stacks and Recursion

- Stacks related to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance benefit):
  - Calls become "push current variables and parameters, set parameters to new values, and loop."
  - Return becomes "pop to restore variables and parameters."

```
findExit(start):
  if isExit(start)
    FOUND
  else if (!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x) && !isCrumb(x)
          findExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legal(start,x) && !isCrumb(x)
            push x on S
```

Call: findExit((0,0))
Exit: (4, 2)



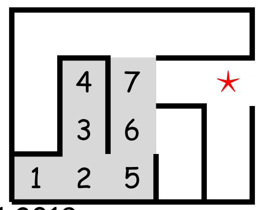
Stacks and Recursion

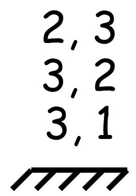
- Stacks related to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance benefit):
  - Calls become "push current variables and parameters, set parameters to new values, and loop."
  - Return becomes "pop to restore variables and parameters."

```
findExit(start):
  if isExit(start)
    FOUND
  else if (!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x) && !isCrumb(x)
          findExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legal(start,x) && !isCrumb(x)
            push x on S
```

Call: findExit((0,0))
Exit: (4, 2)





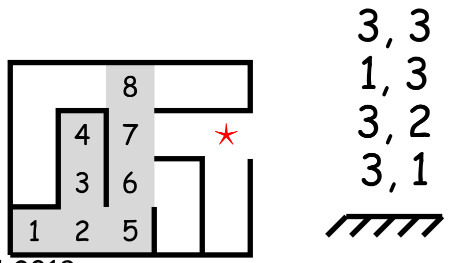
Stacks and Recursion

- Stacks related to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance benefit):
  - Calls become "push current variables and parameters, set parameters to new values, and loop."
  - Return becomes "pop to restore variables and parameters."

```
findExit(start):
  if isExit(start)
    FOUND
  else if (!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x) && !isCrumb(x)
          findExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legal(start,x) && !isCrumb(x)
            push x on S
```

Call: findExit((0,0))
Exit: (4, 2)



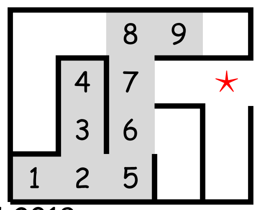
Stacks and Recursion

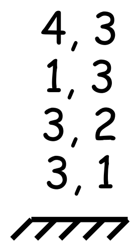
- Stacks related to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance benefit):
  - Calls become "push current variables and parameters, set parameters to new values, and loop."
  - Return becomes "pop to restore variables and parameters."

```
findExit(start):
  if isExit(start)
    FOUND
  else if (!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x) && !isCrumb(x)
          findExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legal(start,x) && !isCrumb(x)
            push x on S
```

Call: findExit((0,0))
Exit: (4, 2)





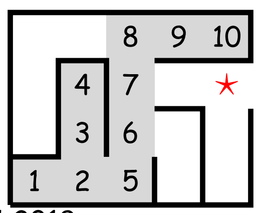
Stacks and Recursion

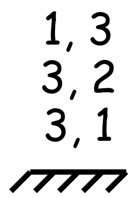
- Stacks related to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance benefit):
  - Calls become "push current variables and parameters, set parameters to new values, and loop."
  - Return becomes "pop to restore variables and parameters."

```
findExit(start):
  if isExit(start)
    FOUND
  else if (!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x) && !isCrumb(x)
          findExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legal(start,x) && !isCrumb(x)
            push x on S
```

Call: findExit((0,0))
Exit: (4, 2)





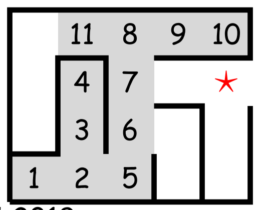
Stacks and Recursion

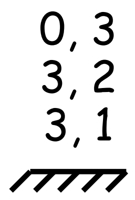
- Stacks related to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance benefit):
  - Calls become "push current variables and parameters, set parameters to new values, and loop."
  - Return becomes "pop to restore variables and parameters."

```
findExit(start):
  if isExit(start)
    FOUND
  else if (!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x) && !isCrumb(x)
          findExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legal(start,x) && !isCrumb(x)
            push x on S
```

Call: findExit((0,0))
Exit: (4, 2)





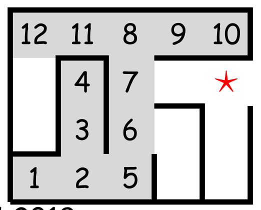
Stacks and Recursion

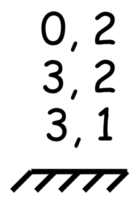
- Stacks related to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance benefit):
  - Calls become "push current variables and parameters, set parameters to new values, and loop."
  - Return becomes "pop to restore variables and parameters."

```
findExit(start):
  if isExit(start)
    FOUND
  else if (!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x) && !isCrumb(x)
          findExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legal(start,x) && !isCrumb(x)
            push x on S
```

Call: findExit((0,0))
Exit: (4, 2)





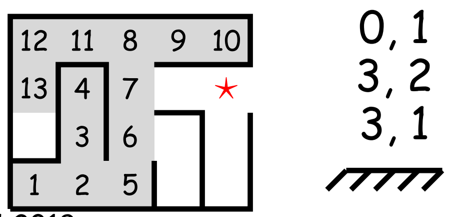
Stacks and Recursion

- Stacks related to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance benefit):
  - Calls become "push current variables and parameters, set parameters to new values, and loop."
  - Return becomes "pop to restore variables and parameters."

```
findExit(start):
  if isExit(start)
    FOUND
  else if (!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x) && !isCrumb(x)
          findExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legal(start,x) && !isCrumb(x)
            push x on S
```

Call: findExit((0,0))
Exit: (4, 2)



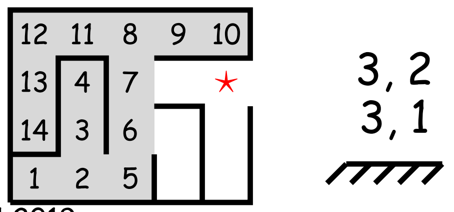
Stacks and Recursion

- Stacks related to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance benefit):
  - Calls become "push current variables and parameters, set parameters to new values, and loop."
  - Return becomes "pop to restore variables and parameters."

```
findExit(start):
  if isExit(start)
    FOUND
  else if (!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x) && !isCrumb(x)
          findExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legal(start,x) && !isCrumb(x)
            push x on S
```

Call: findExit((0,0))
Exit: (4, 2)



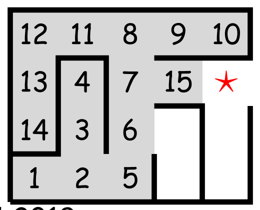
Stacks and Recursion

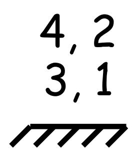
- Stacks related to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance benefit):
  - Calls become "push current variables and parameters, set parameters to new values, and loop."
  - Return becomes "pop to restore variables and parameters."

```
findExit(start):
  if isExit(start)
    FOUND
  else if (!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x) && !isCrumb(x)
          findExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legal(start,x) && !isCrumb(x)
            push x on S
```

Call: findExit((0,0))
Exit: (4, 2)





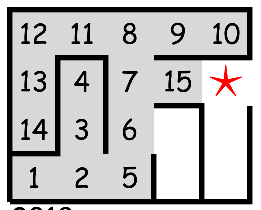
Stacks and Recursion

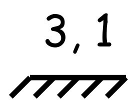
- Stacks related to recursion. In fact, can convert any recursive algorithm to stack-based (however, generally no great performance benefit):
  - Calls become "push current variables and parameters, set parameters to new values, and loop."
  - Return becomes "pop to restore variables and parameters."

```
findExit(start):
  if isExit(start)
    FOUND
  else if (!isCrumb(start))
    leave crumb at start;
    for each square, x,
      adjacent to start:
        if legal(start,x) && !isCrumb(x)
          findExit(x)
```

```
findExit(start):
  S = new empty stack;
  push start on S;
  while S not empty:
    pop S into start;
    if isExit(start)
      FOUND
    else if (!isCrumb(start))
      leave crumb at start;
      for each square, x,
        adjacent to start (in reverse):
          if legal(start,x) && !isCrumb(x)
            push x on S
```

Call: findExit((0,0))
Exit: (4, 2)





Design Choices: Extension, Delegation, Adaptation

- The standard `java.util.Stack` type *extends* `Vector`:

```
class Stack<Item> extends Vector<Item> { void push(Item x) { add(x); } ... }
```

- Could instead have *delegated* to a field:

```
class ArrayStack<Item> {
   private ArrayList<Item> repl = new ArrayList<Item>();
   void push(Item x) { repl.add(x); } ...
}
```

- Or, could generalize, and define an *adapter:* a class used to make objects of one kind behave as another:

```
public class StackAdapter<Item> {
   private List repl;
   /** A stack that uses REPL for its storage. */
   public StackAdapter(List<Item> repl) { this.repl = repl; }
   public void push(Item x) { repl.add(x); } ...
}

class ArrayStack<Item> extends StackAdapter<Item> {
  ArrayStack() { super(new ArrayList<Item>()); }
}
```

CS61B Lecture #20: Trees

A Recursive Structure

- Trees naturally represent recursively defined, hierarchical objects with more than one recursive subpart for each instance.
- Common examples: expressions, sentences.
  - Expressions have definitions such as "an expression consists of a literal or two expressions separated by an operator."
- Also describe structures in which we recursively divide a set into multiple subsets.

Formal Definitions

- Trees come in a variety of flavors, all defined recursively:
  - **61A style:** A tree consists of a *label* value and zero or more *branches* (or *children*), each of them a tree.
  - **61A style, alternative definition:** A tree is a set of *nodes* (or *vertices*), each of which has a label value and one or more *child nodes*, such that no node descends (directly or indirectly) from itself. A node is the *parent* of its children.
  - **Positional trees:** A tree is either *empty* or consists of a node containing a label value and an indexed sequence of zero or more children, each a positional tree. If every node has two positions, we have a *binary tree* and the children are its *left and right sub-trees.* Again, nodes are the parents of their non-empty children.
  - We'll see other varieties when considering graphs.

Tree Characteristics (I)

- The *root* of a tree is a non-empty node with no parent in that tree (its parent might be in some larger tree that contains that tree as a subtree). Thus, every node is the root of a (sub)tree.
- The *order*, *arity*, or *degree* of a node (tree) is its number (maximum number) of children.
- The nodes of a *k-ary tree* each have at most k children.
- A *leaf* node has no children (no non-empty children in the case of positional trees).

Tree Characteristics (II)

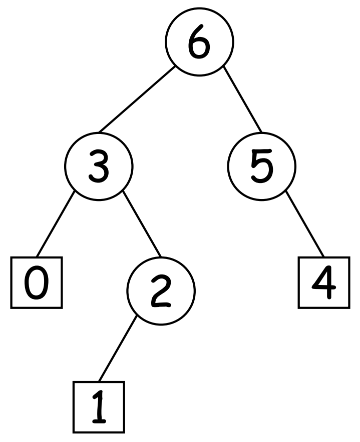
- The *height* of a node in a tree is the largest distance to a leaf. That is, a leaf has height 0 and a non-empty tree's height is one more than the maximum height of its children. The height of a tree is the height of its root.
- The *depth* of a node in a tree is the distance to the root of that tree. That is, in a tree whose root is R, R itself has depth 0 in R, and if node $S \neq R$ is in the tree with root R, then its depth is one greater than its parent's.

A Tree Type, 61A Style

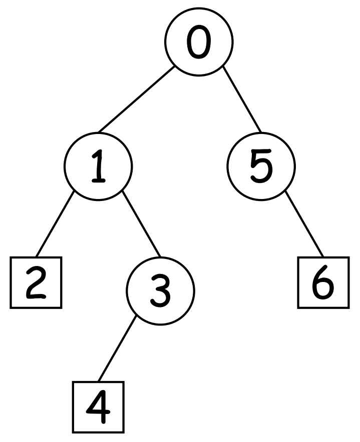
```
public class Tree<Label> {

    // This constructor is convenient, but unfortunately requires this
    // SuppressWarnings annotation to prevent (harmless) warnings
    // that we will explain later.
    @SuppressWarnings("unchecked")
    public Tree(Label label, Tree<Label>... children) {
        _label = label;
        _kids = new ArrayList<>(Arrays.asList(children));
    }

    public int arity() { return _kids.size(); }

    public Label label() { return _label; }

    public Tree<Label> child(int k) { return _kids.get(k); }

    private Label _label;
    private ArrayList<Tree<Label>> _kids;
}
```

Fundamental Operation: Traversal

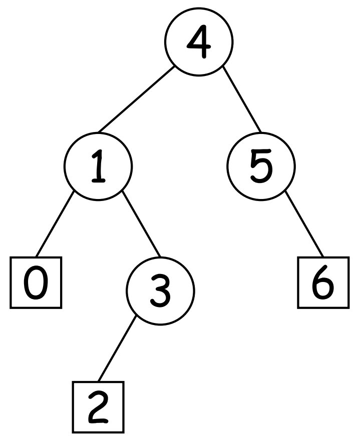
- *Traversing a tree* means enumerating (some subset of) its nodes.
- Typically done recursively, because that is natural description.
- As nodes are enumerated, we say they are *visited*.
- Three basic orders for enumeration (+ variations):
  - **Preorder:** visit node, traverse its children.
  - **Postorder:** traverse children, visit node.
  - **Inorder:** traverse first child, visit node, traverse second child (binary trees only).



Postorder

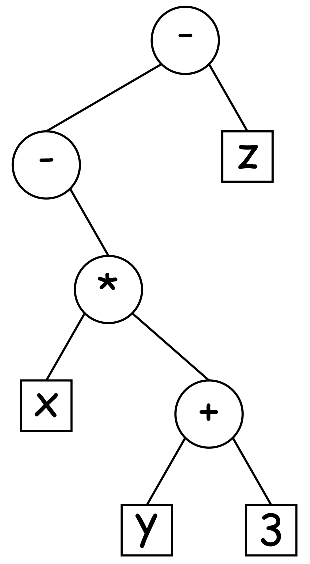


Preorder



inorder

Preorder Traversal and Prefix Expressions

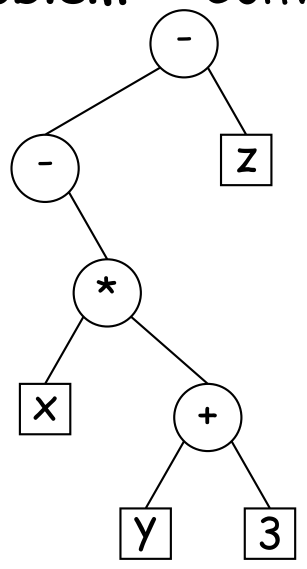
Problem: Convert  **into** `(- (- (* x (+ y 3))) z)`

(Assume `Tree<`*Label*`>` is means "Tree whose labels have type *Label*.")

```
static String toLisp(Tree<String> T) {
  if (T.arity() == 0) return T.label();
  else {
    String R;  R = "(" + T.label();
    for (int i = 0; i < T.arity(); i += 1)
      R += " " + toLisp(T.child(i));
    return R + ")";
  }
}
```

Inorder Traversal and Infix Expressions

Problem: Convert



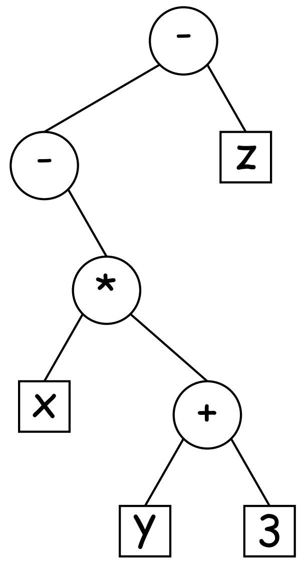
into ((-(x*(y+3)))-z)

To think about: how to get rid of all those parentheses.

```
static String toInfix(Tree<String> T) {
  if (T.arity() == 0) {
    return T.label();
  } else if (T.arity() == 1) {
    return "(" T.label() + toInfix(T.child(0)) + ")";
  } else {
    return "(" toInfix(T.child(0)) + T.label() + toInfix(T.child(1)) + ")";
  }
}
```

Postorder Traversal and Postfix Expressions

Problem: Convert



into `x y 3 +:2 *:2 -:1 z -:2`

```java
static String toPolish(Tree<String> T) {
  String R;   R = "";
  for (int i = 0; i < T.arity(); i += 1)
     R += toPolish(T.child(i)) + " ";
  return R + String.format("%s:%d", T.label(), T.arity());
}
```

A General Traversal: The Visitor Pattern

```
void preorderTraverse(Tree<Label> T, Consumer<Tree<Label>> visit)
{
  if (T != null) {
    visit.accept(T);
    for (int i = 0; i < T.arity(); i += 1)
      preorderTraverse(T.child(i), visit);
  }
}
```

- `java.util.function.Consumer<AType>` is a library interface that works as a function-like type with one void method, `accept`, which takes an argument of type `AType`.
- Now, using Java 8 lambda syntax, I can print all labels in the tree in preorder with:

```
preorderTraverse(myTree, T -> System.out.print(T.label() + " "));
```

Iterative Depth-First Traversals

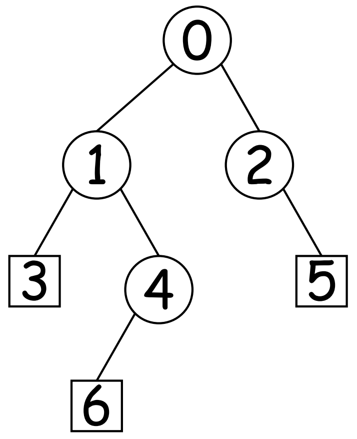
- Tree recursion conceals data: a *stack* of nodes (all the `T` arguments) and a little extra information. Can make the data explicit:

```
void preorderTraverse2(Tree<Label> T, Consumer<Tree<Label>> visit) {
  Stack<Tree<Label>> work = new Stack<>();
  work.push(T);
  while (!work.isEmpty()) {
    Tree<Label> node = work.pop();
    visit.accept(node);
    for (int i = node.arity()-1; i >= 0; i -= 1)
        work.push(node.child(i));  // Why backward?
  }
}
```

- This traversal takes the same $\Theta(\cdot)$ time as doing it recursively, and also the same $\Theta(\cdot)$ space.
- That is, we have substituted an explicit stack data structure (`work`) for Java's built-in execution stack (which handles function calls).

Level-Order (Breadth-First) Traversal

Problem: Traverse all nodes at depth 0, then depth 1, etc:



Breadth-First Traversal Implemented

A simple modification to iterative depth-first traversal gives breadth-first traversal. Just change the (LIFO) stack to a (FIFO) queue:

```
void breadthFirstTraverse(Tree<Label> T, Consumer<Tree<Label>> visit) {
  ArrayDeque<Tree<Label>> work = new ArrayDeque<>();  // (Changed)
  work.push(T);
  while (!work.isEmpty()) {
    Tree<Label> node = work.remove();    // (Changed)
    if (node != null) {
        visit.accept(node);
        for (int i = 0; i < node.arity(); i += 1) // (Changed)
           work.push(node.child(i));
    }
  }
}
```

Times

- The traversal algorithms have roughly the form of the `boom` example in §1.3.3 of *Data Structures*—an exponential algorithm.
- However, the role of M in that algorithm is played by the *height* of the tree, not the number of nodes.
- In fact, easy to see that tree traversal is *linear:* $\Theta(N)$, where N is the # of nodes: Form of the algorithm implies that there is one visit at the root, and then one visit for every *edge* in the tree. Since every node but the root has exactly one parent, and the root has none, must be $N-1$ edges in any non-empty tree.
- In positional tree, is also one recursive call for each empty tree, but # of empty trees can be no greater than kN, where k is arity.
- For k-ary tree (max # children is k), $h+1 \le N \le \frac{k^{h+1}-1}{k-1}$, where h is height.
- So $h \in \Omega(\log_k N) = \Omega(\lg N)$ and $h \in O(N)$.
- Many tree algorithms look at one child only. For them, worst-case time is proportional to the *height* of the tree—$\Theta(\lg N)$—assuming that tree is *bushy*—each level has about as many nodes as possible.

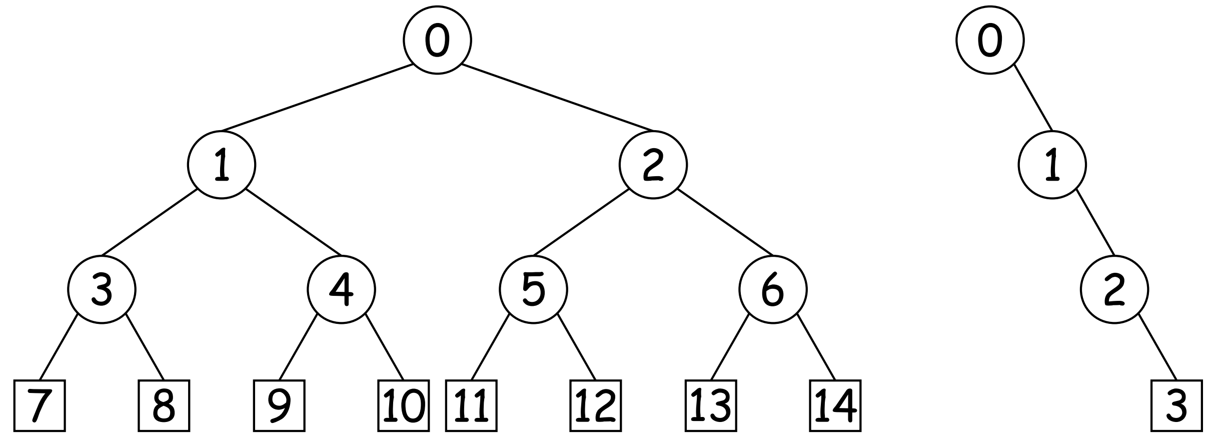
Recursive Breadth-First Traversal: Iterative Deepening

- Previous breadth-first traversal used space proportional to the *width* of the tree, which is $\Theta(N)$ for bushy trees, whereas depth-first traversal takes $\lg N$ space on bushy trees.
- Can we get breadth-first traversal in $\lg N$ space and $\Theta(N)$ time on bushy trees?
- For each level, k, of the tree from 0 to lev, call `doLevel(T,k)`:

```
void doLevel(Tree T, int lev) {
  if (lev == 0)
    visit T
  else
    for each non-null child, C, of T {
      doLevel(C, lev-1);
    }
}
```

- So we do breadth-first traversal by repeated (truncated) depth-first traversals: *iterative deepening*.
- In `doLevel(T, k)`, we skip (i.e., traverse but don't visit) the nodes before level k, and then visit at level k, but not their children.

Iterative Deepening Time?



- Let h be height, N be # of nodes.
- Count # edges traversed (i.e, # of calls, not counting null nodes).
- First (full) tree: 1 for level 0, 3 for level 1, 7 for level 2, 15 for level 3.
- Or in general $(2^1 - 1) + (2^2 - 1) + \ldots + (2^{h+1} - 1) = 2^{h+2} - h \in \Theta(N)$, since $N = 2^{h+1} - 1$ for this tree.
- Second (*right leaning*) tree: 1 for level 0, 2 for level 2, 3 for level 3.
- Or in general $(h+1)(h+2)/2 = N(N+1)/2 \in \Theta(N^2)$, since $N = h+1$ for this kind of tree.

Iterators for Trees

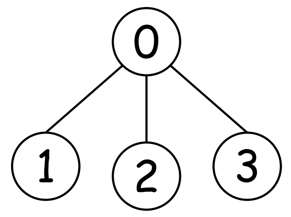
- Frankly, iterators are not terribly convenient on trees.
- But can use ideas from iterative methods.

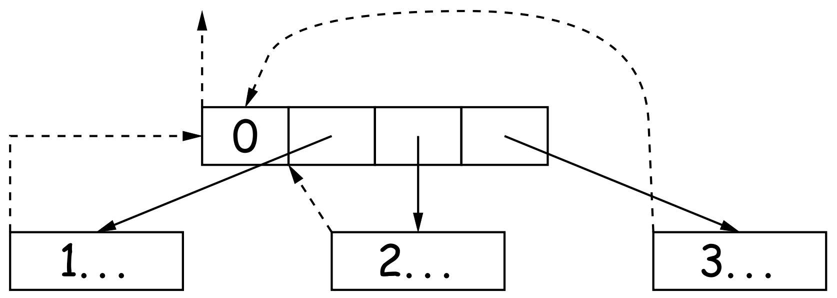
```
class PreorderTreeIterator<Label> implements Iterator<Label> {
  private Stack<Tree<Label>> s = new Stack<Tree<Label>>();

  public PreorderTreeIterator(Tree<Label> T) { s.push(T);  }

  public boolean hasNext() { return !s.isEmpty(); }
  public T next() {
    Tree<Label> result = s.pop();
    for (int i = result.arity()-1; i >= 0; i -= 1)
      s.push(result.child(i));
    return result.label();
  }
}
```

Example: (what do I have to add to class `Tree` first?)

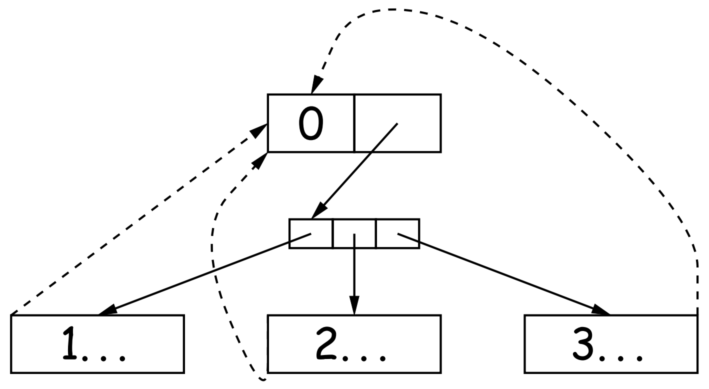
```
for (String label : aTree) System.out.print(label + " ");
```

Tree Representation

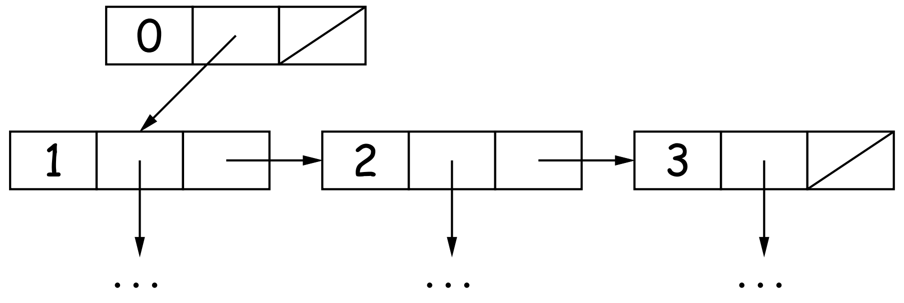




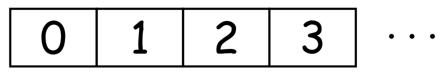
(a) Embedded child pointers
(+ optional parent pointers)



(b) Array of child pointers
(+ optional parent pointers)



(c) child/sibling pointers

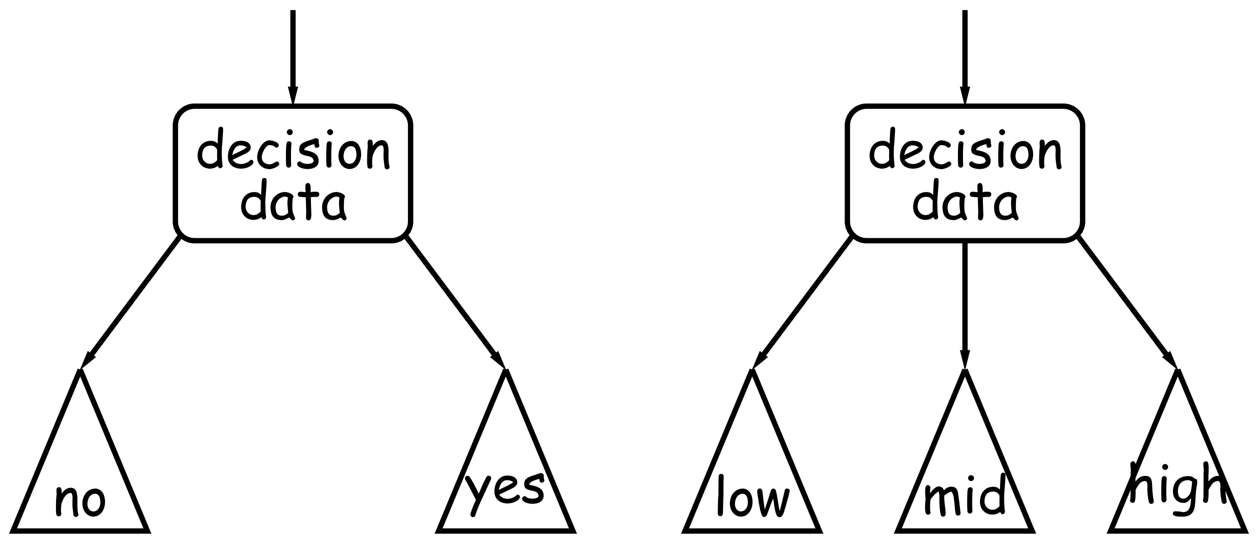


(d) breadth-first array
(complete trees)

CS61B Lecture #21: Tree Searching

Divide and Conquer

- Much (most?) computation is devoted to finding things in response to various forms of query.
- Linear search for response can be expensive, especially when data set is too large for primary memory.
- Preferable to have criteria for *dividing* data to be searched into pieces recursively
- We saw the figure for $\lg N$ algorithms: at 1 μsec per comparison, could process 10^{300000} items in 1 sec.
- Tree is a natural framework for the representation:



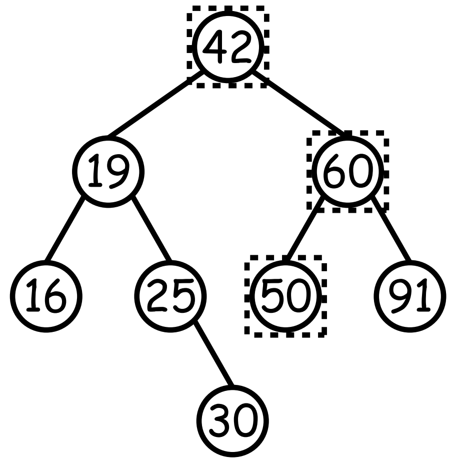
Binary Search Trees

Binary Search Property:

- Tree nodes contain *keys,* and possibly other data.
- All nodes in left subtree of node have *smaller* keys.
- All nodes in right subtree of node have *larger* keys.
- "Smaller" means any complete transitive, anti-symmetric ordering on keys:
  - exactly one of $x \prec y$ and $y \prec x$ true.
  - $x \prec y$ and $y \prec z$ imply $x \prec z$.
  - (To simplify, won't allow duplicate keys this semester).
- E.g., in dictionary database, node label would be (*word, definition*): *word* is the key.
- For concreteness here, we'll just use the standard Java convention of calling `.compareTo`.

Finding

- Searching for 50 and 49:

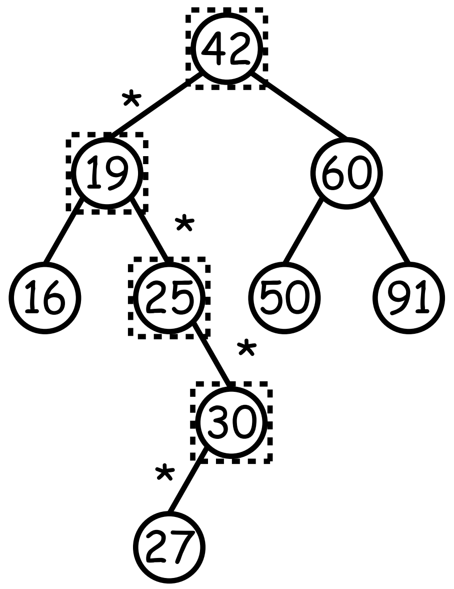


```
/** Node in T containing L, or null if none */
static BST find(BST T, Key L) {
  if (T == null)
    return T;
  if (L.compareTo(T.label()) == 0)
    return T;
  else if (L.compareTo(T.label()) < 0)
    return find(T.left(), L);
  else
    return find(T.right(), L);
}
```

- Dashed boxes show which node labels we look at.
- Number looked at proportional to height of tree.

Inserting

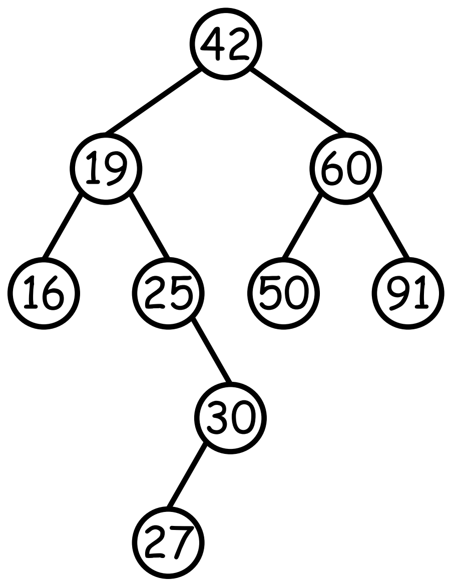
- Inserting 27



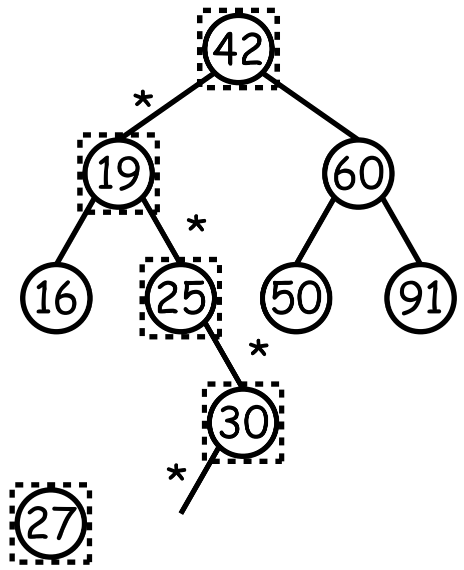
```
/** Insert L in T, replacing existing
 *  value if present, and returning
 *  new tree. */
static BST insert(BST T, Key L) {
  if (T == null)
    return new BST(L);
  if (L.compareTo(T.label()) == 0)
    T.setLabel(L);
  else if (L.compareTo(T.label()) < 0)
    T.setLeft(insert(T.left(), L));
  else
    T.setRight(insert(T.right(), L));
  return T;
}
```

- Starred edges are set (to themselves, unless initially null).
- Again, time proportional to height.

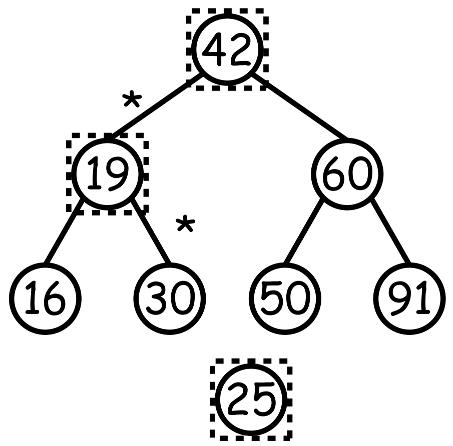
Deletion



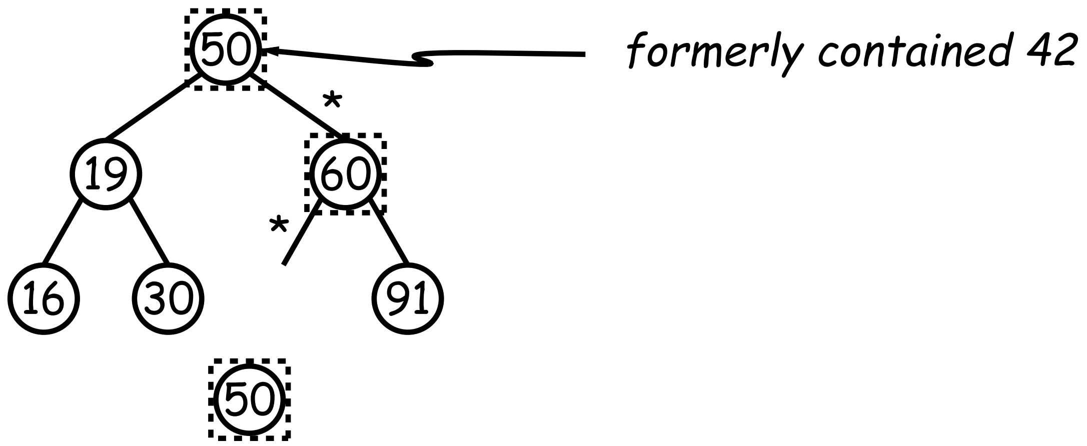
Initial



Remove 27

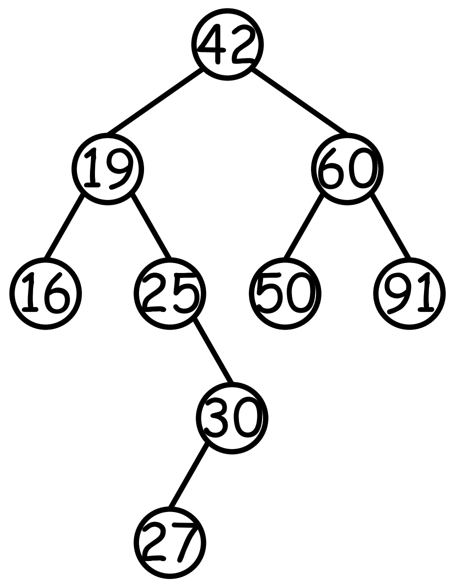


Remove 25



Remove 42

Deletion Algorithm

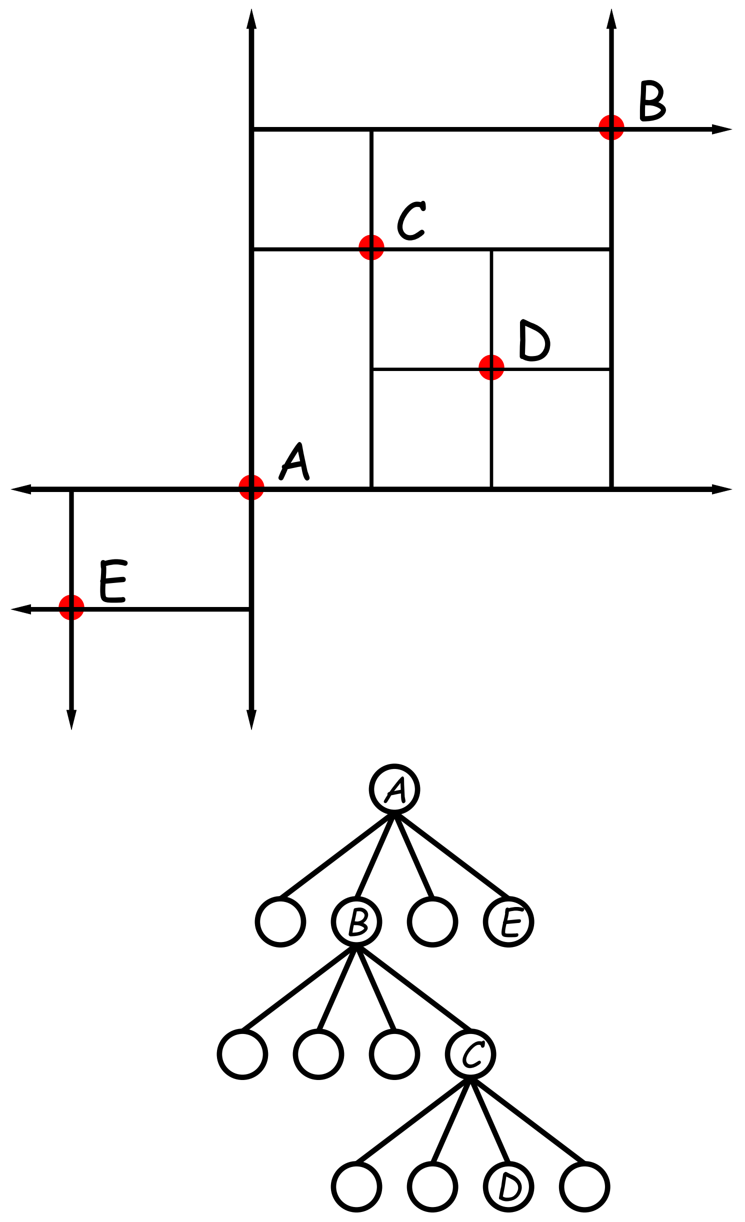


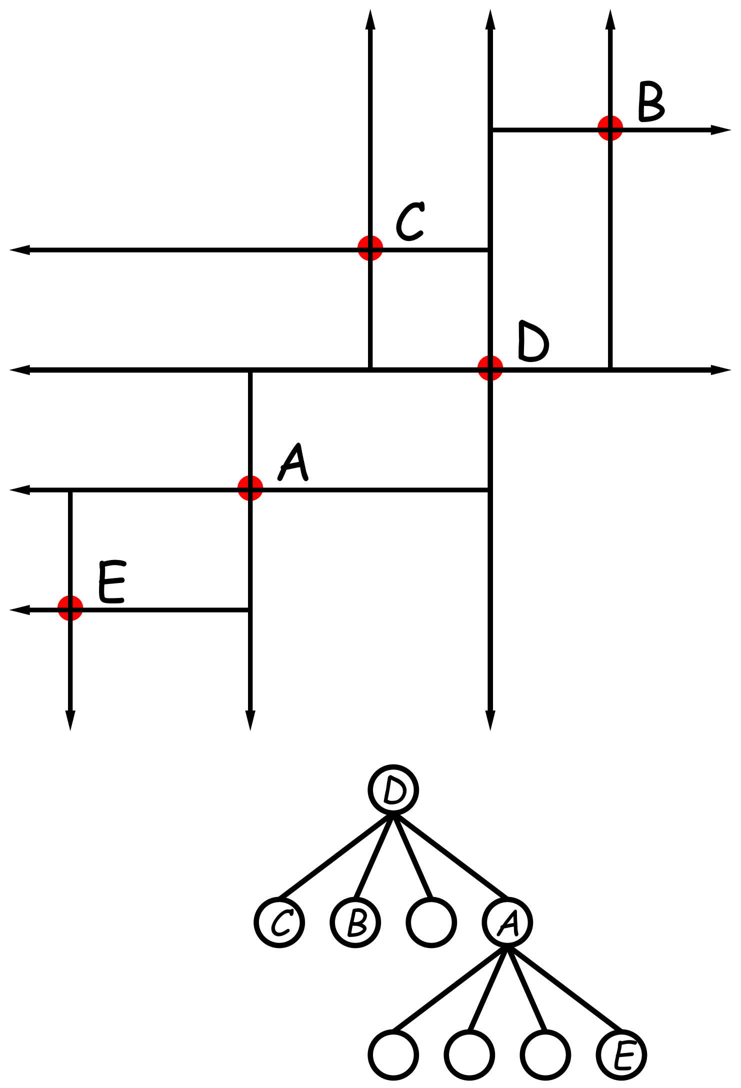
```
/** Remove L from T, returning new tree. */
static BST remove(BST T, Key L) {
  if (T == null)
    return null;
  if (L.compareTo(T.label()) == 0) {
     if (T.left() == null)
         return T.right();
     else if (T.right() == null)
         return T.left();
     else {
         Key smallest = minVal(T.right());  // ??
         T.setRight(remove(T.right(), smallest));
         T.setLabel(smallest);
     }
  }
  else if (L.compareTo(T.label()) < 0)
    T.setLeft(remove(T.left(), L));
  else
    T.setRight(remove(T.right(), L));
  return T;
}
```

More Than Two Choices: Quadtrees

- Want to *index* information about 2D locations so that items can be retrieved by position.
- Quadtrees do so using standard data-structuring trick: *Divide and Conquer*.
- Idea: divide (2D) space into four *quadrants,* and store items in the appropriate quadrant. Repeat this recursively with each quadrant that contains more than one item.
- Original definition: a quadtree is either
  - Empty, or
  - An item at some position (x, y), called the root, plus
  - four quadtrees, each containing only items that are northwest, northeast, southwest, and southeast of (x, y).
- Big idea is that if you are looking for point (x', y') and the root is not the point you are looking for, you can narrow down which of the four subtrees of the root to look in by comparing coordinates (x, y) with (x', y').

Classical Quadtree: Example

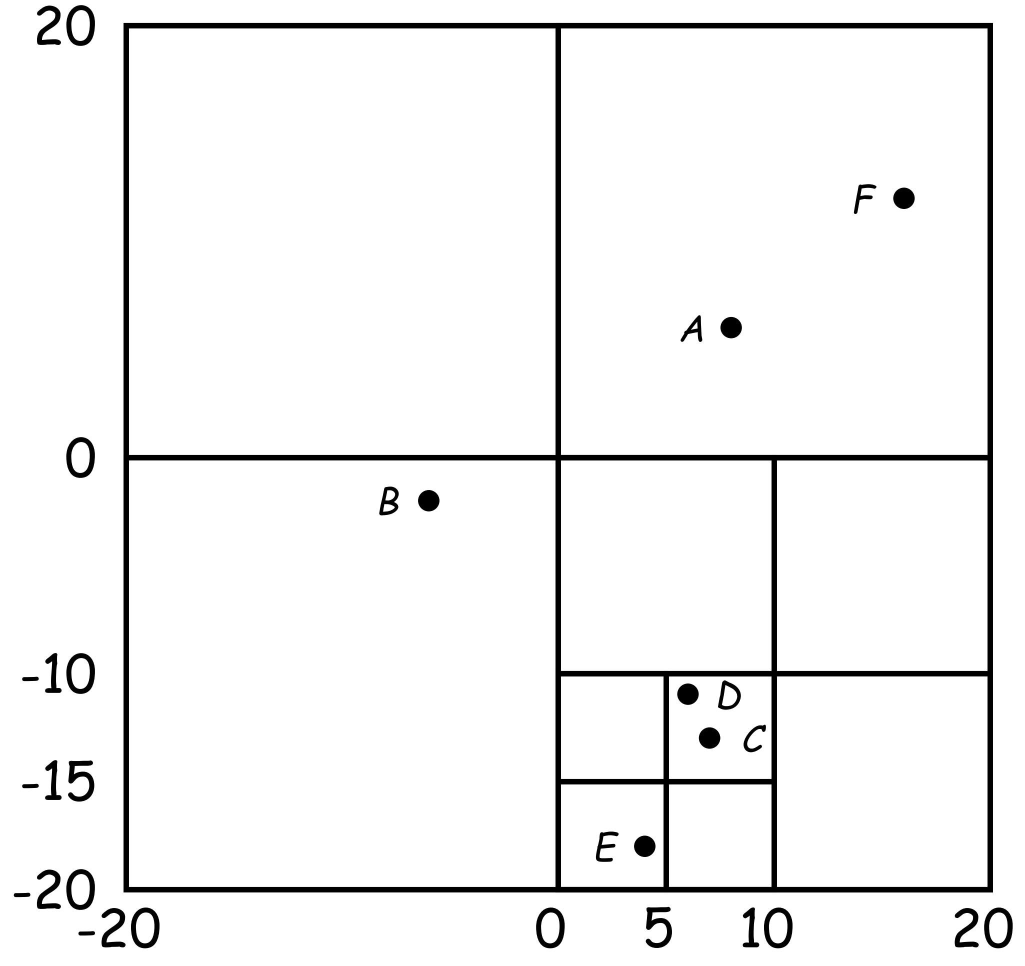


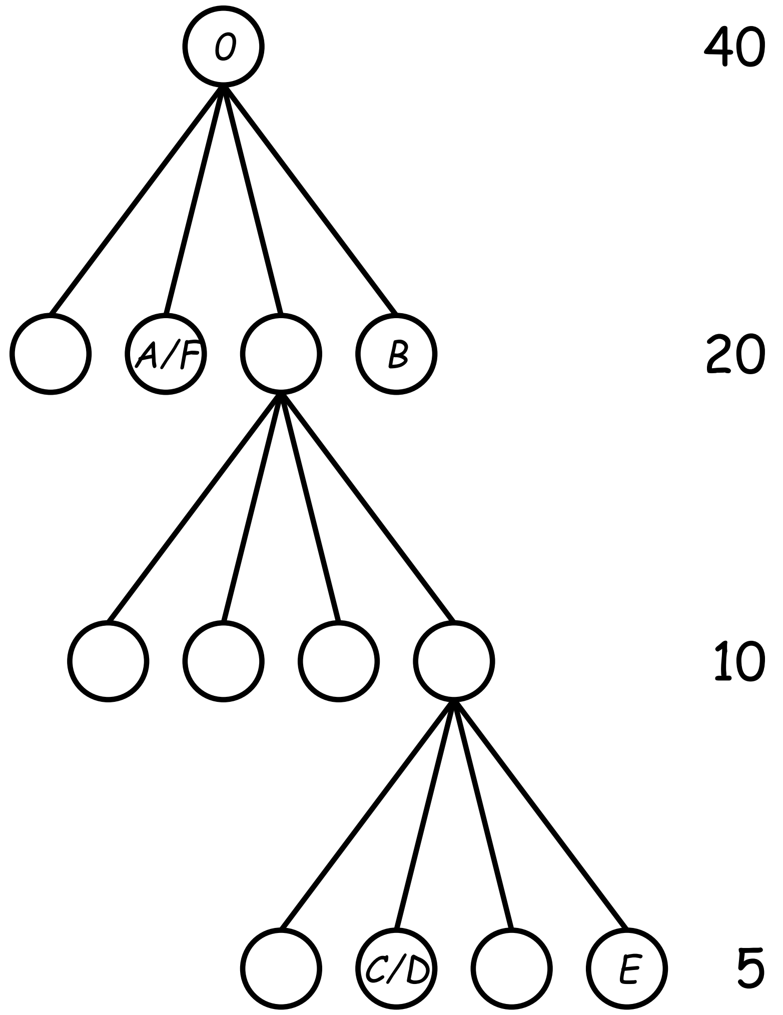


Point-region (PR) Quadtrees

- If we use a Quadtree to track moving objects, it may be useful to be able to *delete* items from a tree: when an object moves, the subtree that it goes in may change.
- Difficult to do with the classical data structure above, so we'll define instead:
- A quadtree consists of a bounding rectangle, B and either
  - Zero up to a small number of items that lie in that rectangle, or
  - Four quadtrees whose bounding rectangles are the four quadrants of B (all of equal size).
- A completely empty quadtree can have an arbitrary bounding rectangle, or you can wait for the first point to be inserted.

Example of PR Quadtree





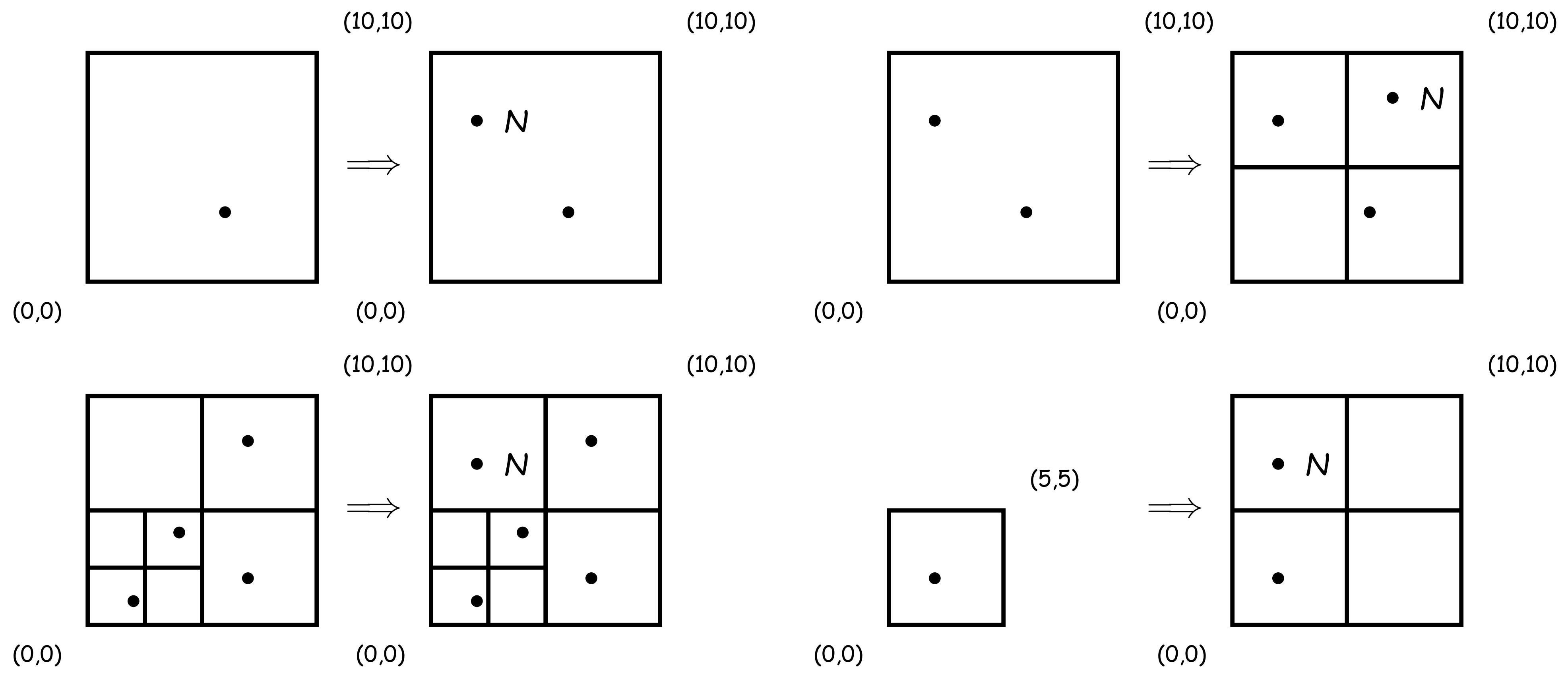
(≤ 2 points per leaf)

Navigating PR Quadtrees

- To find an item at (x, y) in quadtree T,
  1. If (x, y) is outside the bounding rectangle of T, or T is empty, then (x, y) is not in T.
  2. Otherwise, if T contains a small set of items, then (x, y) is in T iff it is among these items.
  3. Otherwise, T consists of four quadtrees. Recursively look for (x, y) in each (however, step #1 above will cause all but one of these bounding boxes to reject the point immediately).
- Similar procedure works when looking for all items within some rectangle, R:
  1. If R does not intersect the bounding rectangle of T, or T is empty, then there are no items in R.
  2. Otherwise, if T contains a set of items, return those that are in R, if any.
  3. Otherwise, T consists of four quadtrees. Recursively look for points in R in each one of them.

Insertion into PR Quadtrees

Various cases for inserting a new point N, assuming maximum occupancy of a region is 2, showing initial state $\Longrightarrow$ final state.



CS61B Lecture #22

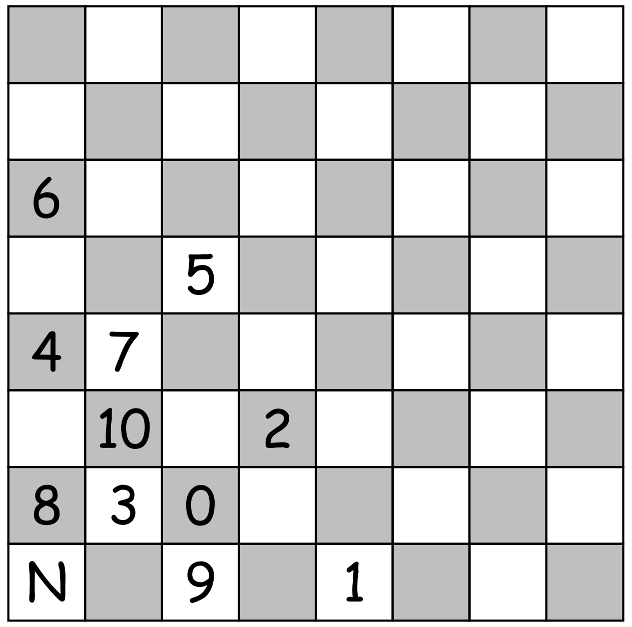
Today: Backtracking searches, game trees (*DSIJ*, Section 6.5)

Searching by “Generate and Test”

- We’ve been considering the problem of searching a set of data stored in some kind of data structure: “Is $x \in S$?”
- But suppose we *don’t* have a set S, but know how to recognize what we’re after if we find it: “Is there an x such that $P(x)$?”
- If we know how to enumerate all possible candidates, can use approach of *Generate and Test:* test all possibilities in turn.
- Can sometimes be more clever: avoid trying things that won’t work, for example.
- What happens if the set of possible candidates is infinite?

Backtracking Search

- Backtracking search is one way to enumerate all possibilities.
- Example: *Knight's Tour.* Find all paths a knight can travel on a chessboard such that it touches every square exactly once and ends up one knight move from where it started.
- In the example below, the numbers indicate position numbers (knight starts at 0).
- Here, knight (N) is stuck; how to handle this?



General Recursive Algorithm

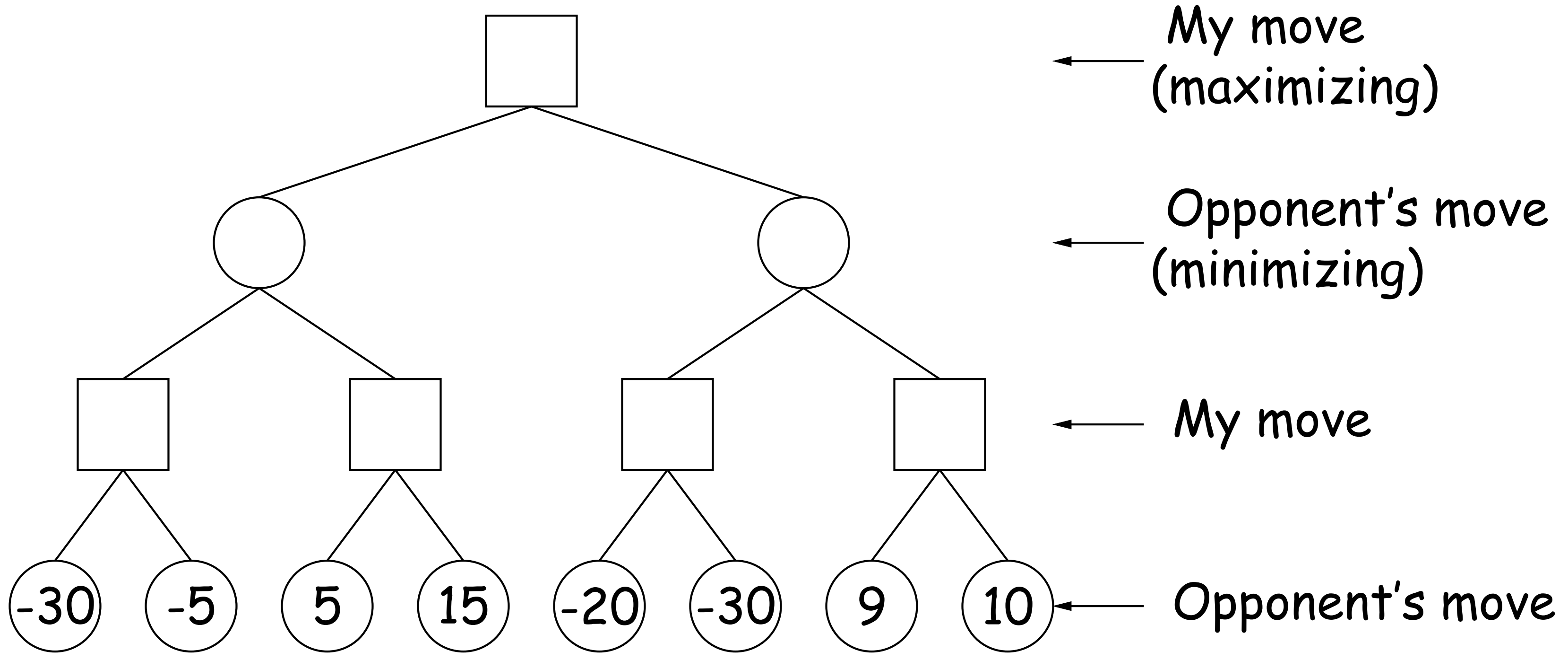
```
/** Append to PATH a sequence of knight moves starting at ROW, COL
 *  that avoids all squares that have been hit already and
 *  that ends up one square away from ENDROW, ENDCOL. B[i][j] is
 *  true iff row i and column j have been hit on PATH so far.
 *  Returns true if it succeeds, else false (with no change to PATH).
 *  Call initially with PATH containing the starting square, and
 *  the starting square (only) marked in B. */

boolean findPath(boolean[][] b, int row, int col,
                 int endRow, int endCol, List path) {
  if (path.size() == 64)   return isKnightMove(row, col, endRow, endCol);
  for (r, c = all possible moves from (row, col)) {
    if (!b[r][c]) {
      b[r][c] = true; // Mark the square
      path.add(new Move(r, c));
      if (findPath(b, r, c, endRow, endCol, path))  return true;
      b[r][c] = false; // Backtrack out of the move.
      path.remove(path.size()-1);
    }
  }
  return false;
}
```

Another Kind of Search: Best Move

- Consider the problem of finding the *best* move in a two-person game.
- One way: assign a *heuristic value* to each possible move and pick highest (aka *static evaluation*). Examples:
  - number of black pieces − number of white pieces in checkers.
  - weighted sum of white piece values − weighted sum of black pieces in chess (Queen=9, Rook=5, etc.)
  - Nearness of pieces to strategic areas (center of board).
- But this is misleading. A move might give us more pieces, but set up a devastating response from the opponent.
- So, for each move, look at *opponent's* possible moves, assume he picks the best one for him, and use that as the value.
- But what if *you* have a great response to his response?
- How do we organize this sensibly?

Game Trees

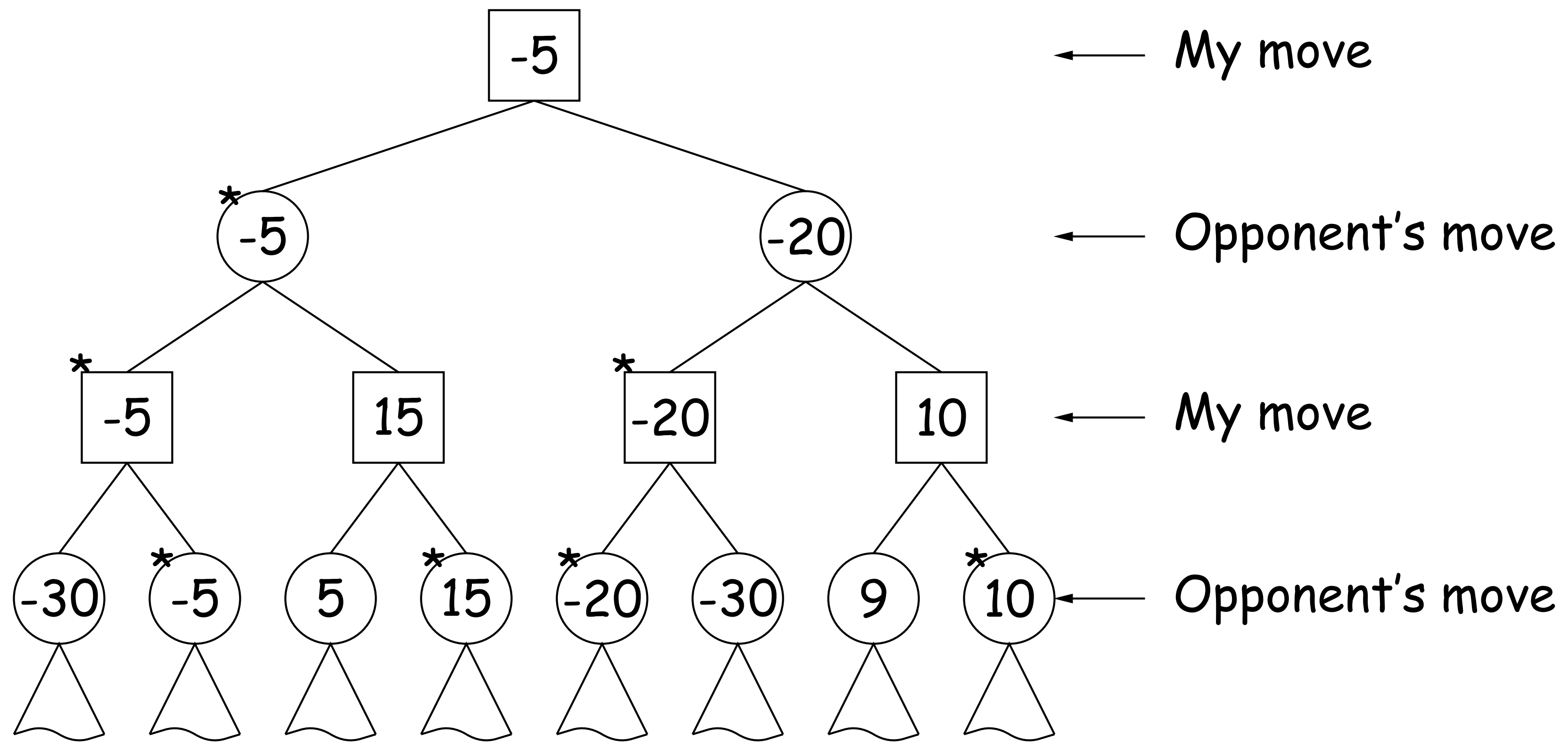
- Think of the space of possible continuations of the game as a tree.
- Each node is a position, each edge a move.



- Suppose numbers at the bottom are the values of those final positions *to me*. Smaller numbers are of more value to *my opponent*.
- What should I move? What value can I get if my opponent plays as well as possible?

Game Trees, Minimax

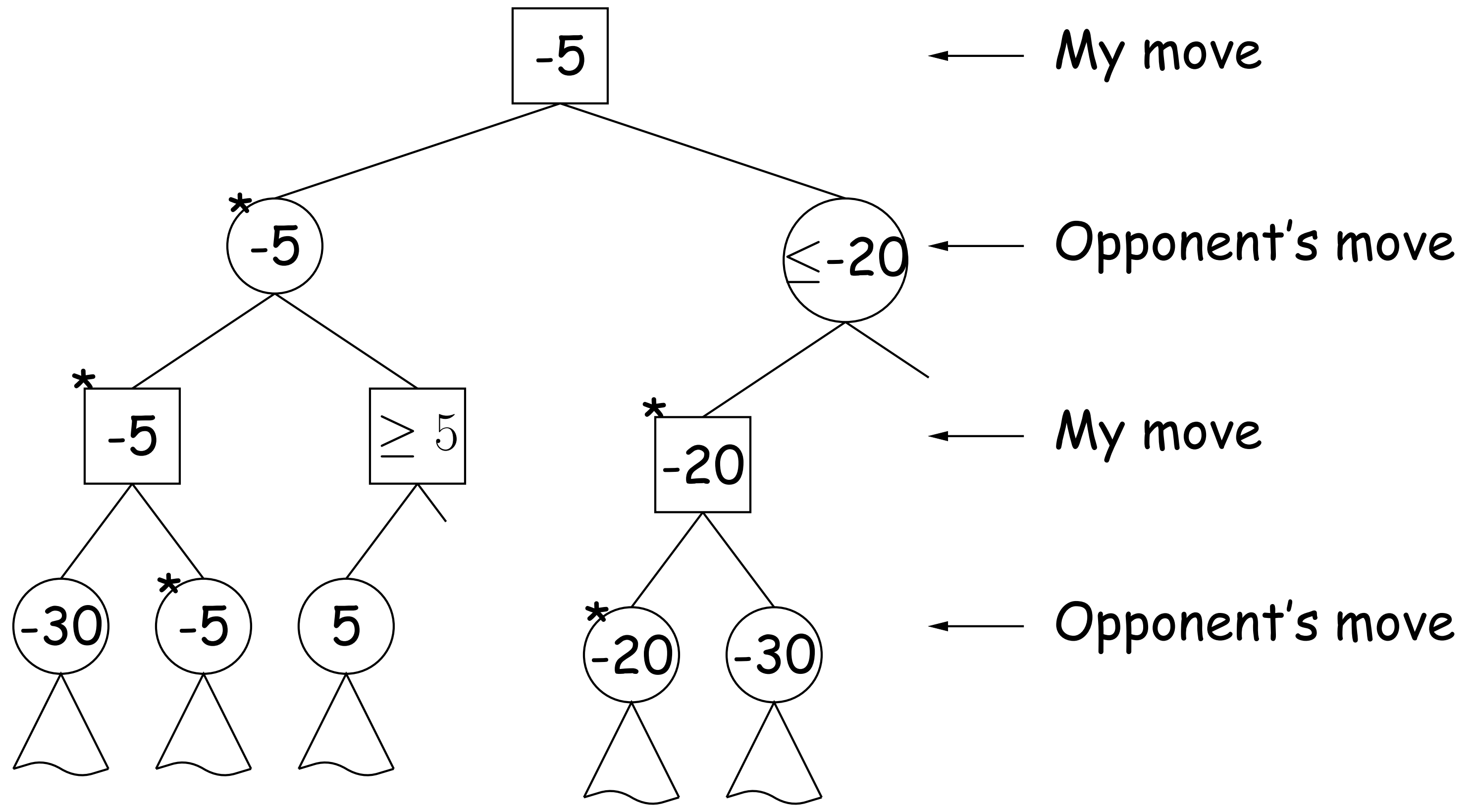
- Think of the space of possible continuations of the game as a tree.
- Each node is a position, each edge a move.



- Numbers are the values we guess for the positions (larger means better for me). Starred nodes would be chosen.
- I always choose child (next position) with maximum value; opponent chooses minimum value ("Minimax algorithm")

Alpha-Beta Pruning

- We can *prune* this tree as we search it.



- At the '≥ 5' position, I know that the opponent will not choose to move here (since he already has a -5 move).
- At the '≤ -20' position, my opponent knows that I will never choose to move here (since I already have a -5 move).

Cutting off the Search

- If you could traverse game tree to the bottom, you'd be able to force a win (if it's possible).
- Sometimes possible near the end of a game.
- Unfortunately, game trees tend to be either infinite or impossibly large.
- So, we choose a maximum *depth,* and use a heuristic value computed on the position alone (called a *static valuation*) as the value at that depth.
- Or we might use *iterative deepening*, repeating the search at increasing depths until time is up.
- Much more sophisticated searches are possible, however (take CS188).

Overall Search Algorithm

- Depending on whose move it is (maximizing player or minimizing player), we'll search for a move estimated to be optimal in one direction or the other.
- Search will be exhaustive down to a particular depth in the game tree; below that, we guess values.
- Also pass α and β limits:
  - High player does not care about exploring a position further once he knows its value is larger than what the minimizing player knows he can get (β), because the minimizing player will never allow that position to come about.
  - Likewise, minimizing player won't explore a positions whose value is less than what the maximizing player knows he can get (α).
- To start, a maximizing player will find a move with

  `findMax(`*current position*, *search depth* $-\infty$, $+\infty$)
- minimizing player:

  `findMin(`*current position*, *search depth* $-\infty$, $+\infty$)

Some Pseudocode for Searching (One Level)

- The most basic kind of game-tree search is to assign some heuristic value to any given position, looking at just the next possible move:

```
Move simpleFindMax(Position posn, double alpha, double beta) {
    if (posn.maxPlayerWon())
        return artificial "Move" with value $+\infty$;
    else if (posn.minPlayerWon())
        return artificial "Move" with value $-\infty$;
    Move bestSoFar = artificial "Move" with value $-\infty$;
    for (each M = a legal move for maximizing player from posn) {
         Position next = posn.makeMove(M);
         next.setValue(heuristicEstimate(next));
         if (next.value() >= bestSoFar.value()) {
              bestSoFar = next;
              alpha = max(alpha, next.value());
              if (beta <= alpha) break;
         }
    }
    return bestSoFar;
}
```

One-Level Search for Minimizing Player

```
Move simpleFindMin(Position posn, double alpha, double beta) {
    if (posn.maxPlayerWon())
        return artificial "Move" with value $+\infty$;
    else if (posn.minPlayerWon())
        return artificial "Move" with value $-\infty$;
    Move bestSoFar = artificial "Move" with value $+\infty$;
    for (each M = a legal move for minimizing player from posn) {
         Position next = posn.makeMove(M);
         next.setValue(heuristicEstimate(next));
         if (next.value() <= bestSoFar.value()) {
              bestSoFar = next;
              beta = min(beta, next.value());
              if (beta <= alpha) break;
         }
    }
    return bestSoFar;
}
```

Some Pseudocode for Searching (Maximizing Player)

```
/** Return a best move for maximizing player from POSN, searching
 *  to depth DEPTH.  Any move with value >= BETA is also
 *  "good enough". */
Move findMax(Position posn, int depth, double alpha, double beta) {
    if (depth == 0 || gameOver(posn))
        return simpleFindMax(posn, alpha, beta);
    Move bestSoFar = artificial “Move” with value $-\infty$;
    for (each M = a legal move for maximizing player from posn) {
         Position next = posn.makeMove(M);
         Move response = findMin(next, depth-1, alpha, beta);
         if (response.value() >= bestSoFar.value()) {
              bestSoFar = next;
              next.setValue(response.value());
              alpha = max(alpha, response.value());
              if (beta <= alpha) break;
         }
    }
    return bestSoFar;
}
```

Some Pseudocode for Searching (Minimizing Player)

```
/** Return a best move for minimizing player from POSN, searching
 *  to depth DEPTH.  Any move with value <= ALPHA is also
 *  "good enough". */
Move findMin(Position posn, int depth, double alpha, double beta) {
    if (depth == 0 || gameOver(posn))
        return simpleFindMin(posn, alpha, beta);
    Move bestSoFar = artificial “Move” with value $+\infty$;
    for (each M = a legal move for minimizing player from posn) {
         Position next = posn.makeMove(M);
         Move response = findMax(next, depth-1, alpha, beta);
         if (response.value() <= bestSoFar.value()) {
              bestSoFar = next;
              next.setValue(response.value());
              beta = min(beta, response.value());
              if (beta <= alpha) break;
         }
    }
    return bestSoFar;
}
```

CS61B Lecture #23

Today:

- Priority queues (*Data Structures* §6.4, §6.5)
- Range queries (§6.2)
- Java utilities: `SortedSet`, `Map`, etc.

Next topic: Hashing (*Data Structures* Chapter 7).

Priority Queues, Heaps

- Priority queue: defined by operations "add," "find largest," "remove largest."
- Examples: scheduling long streams of actions to occur at various future times.
- Also useful for sorting (keep removing largest).
- Common implementation is the *heap*, a kind of tree.
- (Confusingly, this same term is used to described the pool of storage that the **new** operator uses. Sorry about that.)

Heaps

- A *max-heap* is a binary tree that enforces the

  Heap Property: Labels of *both* children of each node are less than node's label.

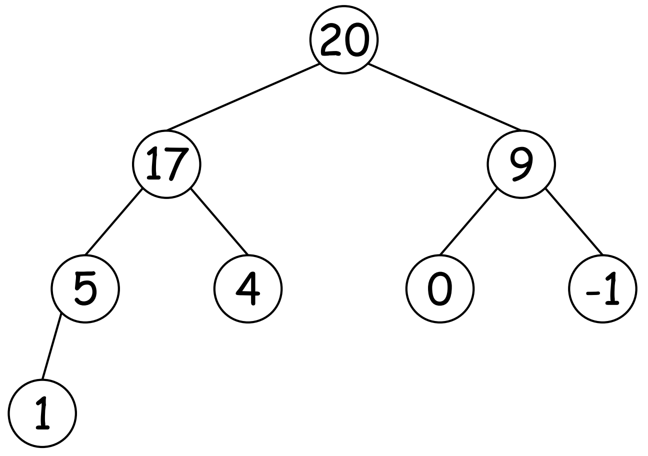
- So node at top has largest label.
- Looser than binary search property, which allows us to keep tree "bushy".
- That is, it's always valid to put the smallest nodes anywhere at the bottom of the tree.
- Thus, heaps can be made *nearly complete:* all but possibly the last row have as many keys as possible.
- As a result, insertion of new value and deletion of largest value always take time proportional to $\lg N$ in worst case.
- A *min-heap* is basically the same, but with the minimum value at the root and children having larger values than their parents.

Example: Inserting into a simple heap

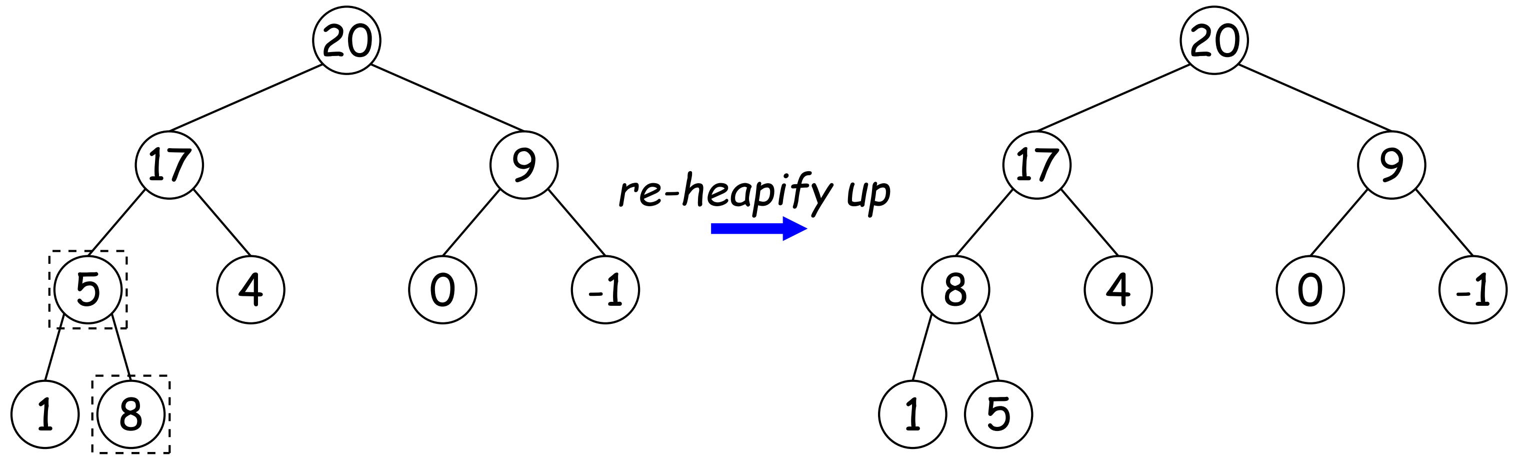
Data:

1 17 4 5 9 0 −1 20

Initial Heap:

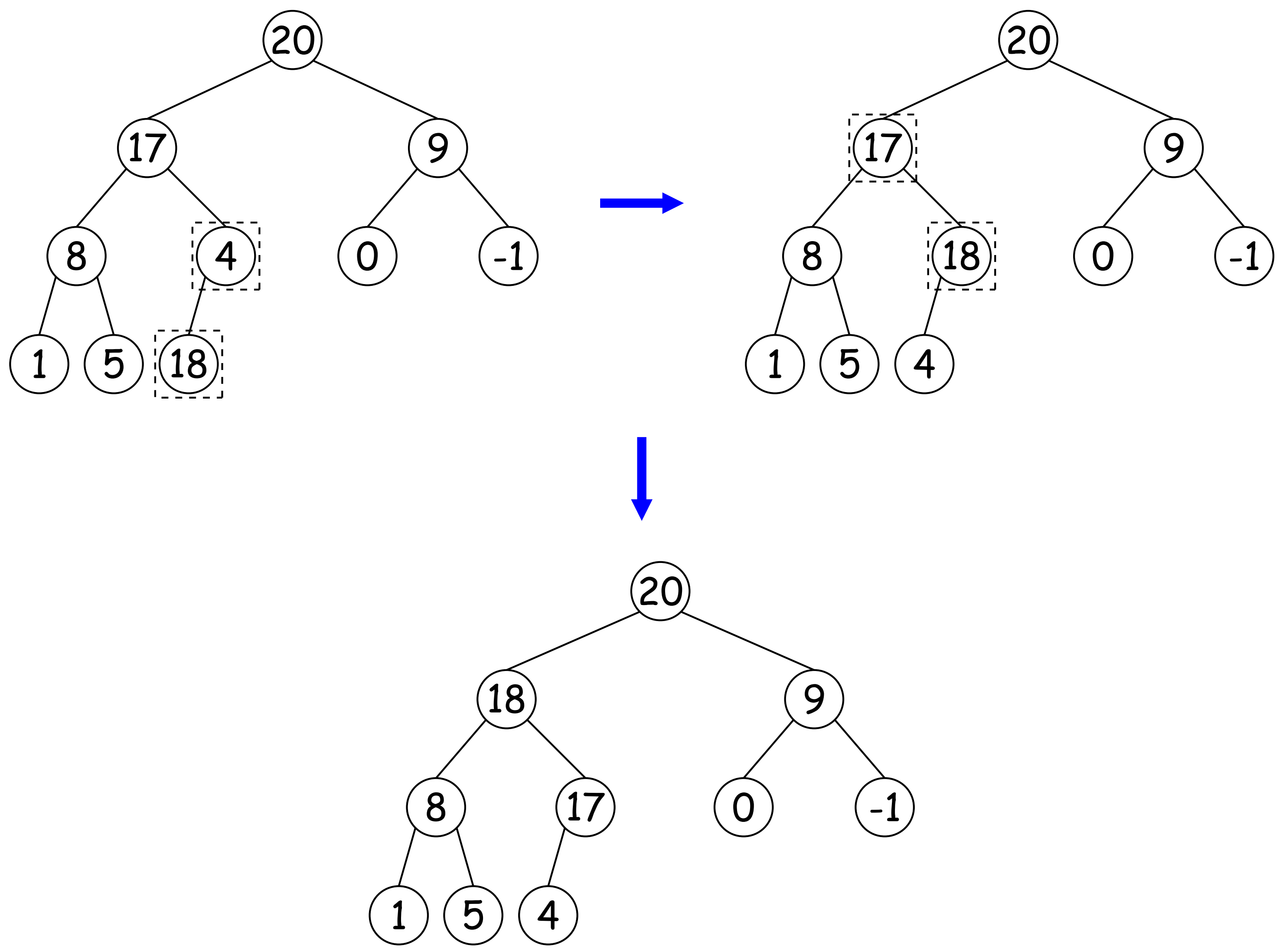


Add 8: Dashed boxes show where heap property violated



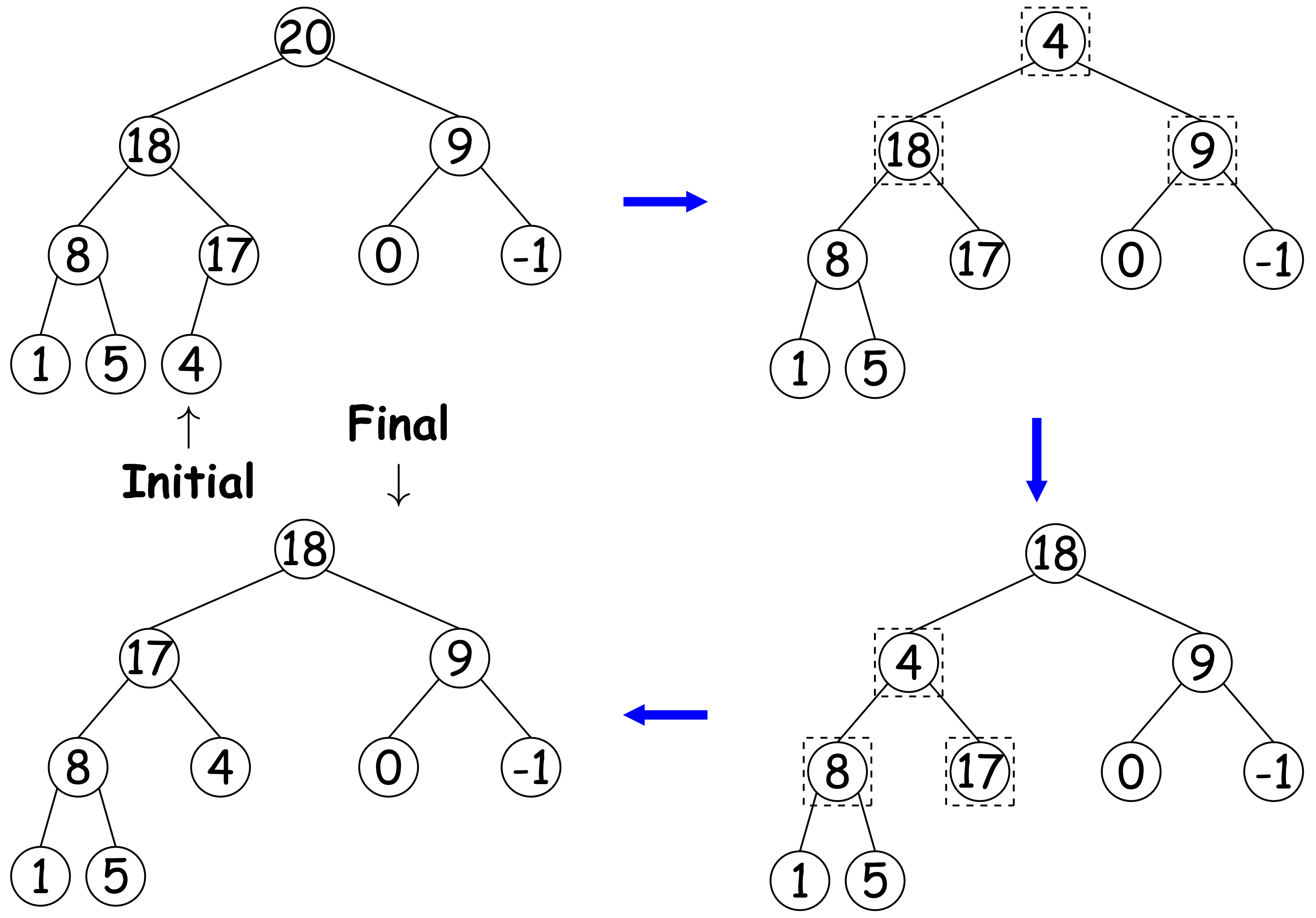
Heap insertion continued

Now insert 18:



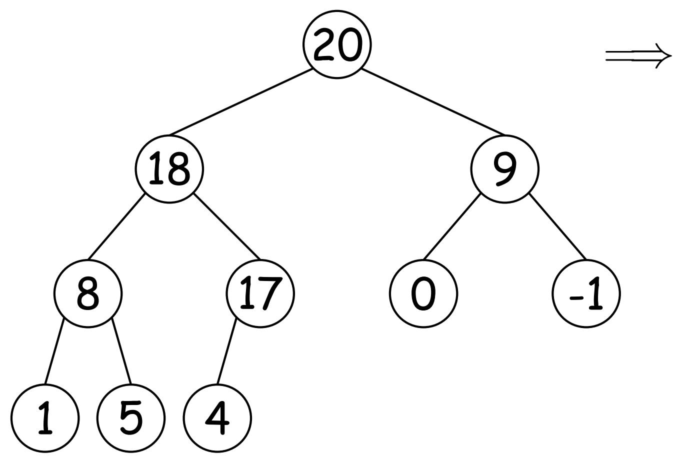
Removing Largest from Heap

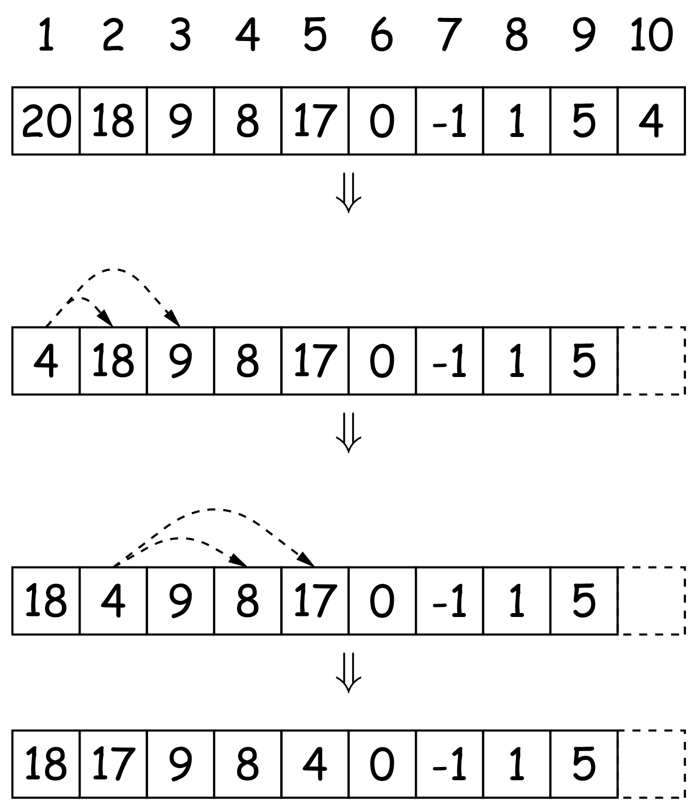
To remove largest: Move bottommost, rightmost node to top, then re-heapify down as needed (swap offending node with larger child) to re-establish heap property.



Heaps in Arrays

- Since heaps are nearly complete (missing items only at bottom level), can use arrays for compact representation.
- Example of removal from last slide (dashed arrows show children):





Nodes stored in level order.
Children of node at index $\#K$ are in $2K$ and $2K+1$ if numbering from 1, or $2K+1$ and $2K+2$ if from 0.

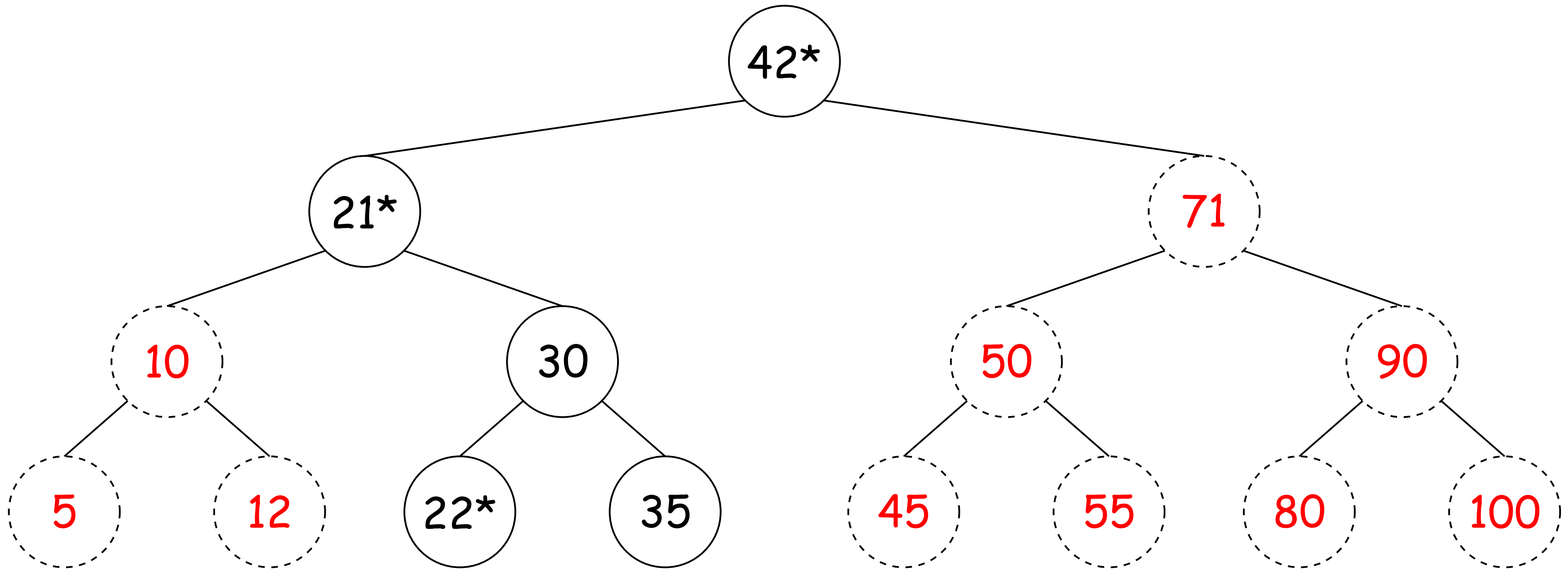
Ranges

- So far, have looked for specific items
- But for BSTs, need an ordering anyway, and can also support looking for *ranges of values*.
- Example: perform some action on all values in a BST that are within some range (in natural order):

```
/** Apply WHATTODO to all labels in T that are >= L and < U,
 *  in ascending natural order. */
static void visitRange(BST<String> T, String L, String U,
                       Consumer<BST<String>> whatToDo) {
  if (T != null) {
    int compLeft = L.compareTo(T.label ()),
        compRight = U.compareTo(T.label ());
    if (compLeft < 0)                  /* L < label */
      visitRange (T.left(), L, U, whatToDo);
    if (compLeft <= 0 && compRight > 0) /* L <= label < U */
      whatToDo.accept(T);
    if (compRight > 0)                 /* label < U */
      visitRange (T.right (), L, U, whatToDo);
  }
}
```

Time for Range Queries

- Time for range query $\in O(h + M)$, where h is height of tree, and M is number of data items that turn out to be in the range.
- Consider searching the tree below for all values $25 \leq x < 40$.
- Dashed nodes are never looked at. Starred nodes are looked at but not output. The h comes from the starred nodes; the M comes from unstarred non-dashed nodes.



Ordered Sets and Range Queries in Java

- Class `SortedSet` supports range queries with *views* of set:
  - `S.headSet(U)`: subset of `S` that is $<$ `U`.
  - `S.tailSet(L)`: subset that is $\geq$ `L`.
  - `S.subSet(L,U)`: subset that is $\geq$ `L`, $<$ `U`.
- Changes to views modify `S`.
- Attempts to, *e.g.*, add to a `headSet` beyond `U` are disallowed.
- Can iterate through a view to process a range:

```
SortedSet<String> fauna = new TreeSet<String>
      (Arrays.asList ("axolotl", "elk", "dog", "hartebeest", "duck"));
for (String item : fauna.subSet ("bison", "gnu"))
     System.out.printf ("%s, ", item);
```

would print "`dog, duck, elk,`"

TreeSet

- Java library type `TreeSet<T>` requires either that `T` be `Comparable`, or that you provide a Comparator, as in:

```
SortedSet<String> rev_fauna = new TreeSet<String>(Collections.reverseOrder());
```

- Comparator is a type of function object:

```
interface Comparator<T> {
    /** Return <0 if LEFT<RIGHT, >0 if LEFT>RIGHT, else 0. */
    int compare(T left, T right);
}
```

  (We'll deal with what `Comparator<T extends Comparable<T>>` is all about later.)

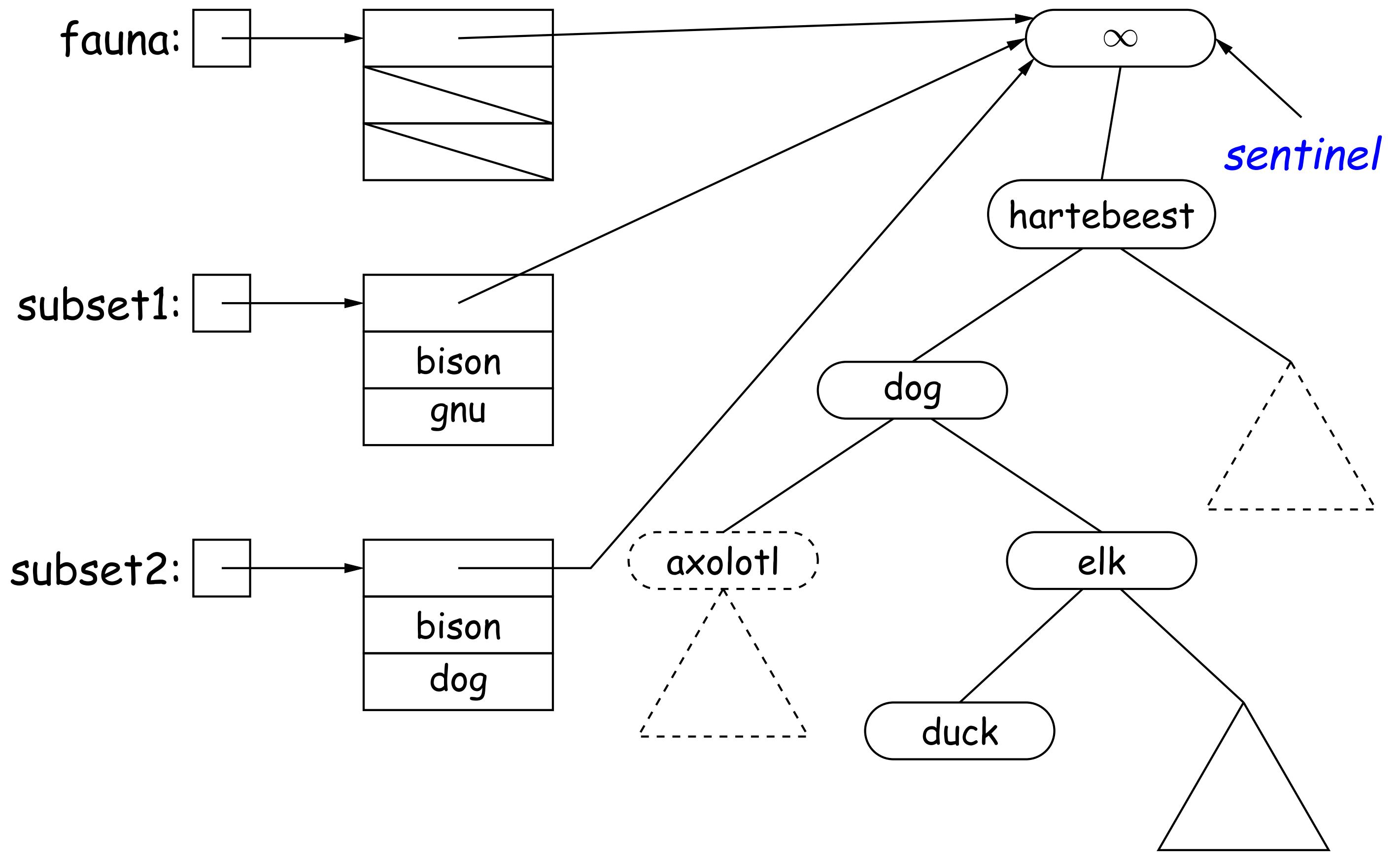
- For example, the `reverseOrder` comparator is defined like this:

```
/** A Comparator that gives the reverse of natural order. */
static <T extends Comparable<T>> Comparator<T> reverseOrder() {
   // Java figures out this lambda expression is a Comparable<T>.
   return (x, y) -> y.compareTo(x);
}
```

Example of Representation: BSTSet

- Same representation for both sets and subsets.
- Pointer to BST, plus bounds (if any).
- `.size()` is expensive!

```
SortedSet<String>
    fauna = new BSTSet<String>(stuff);
    subset1 = fauna.subSet("bison","gnu");
    subset2 = subset1.subSet("axolotl","dog");
```



CS61B Lecture #24: Hashing

Back to Simple Search

- Linear search is OK for small data sets, bad for large.
- So linear search would be OK *if* we could rapidly narrow the search to a few items.
- Suppose that in constant time could put any item in our data set into a numbered *bucket,* where # buckets stays within a constant factor of # keys.
- Suppose also that buckets contain roughly equal numbers of keys.
- Then search would be constant time.

Hash functions

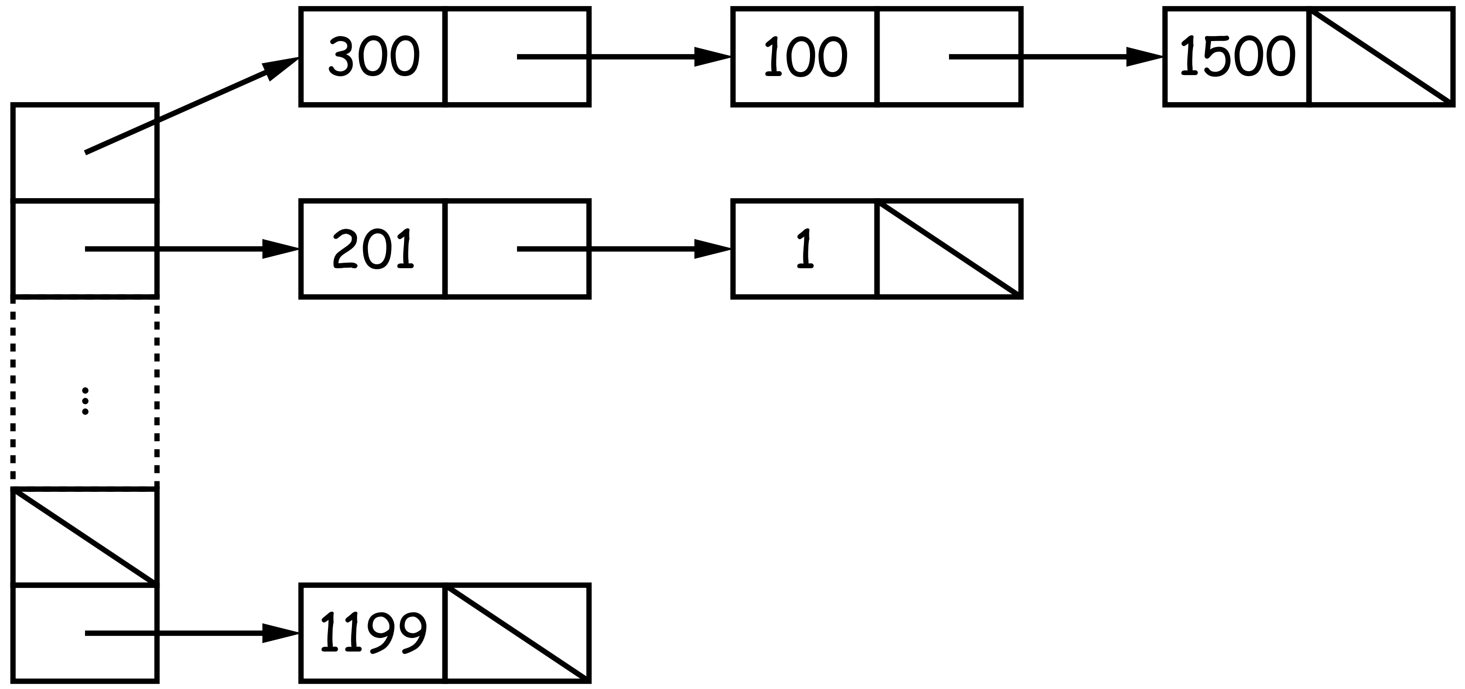
- To do this, must have way to convert key to bucket number: a *hash function*.

  "**hash** /hæʃ/ *2a* a mixture; a jumble. *b* a mess." *Concise Oxford Dictionary, eighth edition*

- Example:
  - $N = 200$ data items.
  - keys are `longs`, evenly spread over the range $0..2^{63} - 1$.
  - Want to keep maximum search to $L = 2$ items.
  - Use hash function $h(K) = K\%M$, where $M = N/L = 100$ is the number of buckets: $0 \leq h(K) < M$.
  - So 100232, 433, and 10002332482 go into different buckets, but 10, 400210, and 210 all go into the same bucket.

External chaining

- Array of M buckets.
- Each bucket is a list of data items.



- Not all buckets have same length, but average is $N/M = L$, the *load factor*.
- To work well, hash function must avoid *collisions:* keys that "hash" to equal values.

Ditching the Chains: Open Addressing

- Idea: Put one data item in each bucket.
- When there is a collision, and bucket is full, just use another.
- Various ways to do this:
  - Linear probes: If there is a collision at $h(K)$, try $h(K)+m$, $h(K)+2m$, etc. (wrap around at end).
  - Quadratic probes: $h(K) + m$, $h(K) + m^2$, $\ldots$
  - Double hashing: $h(K) + h'(K)$, $h(K) + 2h'(K)$, etc.
- Example: $h(K) = K\%M$, with $M = 10$, linear probes with $m = 1$.
  - Add 1, 2, 11, 3, 102, 9, 18, 108, 309 to empty table.

108	1	2	11	3	102	309		18	9

- Things can get slow, even when table is far from full.
- Lots of literature on this technique, but
- Personally, I just settle for external chaining.

Filling the Table

- To get (likely to be) constant-time lookup, need to keep #buckets within constant factor of #items.
- So resize table when load factor gets higher than some limit.
- In general, must *re-hash* all table items.
- Still, this operation constant time per item,
- So by doubling table size each time, get constant *amortized* time for insertion and lookup
- (Assuming, that is, that our hash function is good).

Hash Functions: Strings

- For String, "$s_0 s_1 \cdots s_{n-1}$" want function that takes all characters and their positions into account.
- What's wrong with $s_0 + s_1 + \ldots + s_{n-1}$?
- For strings, Java uses

$$h(s) = s_0 \cdot 31^{n-1} + s_1 \cdot 31^{n-2} + \ldots + s_{n-1}$$

  computed modulo 2^{32} as in Java `int` arithmetic.
- To convert to a table index in $0..N-1$, compute `h(s)%N` (but *don't* use table size that is multiple of 31!)
- Not as hard to compute as you might think; don't even need multiplication!

```
int r; r = 0;
for (int i = 0; i < s.length (); i += 1)
   r = (r << 5) - r + s.charAt (i);
```

Hash Functions: Other Data Structures I

- Lists (`ArrayList`, `LinkedList`, etc.) are analagous to strings: e.g., Java uses

```
hashCode = 1; Iterator i = list.iterator();
while (i.hasNext()) {
   Object obj = i.next();
   hashCode =
     31*hashCode
     + (obj==null ? 0 : obj.hashCode());
}
```

- Can limit time spent computing hash function by not looking at entire list. For example: look only at first few items (if dealing with a `List` or `SortedSet`).

- Causes more collisions, but does *not* cause equal things to go to different buckets.

Hash Functions: Other Data Structures II

- Recursively defined data structures ⇒ recursively defined hash functions.
- For example, on a binary tree, one can use something like

```
hash(T):
    if (T == null)
       return 0;
    else return someHashFunction (T.label ())
                ^ hash(T.left ()) ^ hash(T.right ());
```

Identity Hash Functions

- Can use address of object ("hash on identity") if distinct (`!=`) objects are never considered equal.
- But careful! Won't work for Strings, because `.equal` Strings could be in different buckets:

```
String H  = "Hello",
       S1 = H + ", world!",
       S2 = "Hello, world!";
```

- Here `S1.equals(S2)`, but `S1 != S2`.

What Java Provides

- In class `Object`, is function `hashCode()`.
- By default, returns the identity hash function, or something similar. [Why is this OK as a default?]
- Can override it for your particular type.
- For reasons given on last slide, is overridden for type `String`, as well as many types in the Java library, like all kinds of `List`.
- The types `Hashtable`, `HashSet`, and `HashMap` use `hashCode` to give you fast look-up of objects.

```
HashMap<KeyType,ValueType> map =
     new HashMap<>(approximate size, load factor);
map.put(key, value);           // Map KEY -> VALUE.
... map.get(someKey)           // VALUE last mapped to by SOMEKEY.
... map.containsKey(someKey)   // Is SOMEKEY mapped?
... map.keySet()               // All keys in MAP (a Set)
```

Special Case: Monotonic Hash Functions

- Suppose our hash function is *monotonic:* either nonincreasing or nondescreasing.
- So, e.g., if key $k_1 > k_2$, then $h(k_1) \geq h(k_2)$.
- Example:
  - Items are time-stamped records; key is the time.
  - Hashing function is to have one bucket for every hour.
- In this case, you *can* use a hash table to speed up range queries [How?]
- Could this be applied to strings? When would it work well?

Perfect Hashing

- Suppose set of keys is *fixed*.
- A tailor-made hash function might then hash every key to a different value: *perfect hashing*.
- In that case, there is no search along a chain or in an open-address table: either the element at the hash value is or is not equal to the target key.
- For example, might use first, middle, and last letters of a string (read as a 3-digit base-26 numeral). Would work if those letters differ among all strings in the set.
- Or might use the Java method, but tweak the multipliers until all strings gave different results.

Characteristics

- Assuming good hash function, add, lookup, deletion take $\Theta(1)$ time, amortized.
- Good for cases where one looks up *equal* keys.
- Usually bad for range queries: “Give me every name between Martin and Napoli.” [Why?]
- Hashing is probably not a good idea for small sets that you rapidly create and discard [why?]

Comparing Search Structures

Here, N is #items, k is #answers to query.

Function	Unordered List	Sorted Array	Bushy Search Tree	"Good" Hash Table	Heap
find	$\Theta(N)$	$\Theta(\lg N)$	$\Theta(\lg N)$	$\Theta(1)$	$\Theta(N)$
add (amortized)	$\Theta(1)$	$\Theta(N)$	$\Theta(\lg N)$	$\Theta(1)$	$\Theta(\lg N)$
range query	$\Theta(N)$	$\Theta(k + \lg N)$	$\Theta(k + \lg N)$	$\Theta(N)$	$\Theta(N)$
find largest	$\Theta(N)$	$\Theta(1)$	$\Theta(\lg N)$	$\Theta(N)$	$\Theta(1)$
remove largest	$\Theta(N)$	$\Theta(1)$	$\Theta(\lg N)$	$\Theta(N)$	$\Theta(\lg N)$

CS61B Lecture #25: Java Generics

The Old Days

- Java library types such as `List` didn't used to be parameterized. All `List`s were lists of `Object`s.
- So you'd write things like this:

```
for (int i = 0; i < L.size(); i += 1)
   { String s = (String) L.get(i); ... }
```

- That is, must explicitly cast result of `L.get(i)` to let the compiler know what it is.
- Also, when calling `L.add(x)`, was no check that you put only `String`s into it.
- So, starting with 1.5, the designers tried to alleviate these perceived problems by introducing *parameterized types*, like `List<String>`.
- Unfortunately, it is not as simple as one might think.

Basic Parameterization

- From the definitions of `ArrayList` and `Map` in `java.util`:

```
public class ArrayList<Item> implements List<Item> {
   public Item get(int i) { ... }
   public boolean add(Item x) { ... }
   ...
}
public interface Map<Key, Value> {
   Value get(Key x);
   ...
}
```

- First (blue) occurrences of `Item`, `Key`, and `Value` introduce formal *type parameters*, whose "values" (which are reference types) get substituted for all the other occurrences of `Item`, `Key`, or `Value` when `ArrayList` or `Map` is "called" (as in `ArrayList<String>`, or `ArrayList<int[]>`, or `Map<String, List<Particle>>`).
- Other occurrences of `Item`, `Key`, and `Value` are uses of the formal types, just like uses of a formal parameter in the body of a function.

Type Instantiation

- *Instantiating* a generic type is analogous to calling a function.
- Consider again

```
public class ArrayList<Item> implements List<Item> {
   public Item get(int i) { ... }
   public boolean add(Item x) { ... }
   ...
}
```

- When we write `ArrayList<String>`, we get, in effect, a new type, somewhat like

```
public String_ArrayList implements List<String> {
   public String get(int i) { ... }
   public boolean add(String x) { ... }
```

- And then, likewise, `List<String>` refers to a new interface type as well.

Parameters on Methods

- Functions (methods) may also be parameterized by type. Example of use from `java.util.Collections`:

  ```
  /** A read-only list containing just ITEM. */
  static <T> List<T> singleton(T item) { ... }
  /** An unmodifiable empty list. */
  static <T> List<T> emptyList() { ... }
  ```

  The compiler figures out T in the expression `singleton(x)` by looking at the type of `x`. This is a simple example of *type inference*.

- In the call

  ```
  List<String> empty = Collections.emptyList();
  ```

  the parameters obviously don't suffice, but the compiler deduces the parameter `T` from context: it must be assignable to `String`.

Wildcards

- Consider the definition of something that counts the number of times something occurs in a collection of items. Could write this as

```
/** Number of items in C that are equal to X. */
static <T> int frequency(Collection<T> c, Object x) {
    int n; n = 0;
    for (T y : c) {
        if (x.equals(y))
            n += 1;
    }
    return n;
}
```

- But we don't really care what `T` is; we don't need to declare anything of type `T` in the body, because we could write instead

```
    ...
    for (Object y : c) {
```

- *Wildcard type parameters* say that you don't care what a type parameter is (i.e., it's any subtype of `Object`):

```
static int frequency(Collection<?> c, Object x) {...}
```

Subtyping (I)

- What are the relationships between the types

```
List<String>,  List<Object>, ArrayList<String>, ArrayList<Object>?
```

- We know that `ArrayList` $\preceq$ `List` and `String` $\preceq$ `Object` (using $\preceq$ for "is a subtype of")...
- ... So is `List<String>` $\preceq$ `List<Object>`?

Subtyping (II)

- Consider this fragment:

```
List<String> LS = new ArrayList<String>();
List<Object> LObj = LS;      // OK??
int[] A = { 1, 2 };
LObj.add(A);                 // Legal, since A is an Object
String S = LS.get(0);        // OOPS! A.get(0) is NOT a String,
                             // but spec of List<String>.get
                             // says that it is.
```

- So, having `List<String>` $\preceq$ `List<Object>` would violate *type safety:* The compiler is wrong about the type of a value.
- So in general for `T1<X>` $\preceq$ `T2<Y>`, must have `X` $=$ `Y`.
- But what about `T1` and `T2`?

Subtyping (III)

- Now consider

```
ArrayList<String> ALS = new ArrayList<String>();
List<String> LS = ALS;      // OK??
```

- In this case, everything's fine:
  - The object's dynamic type is `ArrayList<String>`.
  - Therefore, the methods expected for `LS` must be a subset of those for `ALS`.
  - And since the type parameters are the same, the signatures of those methods will be the same.
  - Therefore, all the legal calls on methods of `LS` (according to the compiler) will be valid for the actual object pointed to by `LS`.
- In general, `T1<X>` $\preceq$ `T2<X>` if `T1` $\preceq$ `T2`.

A Java Inconsistency: Arrays

- The Java language design is not entirely consistent when it comes to subtyping.
- For the same reason that `ArrayList<String>` $\not\preceq$ `ArrayList<Object>`, you'd also expect that `String[]` $\not\preceq$ `Object[]`.
- And yet, Java *does* make `String[]` $\preceq$ `Object[]`.
- And, just as explained above, one gets into trouble with

```
String[] AS = new String[3];
Object[] AObj = AS;
AObj[0] = new int[] { 1, 2 };  // Bad
```

- So in Java, the Bad line causes an `ArrayStoreException`—a (dynamic) runtime error instead of a (static) compile-time error.
- Why do it this way? Basically, because otherwise there'd be no way to implement, e.g., `ArrayList`.

Type Bounds (I)

- Sometimes, your program needs to ensure that a particular type parameter is replaced only by a subtype (or supertype) of a particular type (sort of like specifying the "type of a type.").
- For example,

```
class NumericSet<T extends Number> extends HashSet<T> {
    /** My minimal element */
    T min() { ... }
    ...
}
```

  Requires that all type parameters to `NumbericSet` must be subtypes of `Number` (the "type bound"). `T` can either extend or implement the bound, as appropriate.

Type Bounds (II)

- Another example:

```
/** Set all elements of L to X. */
static <T> void fill(List<? super T> L, T x) { ... }
```

  means that `L` can be a `List<Q>` for any `Q` as long as `T` is a subtype of (extends or implements) `Q`.

- Why didn't the library designers just define this as

```
/** Set all elements of L to X. */
static <T> void fill(List<T> L, T x) { ... }
```

  ?

Type Bounds (II)

- Another example:

```
/** Set all elements of L to X. */
static <T> void fill(List<? super T> L, T x) { ... }
```

  means that `L` can be a `List<Q>` for any `Q` as long as `T` is a subtype of (extends or implements) `Q`.

- Why didn't the library designers just define this as

```
/** Set all elements of L to X. */
static <T> void fill(List<T> L, T x) { ... }
```

  ? -

- Consider

```
static void blankIt(List<Object> L) {
    fill(L, " ");
}
```

  This would be illegal if `L` were forced to be a `List<String>`.

Type Bounds (III)

- And one more:

```
/** Search sorted list L for KEY, returning either its position (if
 *  present), or k-1, where k is where KEY should be inserted.  */
static <T> int binarySearch(List<? extends Comparable<? super T>> L,
                            T key)
```

- Here, the items of `L` have to have a type that is comparable to `T`'s or to some supertype of `T`.
- Does `L` have to be able to contain the value `key`?
- Why does this make sense?

Type Bounds (III)

- And one more:

```
/** Search sorted list L for KEY, returning either its position (if
 *  present), or k-1, where k is where KEY should be inserted.  */
static <T> int binarySearch(List<? extends Comparable<? super T>> L,
                            T key)
```

- Here, the items of L have to have a type that is comparable to T's or to some supertype of T.
- Does L have to be able to contain the value key?
- Why does this make sense?
- As long as the items in L can be compared to key, it doesn't really matter whether they might include key (not that this is often useful).

Dirty Secrets Behind the Scenes

- Java's design for parameterized types was constrained by a desire for backward compatibility.
- Actually, when you write

```
class Foo<T> {
    T x;                                   Foo<Integer> q = new Foo<Integer>();
    T mogrify(T y) { ... }                 Integer r = q.mogrify(s);
}
```

Java really gives you

```
class Foo {
    Object x;                              Foo q = new Foo();
    Object mogrify(Object y) { ... }       Integer r =
}                                              (Integer) q.mogrify((Integer) s);
```

That is, it supplies the casts automatically, and also throws in some additional checks. If it can't guarantee that all those casts will work, gives you a warning about "unsafe" constructs.

Limitations

Because of Java's design choices, there are some limitations to generic programming:

- Since all kinds of `Foo` or `List` are really the same,
  - `L instanceof List<String>` will be true when `L` is a `List<Integer>`.
  - Inside, e.g., class `Foo`, you cannot write `new T()`, `new T[]`, or `x instanceof T`.
- Primitive types are not allowed as type parameters.
  - Can't have `ArrayList<int>`, just `ArrayList<Integer>`.
  - Fortunately, automatic boxing and unboxing makes this substitution easy:

```
int sum(ArrayList<Integer> L) {
    int N;  N = 0;
    for (int x : L) { N += x; }
    return N;
}
```

  - Unfortunately, boxing and unboxing have significant costs.

CS61B Lecture #26

Today:

- Sorting algorithms: why?
- Insertion Sort.
- Inversions

Purposes of Sorting

- Sorting supports searching
- Binary search standard example
- Also supports other kinds of search:
  - Are there two equal items in this set?
  - Are there two items in this set that both have the same value for property X?
  - What are my nearest neighbors?
- Used in numerous unexpected algorithms, such as convex hull (smallest convex polygon enclosing set of points).

Some Definitions

- A *sorting algorithm* (or *sort*) *permutes* (re-arranges) a sequence of elements to brings them into order, according to some *total order.*
- A total order, $\preceq$, is:
  - **Total:** $x \preceq y$ or $y \preceq x$ for all x, y.
  - **Reflexive:** $x \preceq x$;
  - **Antisymmetric:** $x \preceq y$ and $y \preceq x$ iff $x = y$.
  - **Transitive:** $x \preceq y$ and $y \preceq z$ implies $x \preceq z$.
- However, our orderings may treat unequal items as equivalent:
  - E.g., there can be two dictionary definitions for the same word. If we sort only by the word being defined (ignoring the definition), then sorting could put either entry first.
  - A sort that does not change the relative order of equivalent entries (compared to the input) is called *stable.*

Classifications

- *Internal sorts* keep all data in primary memory.
- *External sorts* process large amounts of data in batches, keeping what won't fit in secondary storage (in the old days, tapes).
- *Comparison-based* sorting assumes only thing we know about keys is their order.
- *Radix sorting* uses more information about key structure.
- *Insertion sorting* works by repeatedly inserting items at their appropriate positions in the sorted sequence being constructed.
- *Selection sorting* works by repeatedly selecting the next larger (smaller) item in order and adding it to one end of the sorted sequence being constructed.

Sorting Arrays of Primitive Types in the Java Library

- The java library provides static methods to sort arrays in the class `java.util.Arrays`.
- For each primitive type `P` other than `boolean`, there are

```
/** Sort all elements of ARR into non-descending order. */
static void sort(P[] arr) { ... }

/** Sort elements FIRST .. END-1 of ARR into non-descending
 *  order. */
static void sort(P[] arr, int first, int end) { ... }

/** Sort all elements of ARR into non-descending order,
 *  possibly using multiprocessing for speed. */
static void parallelSort(P[] arr) { ... }

/** Sort elements FIRST .. END-1 of ARR into non-descending
 *  order, possibly using multiprocessing for speed. */
static void parallelSort(P[] arr, int first, int end) {...}
```

Sorting Arrays of Reference Types in the Java Library

- For reference types, `C`, that have a *natural order* (that is, that implement `java.lang.Comparable`), we have four analogous methods (one-argument `sort`, three-argument `sort`, and two `parallelSort` methods):

  ```
  /** Sort all elements of ARR stably into non-descending
   * order. */
  static <C extends Comparable<? super C>> sort(C[] arr) {...}
  ```
  etc.

- And for all reference types, `R`, we have four more:

  ```
  /** Sort all elements of ARR stably into non-descending order
   *  according to the ordering defined by COMP. */
  static <R> void sort(R[] arr, Comparator<? super R> comp) {...}
  ```
  etc.

- **Q:** Why the fancy generic arguments?

Sorting Arrays of Reference Types in the Java Library

- For reference types, `C`, that have a *natural order* (that is, that implement `java.lang.Comparable`), we have four analogous methods (one-argument `sort`, three-argument `sort`, and two `parallelSort` methods):

  ```
  /** Sort all elements of ARR stably into non-descending
   * order. */
  static <C extends Comparable<? super C>> sort(C[] arr) {...}
  ```
  etc.

- And for all reference types, `R`, we have four more:

  ```
  /** Sort all elements of ARR stably into non-descending order
   *  according to the ordering defined by COMP. */
  static <R> void sort(R[] arr, Comparator<? super R> comp) {...}
  ```
  etc.

- **Q:** Why the fancy generic arguments?
- **A:** We want to allow types that have `compareTo` methods that apply also to more general types.

Sorting Lists in the Java Library

- The class `java.util.Collections` contains two methods similar to the sorting methods for arrays of reference types:

```
/** Sort all elements of LST stably into non-descending
 *  order. */
static <C extends Comparable<? super C>> sort(List<C> lst) {...}
```

  etc.

```
/** Sort all elements of LST stably into non-descending
 *  order according to the ordering defined by COMP. */
static <R> void sort(List<R> , Comparator<? super R> comp) {...}
```

  etc.

- Also an instance method in the `List<R>` interface itself:

```
/** Sort all elements of LST stably into non-descending
 *  order according to the ordering defined by COMP. */
void sort(Comparator<? super R> comp) {...}
```

Examples

- Assume:

```
import static java.util.Arrays.*;
import static java.util.Collections.*;
```

- Sort `X`, a `String[]` or `List<String>`, into non-descending order:

```
sort(X);     // or ...
```

- Sort `X` into reverse order (Java 8):

```
sort(X, (String x, String y) -> { return y.compareTo(x); });
// or
sort(X, Collections.reverseOrder());   // or
X.sort(Collections.reverseOrder());    // for X a List
```

- Sort `X[10], ..., X[100]` in array or `List X` (rest unchanged):

```
sort(X, 10, 101);
```

- Sort `L[10], ..., L[100]` in list `L` (rest unchanged):

```
sort(L.sublist(10, 101));
```

Sorting by Insertion

- Simple idea:
  - starting with empty sequence of outputs.
  - add each item from input, *inserting* into output sequence at right point.
- Very simple, good for small sets of data.
- With vector or linked list, time for find + insert of one item is at worst $\Theta(k)$, where k is # of outputs so far.
- This gives us a $\Theta(N^2)$ algorithm (worst case as usual).
- Can we say more?

Inversions

- Can run in $\Theta(N)$ comparisons if already sorted.
- Consider a typical implementation for arrays:

```
for (int i = 1; i < A.length; i += 1) {
  int j;
  Object x = A[i];
  for (j = i-1; j >= 0; j -= 1) {
    if (A[j].compareTo(x) <= 0)  /* (1) */
      break;
    A[j+1] = A[j];               /* (2) */
  }
  A[j+1] = x;
}
```

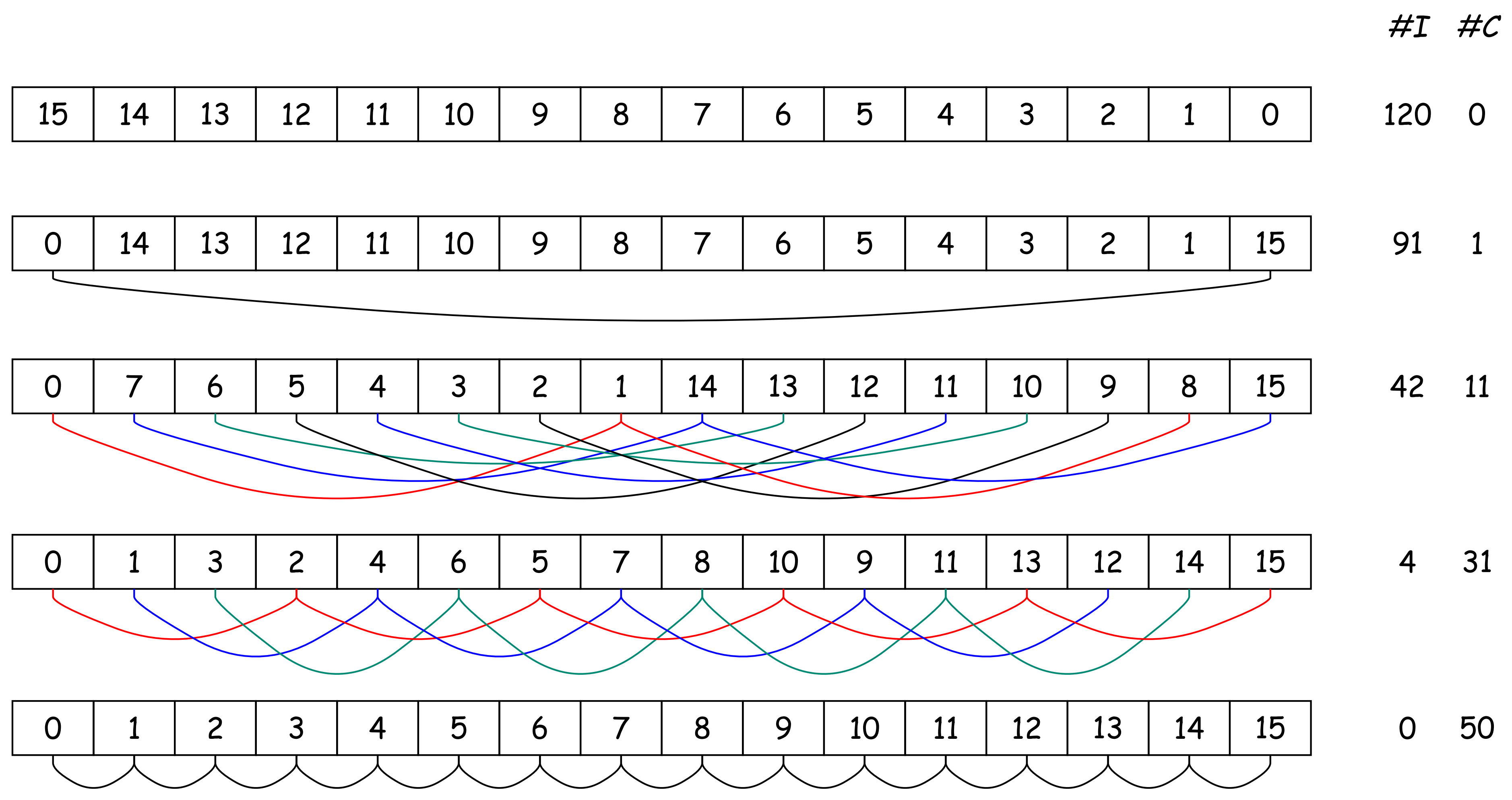
- #times (1) executes for each `j` $\approx$ how far `x` must move.
- If all items within K of proper places, then takes $O(KN)$ operations.
- Thus good for any amount of *nearly sorted* data.
- One measure of unsortedness: # of *inversions:* pairs that are out of order (= 0 when sorted, $N(N-1)/2$ when reversed).
- Each execution of (2) decreases inversions by 1.

Shell's sort

Idea: Improve insertion sort by first sorting *distant* elements:

- First sort subsequences of elements 2^k-1 apart:
  - sort items #$0,\ 2^k-1,\ 2(2^k-1),\ 3(2^k-1),\ \ldots$, then
  - sort items #$1,\ 1+2^k-1,\ 1+2(2^k-1),\ 1+3(2^k-1),\ \ldots$, then
  - sort items #$2,\ 2+2^k-1,\ 2+2(2^k-1),\ 2+3(2^k-1),\ \ldots$, then
  - *etc.*
  - sort items #$2^k-2,\ 2(2^k-1)-1,\ 3(2^k-1)-1,\ \ldots$,
  - Each time an item moves, can reduce #inversions by as much as 2^k+1.
- Now sort subsequences of elements $2^{k-1}-1$ apart:
  - sort items #$0,\ 2^{k-1}-1,\ 2(2^{k-1}-1),\ 3(2^{k-1}-1),\ \ldots$, then
  - sort items #$1,\ 1+2^{k-1}-1,\ 1+2(2^{k-1}-1),\ 1+3(2^{k-1}-1),\ \ldots$,
  - $\vdots$
- End at plain insertion sort ($2^0=1$ apart), but with most inversions gone.
- Sort is $\Theta(N^{3/2})$ (take CS170 for why!).

Example of Shell's Sort



																#I	*#C*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	120	0
0	14	13	12	11	10	9	8	7	6	5	4	3	2	1	15	91	1
0	7	6	5	4	3	2	1	14	13	12	11	10	9	8	15	42	11
0	1	3	2	4	6	5	7	8	10	9	11	13	12	14	15	4	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	50

I: Inversions left.

C: Cumulative comparisons used to sort subsequences by insertion sort.

CS61B Lectures #27

Today:

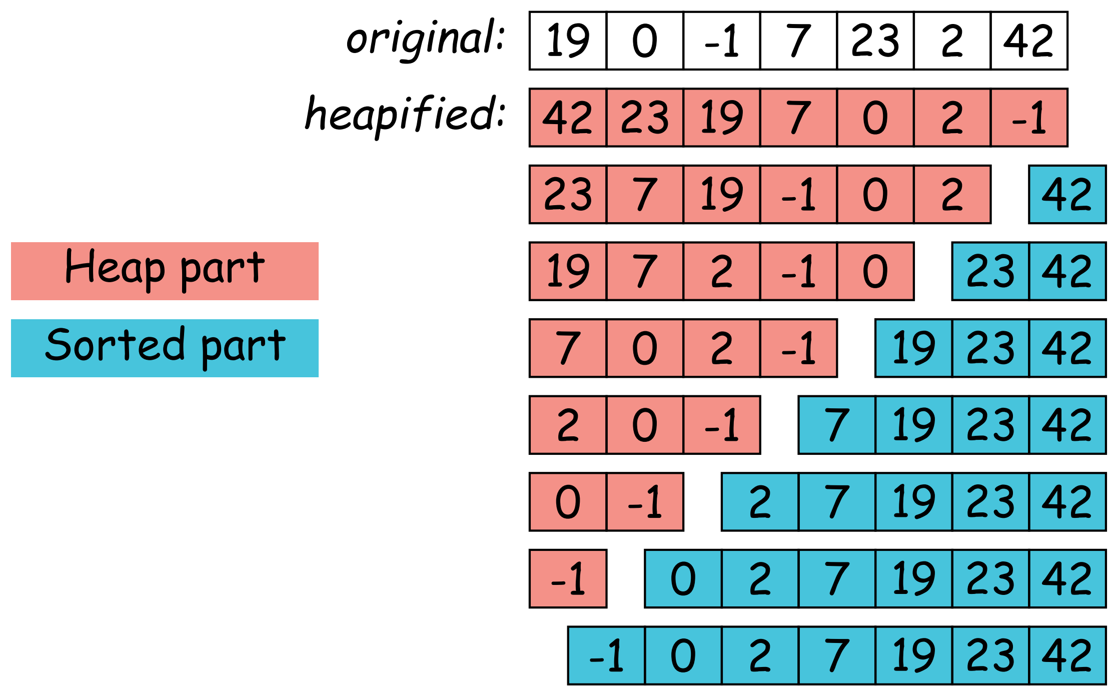
- Selection sorts, heap sort
- Merge sorts
- Quicksort

Readings: Today: *DS(IJ)*, Chapter 8; Next topic: Chapter 9.

Sorting by Selection: Heapsort

Idea: Keep selecting smallest (or largest) element.

- Really bad idea on a simple list or vector.
- But we've already seen it in action: use heap.
- Gives $O(N \lg N)$ algorithm (N remove-first operations).
- Since we remove items from end of heap, we can use that area to accumulate result:



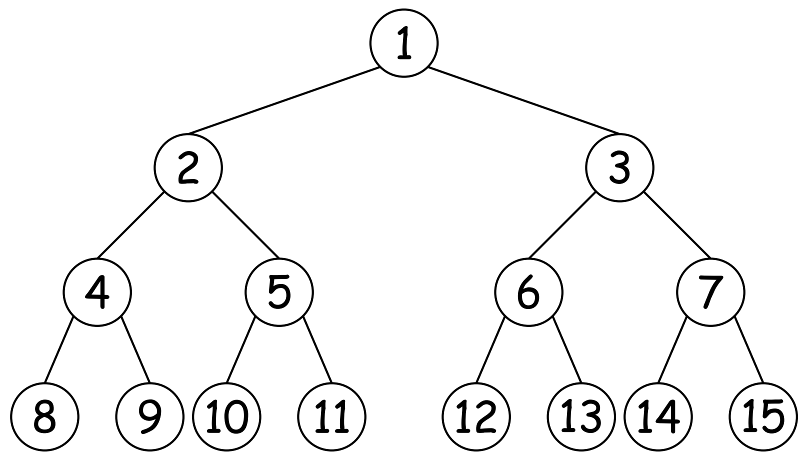
Sorting By Selection: Initial Heapifying

- When covering heaps before, we created them by insertion in an initially empty heap.
- When given an array of unheaped data to start with, there is a faster procedure (assume heap indexed from 0):

```
void heapify(int[] arr) {
    int N = arr.length;
    for (int k = N / 2; k >= 0; k -= 1) {
        for (int p = k, c = 0; 2*p + 1 < N; p = c) {
            c = 2k+1 or 2k+2, whichever is < N
                and indexes larger value in arr;
            swap elements c and k of arr;
        }
    }
}
```

- Looks like the procedure for re-inserting an element after the top element of the heap is removed, repeated $N/2$ times.
- But instead of being $\Theta(N \lg N)$, it's just $\Theta(N)$.

Cost of Creating Heap



1 node $\times$ 3 steps down

2 nodes $\times$ 2 steps down

4 nodes $\times$ 1 step down

- In general, worst-case cost for a heap with $h+1$ levels is

$$\begin{aligned} & 2^0 \cdot h + 2^1 \cdot (h-1) + \ldots + 2^{h-1} \cdot 1 \\ = \; & (2^0 + 2^1 + \ldots + 2^{h-1}) + (2^0 + 2^1 + \ldots + 2^{h-2}) + \ldots + (2^0) \\ = \; & (2^h - 1) + (2^{h-1} - 1) + \ldots + (2^1 - 1) \\ = \; & 2^{h+1} - 1 - h \\ \in \; & \Theta(2^h) = \Theta(N) \end{aligned}$$

- Alas, since the rest of heapsort still takes $\Theta(N \lg N)$, this does not improve its asymptotic cost.

Merge Sorting

Idea: Divide data in 2 equal parts; recursively sort halves; merge results.

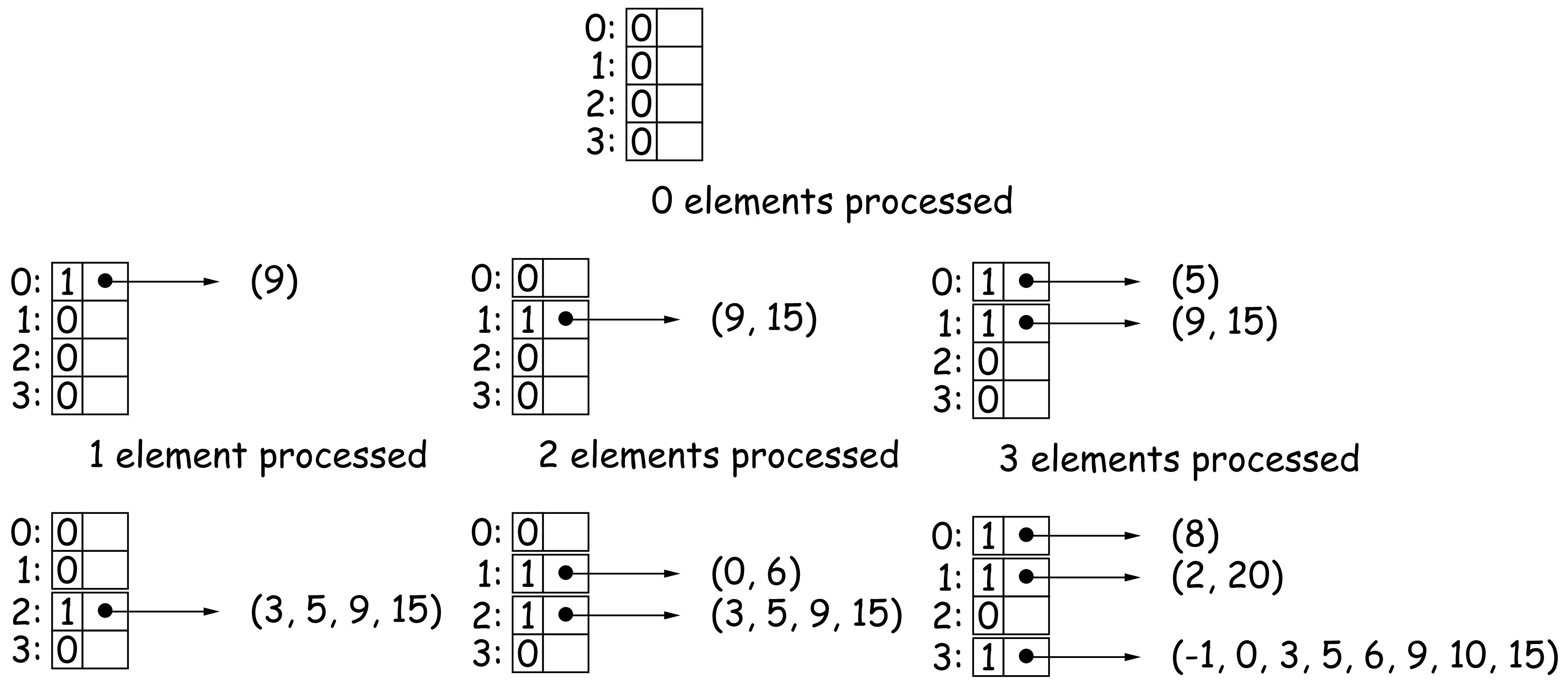
- Already seen analysis: $\Theta(N \lg N)$.
- Good for *external sorting:*
  - First break data into small enough chunks to fit in memory and sort.
  - Then repeatedly merge into bigger and bigger sequences.
- Can merge K sequences of *arbitrary size* on secondary storage using $\Theta(K)$ storage:

```
Data[] V = new Data[K];
For all i, set V[i] to the first data item of sequence i;
while there is data left to sort:
    Find k so that V[k] is smallest;
    Output V[k], and read new value into V[k] (if present).
```

Illustration of Internal Merge Sort

For internal sorting, can use a *binomial comb* to orchestrate:

L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)



0 elements processed

1 element processed

2 elements processed

3 elements processed

4 elements processed

6 elements processed

11 elements processed

Quicksort: Speed through Probability

Idea:

- *Partition* data into pieces: everything $>$ a *pivot* value at the high end of the sequence to be sorted, and everything $\leq$ on the low end.
- Repeat recursively on the high and low pieces.
- For speed, stop when pieces are "small enough" and do insertion sort on the whole thing.
- Reason: insertion sort has low constant factors. By design, no item will move out of its will move out of its piece [why?], so when pieces are small, #inversions is, too.
- Have to choose pivot well. E.g.: *median* of first, last and middle items of sequence.

Example of Quicksort

- In this example, we continue until pieces are size ≤ 4.
- Pivots for next step are starred. Arrange to move pivot to dividing line each time.
- Last step is insertion sort.

$$\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}\hline 16 & 10 & 13 & 18 & -4 & -7 & 12 & -5 & 19 & 15 & 0 & 22 & 29 & 34 & -1^* \\ \hline\end{array}$$

$$\begin{array}{|c|c|c||c||c|c|c|c|c|c|c|c|c|c|c|}\hline -4 & -5 & -7 & -1 & 18 & 13 & 12 & 10 & 19 & 15 & 0 & 22 & 29 & 34 & 16^* \\ \hline\end{array}$$

$$\begin{array}{|c|c|c||c||c|c|c|c|c||c||c|c|c|c|c|}\hline -4 & -5 & -7 & -1 & 15 & 13 & 12^* & 10 & 0 & 16 & 19^* & 22 & 29 & 34 & 18 \\ \hline\end{array}$$

$$\begin{array}{|c|c|c||c||c|c||c||c|c||c||c||c||c|c|c|}\hline -4 & -5 & -7 & -1 & 10 & 0 & 12 & 15 & 13 & 16 & 18 & 19 & 29 & 34 & 22 \\ \hline\end{array}$$

- Now everything is “close to” right, so just do insertion sort:

$$\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}\hline -7 & -5 & -4 & -1 & 0 & 10 & 12 & 13 & 15 & 16 & 18 & 19 & 22 & 29 & 34 \\ \hline\end{array}$$

Performance of Quicksort

- Probabalistic time:
  - If choice of pivots good, divide data in two each time: $\Theta(N \lg N)$ with a good constant factor relative to merge or heap sort.
  - If choice of pivots bad, most items on one side each time: $\Theta(N^2)$.
  - $\Omega(N \lg N)$ in best case, so insertion sort better for nearly ordered input sets.
- Interesting point: randomly shuffling the data before sorting makes $\Omega(N^2)$ time *very* unlikely!

Quick Selection

The Selection Problem: for given k, find kth smallest element in data.

- Obvious method: sort, select element #k, time $\Theta(N \lg N)$.
- If $k \leq$ some constant, can easily do in $\Theta(N)$ time:
  - Go through array, keep smallest k items.
- Get *probably $\Theta(N)$ time* for all k by adapting quicksort:
  - Partition around some pivot, p, as in quicksort, arrange that pivot ends up at dividing line.
  - Suppose that in the result, pivot is at index m, all elements $\leq$ pivot have indicies $\leq m$.
  - If $m = k$, you're done: p is answer.
  - If $m > k$, recursively select kth from left half of sequence.
  - If $m < k$, recursively select $(k - m - 1)$th from right half of sequence.

Selection Example

Problem: Find just item #10 in the sorted version of array:

Initial contents:

51	60	21	-4	37	4	49	10	40*	59	0	13	2	39	11	46	31
0																

Looking for #10 to left of pivot 40:

13	31	21	-4	37	4*	11	10	39	2	0	‖	40	‖	59	51	49	46	60
0																		

Looking for #6 to right of pivot 4:

-4	0	2	‖	4	‖	37	13	11	10	39	21	31*	‖	40	‖	59	51	49	46	60
						4														

Looking for #1 to right of pivot 31:

-4	0	2	‖	4	‖	21	13	11	10	‖	31	‖	39	37	‖	40	‖	59	51	49	46	60
													9									

Just two elements; just sort and return #1:

-4	0	2	‖	4	‖	21	13	11	10	‖	31	‖	37	39	‖	40	‖	59	51	49	46	60
													9									

Result: 39

Selection Performance

- For this algorithm, if m roughly in middle each time, cost is

$$\begin{aligned} C(N) &= \begin{cases} 1, & \text{if } N = 1, \\ N + C(N/2), & \text{otherwise.} \end{cases} \\ &= N + N/2 + \ldots + 1 \\ &= 2N - 1 \in \Theta(N) \end{aligned}$$

- But in worst case, get $\Theta(N^2)$, as for quicksort.
- By another, non-obvious algorithm, can get $\Theta(N)$ worst-case time for all k (take CS170).

CS61B Lectures #28

Today:

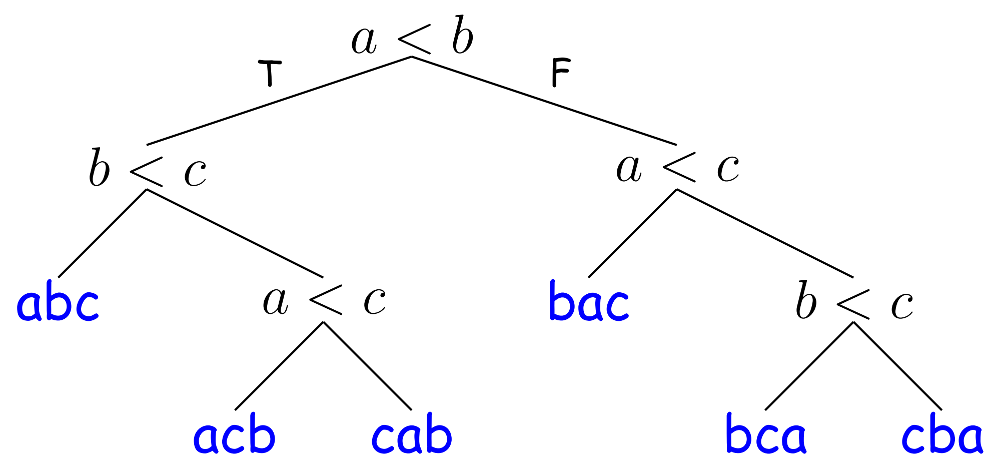
- Lower bounds on sorting by comparison
- Distribution counting, radix sorts

Readings: Today: *DS(IJ)*, Chapter 8; Next topic: Chapter 9.

Better than N lg N?

- Can prove that *if all you can do to keys is compare them,* then sorting must take $\Omega(N \lg N)$.
- Basic idea: there are $N!$ possible ways the input data could be scrambled.
- Therefore, your program must be prepared to do $N!$ different combinations of data-moving operations.
- Therefore, there must be $N!$ possible combinations of outcomes of all the **if**-tests in your program, since those determine what move gets moved where (we're assuming that comparisons are 2-way).

Decision Tree
Height $\propto$ Sorting time



Necessary Choices

- Since each **if**-test goes two ways, number of possible different outcomes for k **if**-tests is 2^k.
- Thus, need enough tests so that $2^k \geq N!$, which means $k \in \Omega(\lg N!)$.
- Using Stirling's approximation,

$$\begin{aligned} N! &\in \sqrt{2\pi N}\left(\frac{N}{e}\right)^N\left(1+\Theta\left(\frac{1}{N}\right)\right), \\ \lg(N!) &\in 1/2(\lg 2\pi + \lg N) + N\lg N - N\lg e + \lg\left(1+\Theta\left(\frac{1}{N}\right)\right) \\ &= \Theta(N\lg N) \end{aligned}$$

- This tells us that k, the worst-case number of tests needed to sort N items by comparison sorting, is in $\Omega(N\lg N)$: there must be cases where we need (some multiple of) $N\lg N$ comparisons to sort N things.

Beyond Comparison: Distribution

- But suppose can do more than compare keys?
- For example, how can we sort a set of N integer keys whose values range from 0 to kN, for some small constant k?
- One technique: put the integers into N buckets, with an integer p going to bucket $\lfloor p/k \rfloor$.
- At most k keys per bucket, so catenate and use insertion sort, which will now be fast.
- E.g., $k = 2, N = 10:$

```
Start:
  14  3   10  13  4   2   19  17  0   9
In buckets:
  | 0 | 3  2 |  4 |   |  9  | 10 | 13 | 14 |  17 |  19  |
```

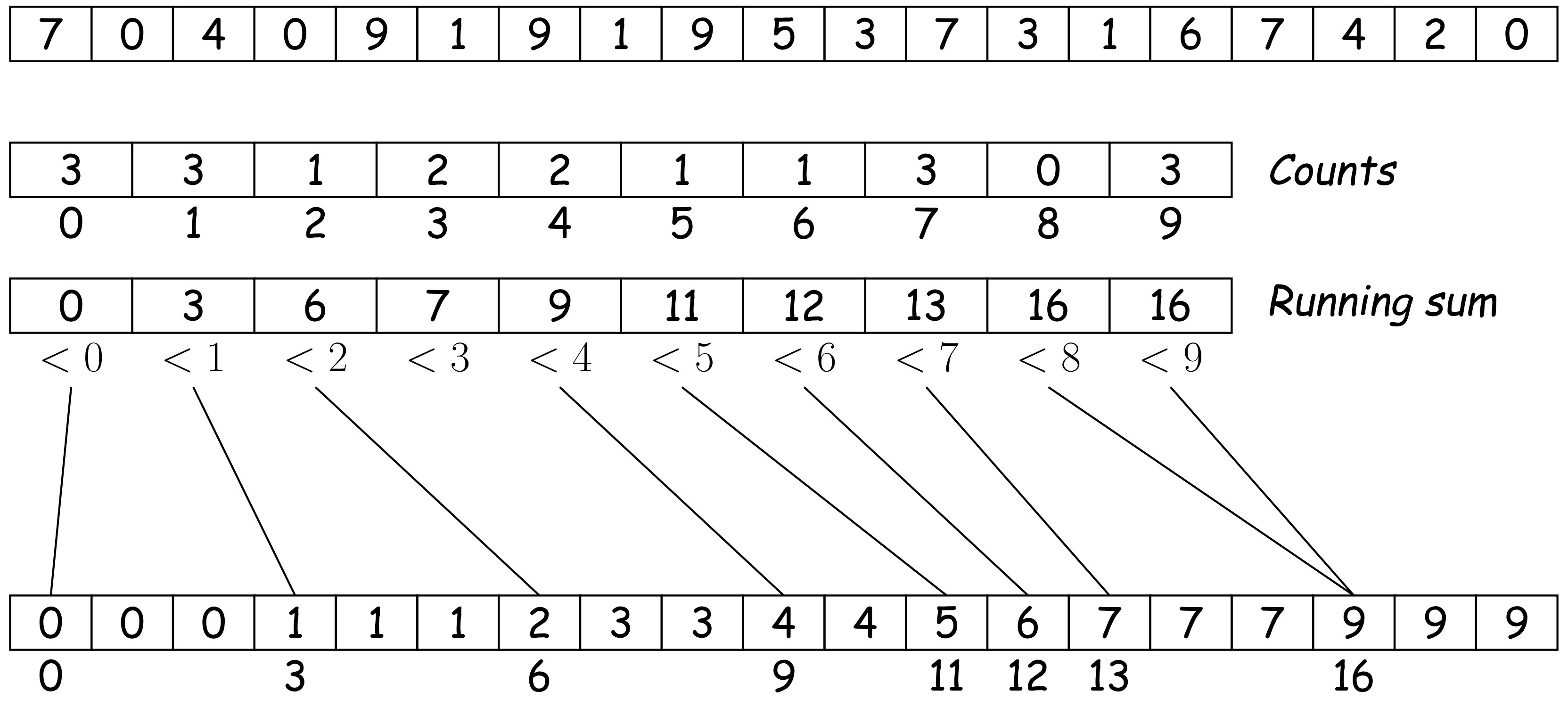
- Now insertion sort is fast. Putting in buckets takes time $\Theta(N)$, and insertion sort takes $\Theta(kN)$. When k is fixed (constant), we have sorting in time $\Theta(N)$.

Distribution Counting

- Another technique: *count* the number of items < 1, < 2, etc.
- If M_p = #items with value $< p$, then in sorted order, the j^{th} item with value p must be item #$M_p + j$.
- Gives another *linear-time* algorithm.

Distribution Counting Example

- Suppose all items are between 0 and 9 as in this example:



- "Counts" line gives # occurrences of each key.
- "Running sum" gives cumulative count of keys $<$ each value. . .
- . . . which tells us where to put each key:
- The first instance of key k goes into slot m, where m is the number of key instances that are $< k$.

Distribution Counting Example (II)

7	0	4	0	9	1	9	1	9	5	3	7	3	1	6	7	4	2	0

3	3	1	2	2	1	1	3	0	3	*Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	13	16	16	*Running sum of Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	13	16	16	*Next positions*
0	1	2	3	4	5	6	7	8	9	

																			Output
0			3			6			9			12			15			18	

Distribution Counting Example (II)

7	0	4	0	9	1	9	1	9	5	3	7	3	1	6	7	4	2	0

3	3	1	2	2	1	1	3	0	3	*Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	13	16	16	*Running sum of Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	14	16	16	*Next positions*
0	1	2	3	4	5	6	7	8	9	

													7						Output
0			3			6			9			12			15			18	

Distribution Counting Example (II)

7	0	4	0	9	1	9	1	9	5	3	7	3	1	6	7	4	2	0

3	3	1	2	2	1	1	3	0	3	*Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	13	16	16	*Running sum of Counts*
0	1	2	3	4	5	6	7	8	9	

1	3	6	7	9	11	12	14	16	16	*Next positions*
0	1	2	3	4	5	6	7	8	9	

0													7						Output
0			3			6			9			12			15			18	

Distribution Counting Example (II)

7	0	4	0	9	1	9	1	9	5	3	7	3	1	6	7	4	2	0

3	3	1	2	2	1	1	3	0	3	*Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	13	16	16	*Running sum of Counts*
0	1	2	3	4	5	6	7	8	9	

1	3	6	7	10	11	12	14	16	16	*Next positions*
0	1	2	3	4	5	6	7	8	9	

0									4				7						Output
0			3			6			9			12			15			18	

Distribution Counting Example (II)

7	0	4	0	9	1	9	1	9	5	3	7	3	1	6	7	4	2	0

3	3	1	2	2	1	1	3	0	3	*Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	13	16	16	*Running sum of Counts*
0	1	2	3	4	5	6	7	8	9	

2	3	6	7	10	11	12	14	16	16	*Next positions*
0	1	2	3	4	5	6	7	8	9	

0	0								4				7						Output
0			3			6			9			12			15			18	

Distribution Counting Example (II)

7	0	4	0	9	1	9	1	9	5	3	7	3	1	6	7	4	2	0

3	3	1	2	2	1	1	3	0	3	*Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	13	16	16	*Running sum of Counts*
0	1	2	3	4	5	6	7	8	9	

2	3	6	7	10	11	12	14	16	17	*Next positions*
0	1	2	3	4	5	6	7	8	9	

0	0								4				7			9			Output
0			3			6			9			12			15			18	

Distribution Counting Example (II)

7	0	4	0	9	1	9	1	9	5	3	7	3	1	6	7	4	2	0

3	3	1	2	2	1	1	3	0	3	*Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	13	16	16	*Running sum of Counts*
0	1	2	3	4	5	6	7	8	9	

2	4	6	7	10	11	12	14	16	17	*Next positions*
0	1	2	3	4	5	6	7	8	9	

0	0		1						4				7			9			Output
0			3			6			9			12			15			18	

Distribution Counting Example (II)

7	0	4	0	9	1	9	1	9	5	3	7	3	1	6	7	4	2	0

3	3	1	2	2	1	1	3	0	3	*Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	13	16	16	*Running sum of Counts*
0	1	2	3	4	5	6	7	8	9	

2	4	6	7	10	11	12	14	16	18	*Next positions*
0	1	2	3	4	5	6	7	8	9	

0	0		1						4				7			9	9		Output
0			3			6			9			12			15			18	

Distribution Counting Example (II)

7	0	4	0	9	1	9	1	9	5	3	7	3	1	6	7	4	2	0

3	3	1	2	2	1	1	3	0	3	*Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	13	16	16	*Running sum of Counts*
0	1	2	3	4	5	6	7	8	9	

2	5	6	7	10	11	12	14	16	18	*Next positions*
0	1	2	3	4	5	6	7	8	9	

0	0		1	1					4				7			9	9		Output
0			3			6			9			12			15			18	

Distribution Counting Example (II)

7	0	4	0	9	1	9	1	9	5	3	7	3	1	6	7	4	2	0

3	3	1	2	2	1	1	3	0	3	*Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	13	16	16	*Running sum of Counts*
0	1	2	3	4	5	6	7	8	9	

2	5	6	7	10	11	12	14	16	19	*Next positions*
0	1	2	3	4	5	6	7	8	9	

0	0		1	1					4				7			9	9	9	Output
0			3			6			9			12			15			18	

Distribution Counting Example (II)

7	0	4	0	9	1	9	1	9	5	3	7	3	1	6	7	4	2	0

3	3	1	2	2	1	1	3	0	3	*Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	13	16	16	*Running sum of Counts*
0	1	2	3	4	5	6	7	8	9	

2	5	6	7	10	12	12	14	16	19	*Next positions*
0	1	2	3	4	5	6	7	8	9	

0	0		1	1					4		5		7			9	9	9	Output
0			3			6			9			12			15			18	

Distribution Counting Example (II)

7	0	4	0	9	1	9	1	9	5	3	7	3	1	6	7	4	2	0

3	3	1	2	2	1	1	3	0	3	*Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	13	16	16	*Running sum of Counts*
0	1	2	3	4	5	6	7	8	9	

2	5	6	8	10	12	12	14	16	19	*Next positions*
0	1	2	3	4	5	6	7	8	9	

0	0		1	1			3		4		5		7			9	9	9	Output
0			3			6			9			12			15			18	

Distribution Counting Example (II)

7	0	4	0	9	1	9	1	9	5	3	7	3	1	6	7	4	2	0

3	3	1	2	2	1	1	3	0	3	*Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	13	16	16	*Running sum of Counts*
0	1	2	3	4	5	6	7	8	9	

2	5	6	8	10	12	12	15	16	19	*Next positions*
0	1	2	3	4	5	6	7	8	9	

0	0		1	1			3		4		5		7	7		9	9	9	Output
0			3			6			9			12			15			18	

Distribution Counting Example (II)

7	0	4	0	9	1	9	1	9	5	3	7	3	1	6	7	4	2	0

3	3	1	2	2	1	1	3	0	3	*Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	13	16	16	*Running sum of Counts*
0	1	2	3	4	5	6	7	8	9	

2	5	6	9	10	12	12	15	16	19	*Next positions*
0	1	2	3	4	5	6	7	8	9	

0	0		1	1			3	3	4		5		7	7		9	9	9	Output
0			3			6			9			12			15			18	

Distribution Counting Example (II)

7	0	4	0	9	1	9	1	9	5	3	7	3	1	6	7	4	2	0

3	3	1	2	2	1	1	3	0	3	*Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	13	16	16	*Running sum of Counts*
0	1	2	3	4	5	6	7	8	9	

2	6	6	9	10	12	12	15	16	19	*Next positions*
0	1	2	3	4	5	6	7	8	9	

0	0		1	1	1		3	3	4		5		7	7		9	9	9	Output
0			3			6			9			12			15			18	

Distribution Counting Example (II)

7	0	4	0	9	1	9	1	9	5	3	7	3	1	6	7	4	2	0

3	3	1	2	2	1	1	3	0	3	*Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	13	16	16	*Running sum of Counts*
0	1	2	3	4	5	6	7	8	9	

2	6	6	9	10	12	13	15	16	19	*Next positions*
0	1	2	3	4	5	6	7	8	9	

0	0		1	1	1		3	3	4		5	6	7	7		9	9	9	Output
0			3			6			9			12			15			18	

Distribution Counting Example (II)

7	0	4	0	9	1	9	1	9	5	3	7	3	1	6	7	4	2	0

3	3	1	2	2	1	1	3	0	3	*Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	13	16	16	*Running sum of Counts*
0	1	2	3	4	5	6	7	8	9	

2	6	6	9	10	12	13	16	16	19	*Next positions*
0	1	2	3	4	5	6	7	8	9	

0	0		1	1	1		3	3	4		5	6	7	7	7	9	9	9	Output
0			3			6			9			12			15			18	

Distribution Counting Example (II)

7	0	4	0	9	1	9	1	9	5	3	7	3	1	6	7	4	2	0

3	3	1	2	2	1	1	3	0	3	*Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	13	16	16	*Running sum of Counts*
0	1	2	3	4	5	6	7	8	9	

2	6	6	9	11	12	13	16	16	19	*Next positions*
0	1	2	3	4	5	6	7	8	9	

0	0		1	1	1		3	3	4	4	5	6	7	7	7	9	9	9	Output
0			3			6			9			12			15			18	

Distribution Counting Example (II)

7	0	4	0	9	1	9	1	9	5	3	7	3	1	6	7	4	2	0

3	3	1	2	2	1	1	3	0	3	*Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	13	16	16	*Running sum of Counts*
0	1	2	3	4	5	6	7	8	9	

2	6	7	9	11	12	13	16	16	19	*Next positions*
0	1	2	3	4	5	6	7	8	9	

0	0		1	1	1	2	3	3	4	4	5	6	7	7	7	9	9	9	Output
0			3			6			9			12			15			18	

Distribution Counting Example (II)

7	0	4	0	9	1	9	1	9	5	3	7	3	1	6	7	4	2	0

3	3	1	2	2	1	1	3	0	3	*Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	13	16	16	*Running sum of Counts*
0	1	2	3	4	5	6	7	8	9	

3	6	7	9	11	12	13	16	16	19	*Next positions*
0	1	2	3	4	5	6	7	8	9	

0	0	0	1	1	1	2	3	3	4	4	5	6	7	7	7	9	9	9	Output
0			3			6			9			12			15			18	

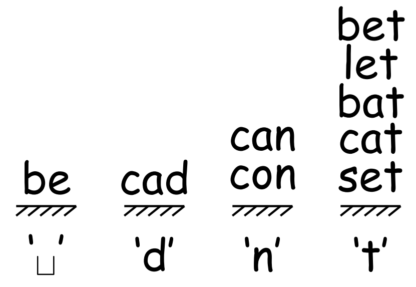
Radix Sort

Idea: Sort keys *one character at a time.*

- Can use distribution counting for each digit.
- Can work either right to left (LSD radix sort) or left to right (MSD radix sort)
- LSD radix sort is venerable: used for punched cards.

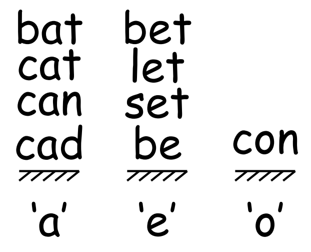
Initial: set, cat, cad, con, bat, can, be, let, bet

Pass 1 (by char #2)



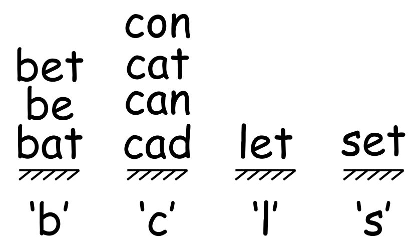
be, cad, con, can, set, cat, bat, let, bet

Pass 2 (by char #1)



cad, can, cat, bat, be, set, let, bet, con

Pass 3 (by char #0)



bat, be, bet, cad, can, cat, con, let, set

MSD Radix Sort

- A bit more complicated: must keep lists from each step separate
- But, can stop processing 1-element lists

A	posn
$\star$ set, cat, cad, con, bat, can, be, let, bet	0
$\star$ bat, be, bet / cat, cad, con, can / let / set	1
bat / $\star$ be, bet / cat, cad, con, can / let / set	2
bat / be / bet / $\star$ cat, cad, con, can / let / set	1
bat / be / bet / $\star$ cat, cad, can / con / let / set	2
bat / be / bet / cad / can / cat / con / let / set	

Performance of Radix Sort

- Radix sort takes $\Theta(B)$ time where B is *total size of the key data*.
- Have measured other sorts as function of #records.
- How to compare?
- To have N different records, must have keys at least $\Theta(\lg N)$ long [why?]
- Furthermore, comparison actually takes time $\Theta(K)$ where K is size of key in worst case [why?]
- So $N \lg N$ comparisons really means $N(\lg N)^2$ operations.
- While radix sort would take $B = N \lg N$ time with minimal-length keys.
- On the other hand, must work to get good constant factors with radix sort.

And Don't Forget Search Trees

Idea: A search tree is in sorted order, when read in inorder.

- Need *balance* to really use for sorting [next topic].
- Given balance, same performance as heapsort: N insertions in time $\lg N$ each, plus $\Theta(N)$ to traverse, gives

$$\Theta(N + N \lg N) = \Theta(N \lg N)$$

Summary

- Insertion sort: $\Theta(Nk)$ comparisons and moves, where k is maximum amount data is displaced from final position.
  - Good for small datasets or almost ordered data sets.
- Quicksort: $\Theta(N \lg N)$ with good constant factor if data is not pathological. Worst case $O(N^2)$.
- Merge sort: $\Theta(N \lg N)$ guaranteed. Good for external sorting.
- Heapsort, treesort with guaranteed balance: $\Theta(N \lg N)$ guaranteed.
- Radix sort, distribution sort: $\Theta(B)$ (number of bytes). Also good for external sorting.

CS61B Lecture #29

Today:

- Balanced search structures (*DS(IJ)*, Chapter 9

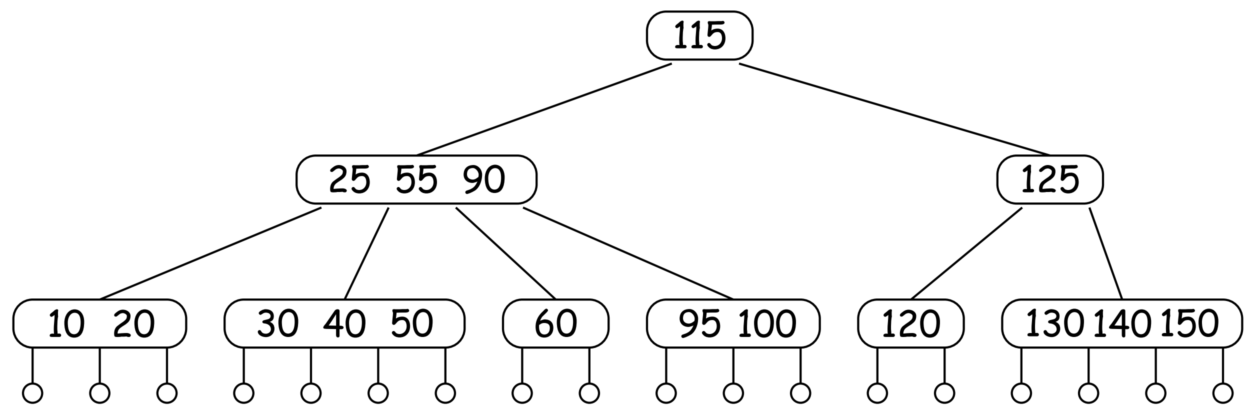
Coming Up:

- Pseudo-random Numbers (*DS(IJ)*, Chapter 11)

Balanced Search: The Problem

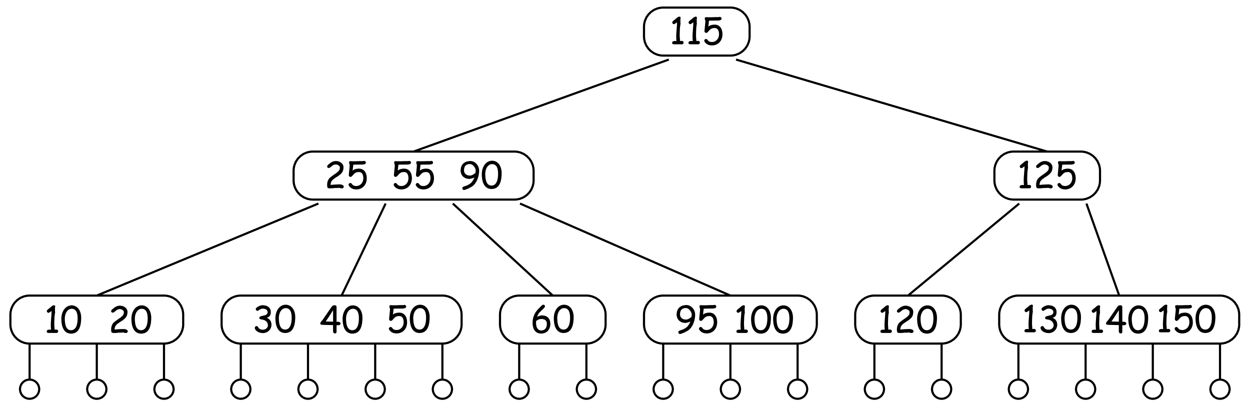
- Why are search trees important?
  - Insertion/deletion fast (on every operation, unlike hash table, which has to expand from time to time).
  - Support range queries, sorting (unlike hash tables)
- But $O(\lg N)$ performance from binary search tree requires remaining keys be divided $\approx$ by some some constant > 1 at each node.
- In other words, that tree be "bushy"
- "Stringy" trees (most inner nodes with one child) perform like linked lists.
- Suffices that heights of any two subtrees of a node always differ by no more than constant factor K.

Example of Direct Approach: B-Trees



- *Order M B-tree* is an M-ary search tree, $M > 2$.
- Obeys search-tree property:
  - Keys are sorted in each node.
  - All keys in subtrees to left of a key, K, are $< K$, and all to right are $> K$.
- Children at bottom of tree are all empty (don't really exist) and equidistant from root.
- Searching is simple generalization of binary search.

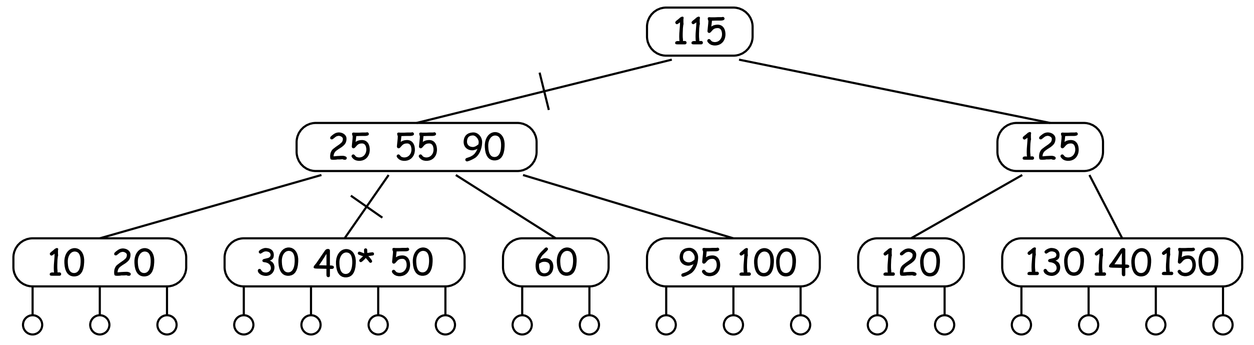
Example of Direct Approach: B-Trees



Idea: If tree grows/shrinks only at root, then two sides always have same height.

- Each node, except root, has from $\lceil M/2 \rceil$ to M children, and one key "between" each two children.
- Root has from 2 to M children (in non-empty tree).
- Insertion: add just above bottom; split overfull nodes as needed, moving one key up to parent.

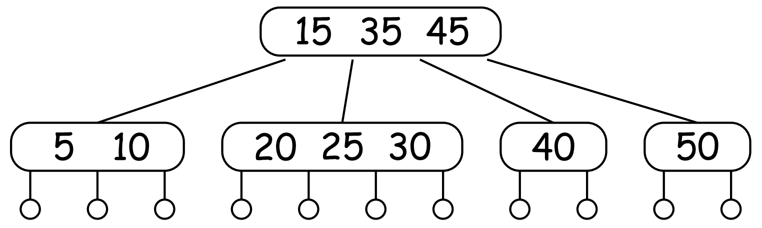
Sample Order 4 B-tree ((2,4) Tree)



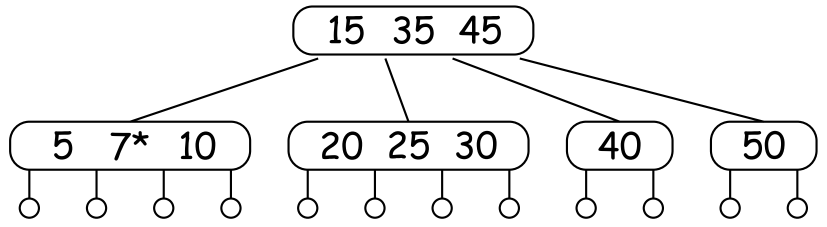
- Crossed lines show path when finding 40.
- Keys on either side of each child pointer in path bracket 40.
- Each node has at least 2 children, and all leaves (little circles) are at the bottom, so height must be $O(\lg N)$.
- In real-life B-tree, order typically much bigger
  - comparable to size of disk sector, page, or other convenient unit of I/O

Inserting in B-tree (Simple Case)

- Start:

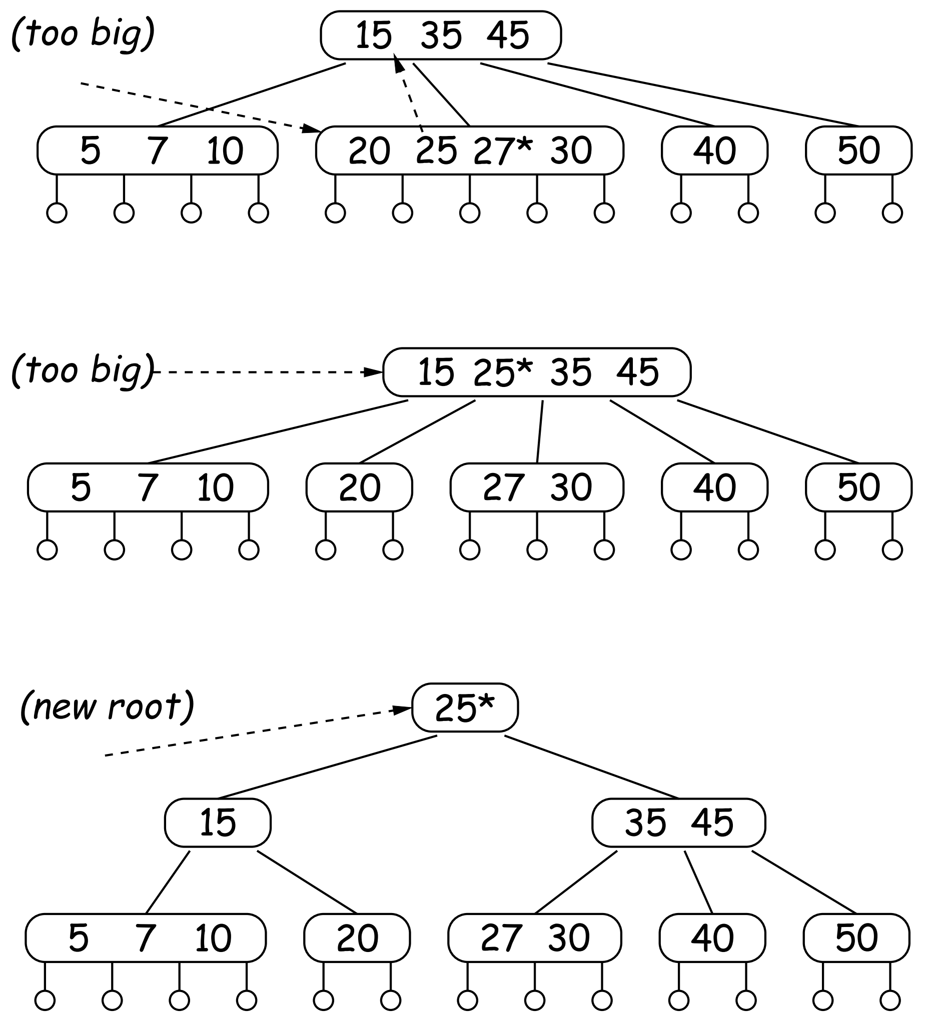


- Insert 7:



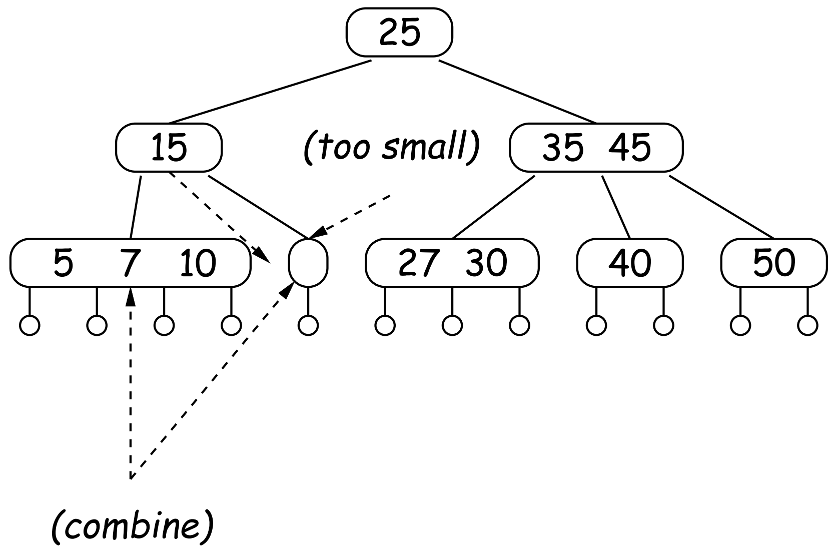
Inserting in B-Tree (Splitting)

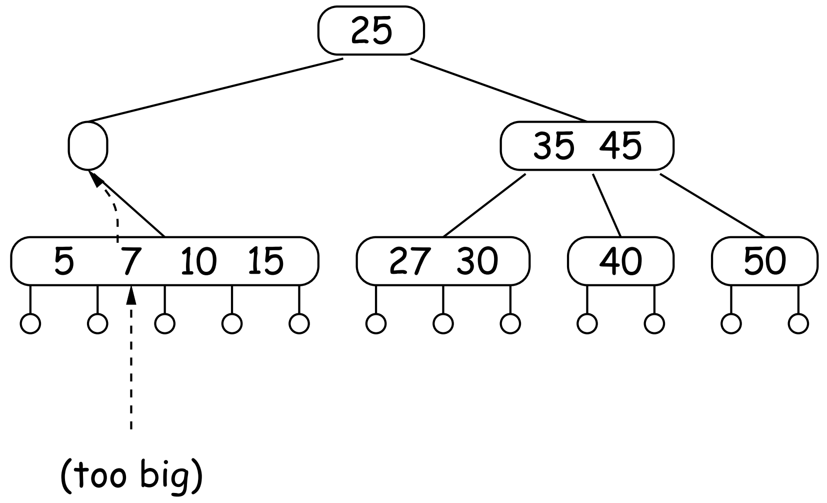
- Insert 27:

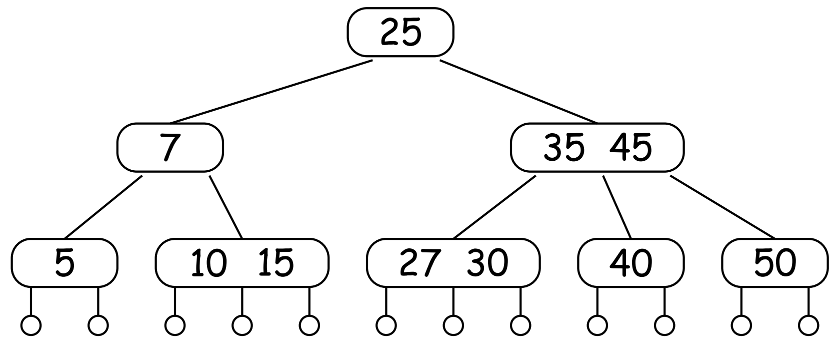


Deleting Keys from B-tree

- Remove 20 from last tree.



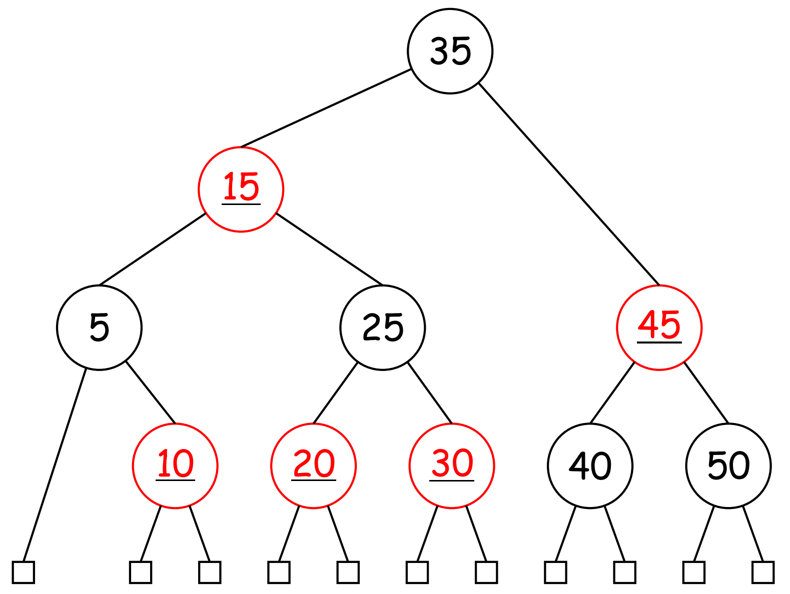




Red-Black Trees

- Red-black tree is a binary search tree with additional constraints that limit how unbalanced it can be.
- Thus, searching is always $O(\lg N)$.
- Used for Java's `TreeSet` and `TreeMap` types.
- When items are inserted or deleted, tree is *rotated* and *recolored* as needed to restore balance.

Red-Black Tree Constraints

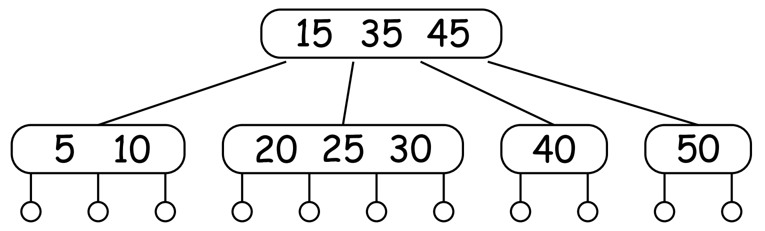


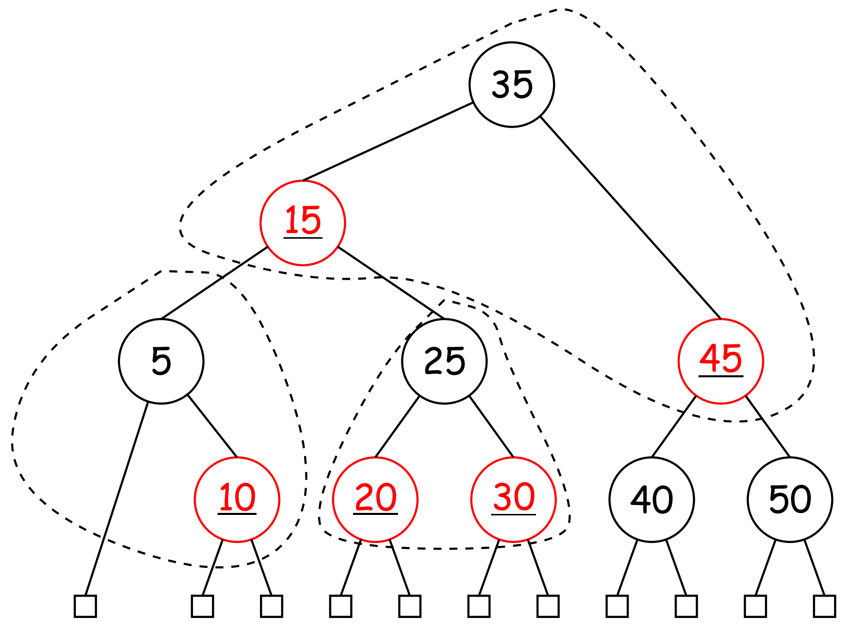
1. Each node is (conceptually) colored red or black.
2. Root is black.
3. Every leaf node contains no data (as for B-trees) and is black.
4. Every leaf has same number of black ancestors.
5. Every internal node has two children.
6. Every red node has two black children.

- Conditions 4, 5, and 6 guarantee $O(\lg N)$ searches.

Red-Black Trees and (2,4) Trees

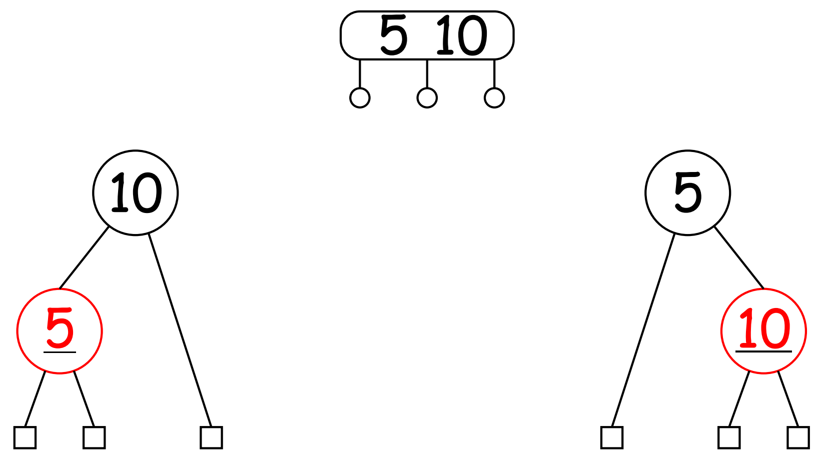
- Every red-black tree corresponds to a (2,4) tree, and the operations on one correspond to those on the other.
- Each node of (2,4) tree corresponds to a cluster of 1–3 red-black nodes in which the top node is black and any others are red.





Additional Constraints: Left-Leaning Red-Black Trees

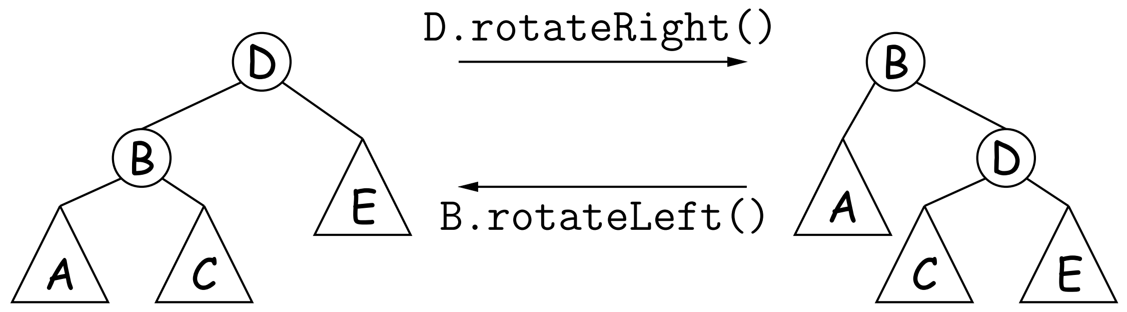
- A node in a (2,4) or (2,3) tree with three children may be represented in two different ways in a red-black tree:



- We can considerably simplify insertion and deletion in a red-black tree by always choosing the option on the left.
- With this constraint, there is a one-to-one relationship between (2,4) trees and red-black trees.
- The resulting trees are called *left-leaning red-black trees.*
- As a further simplification, let's restrict ourselves to red-black trees that correspond to (2,3) trees (whose nodes have no more than 3 children), so that no red-black node has two red children.

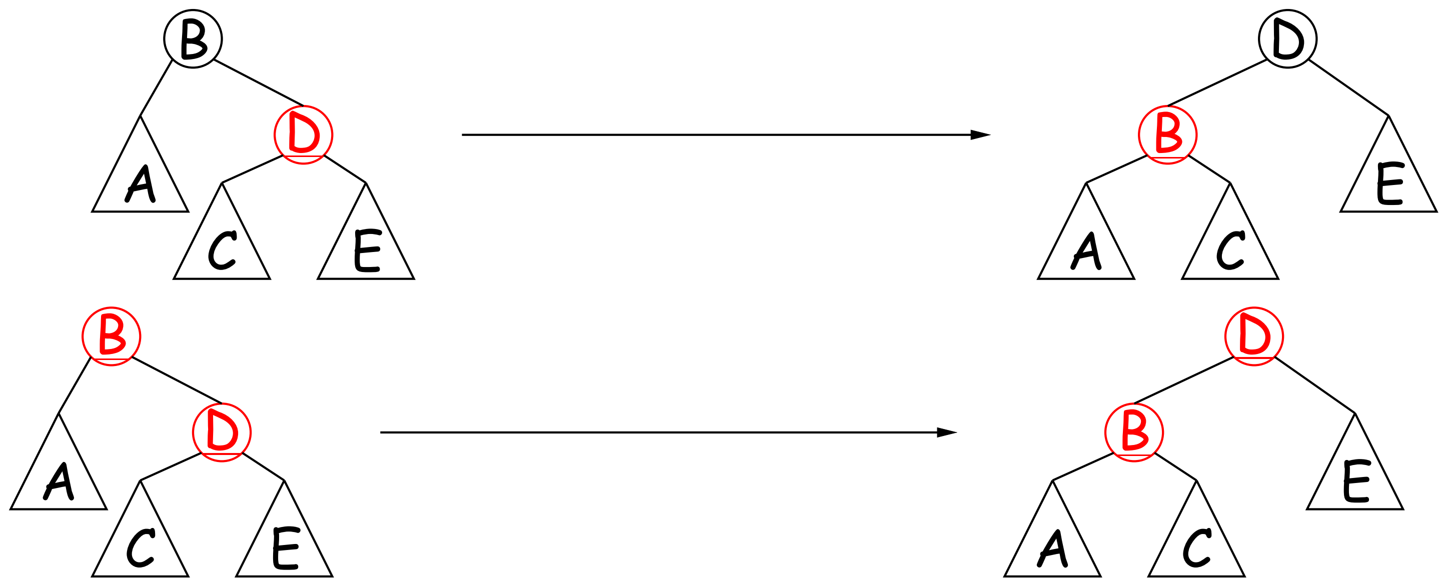
Red-Black Insertion and Rotations

- Insert at bottom just as for binary tree (color red except when tree initially empty).
- Then rotate (and recolor) to restore red-black property, and thus balance.
- Rotation of trees *preserves* binary tree property, but changes balance.



Rotations and Recolorings

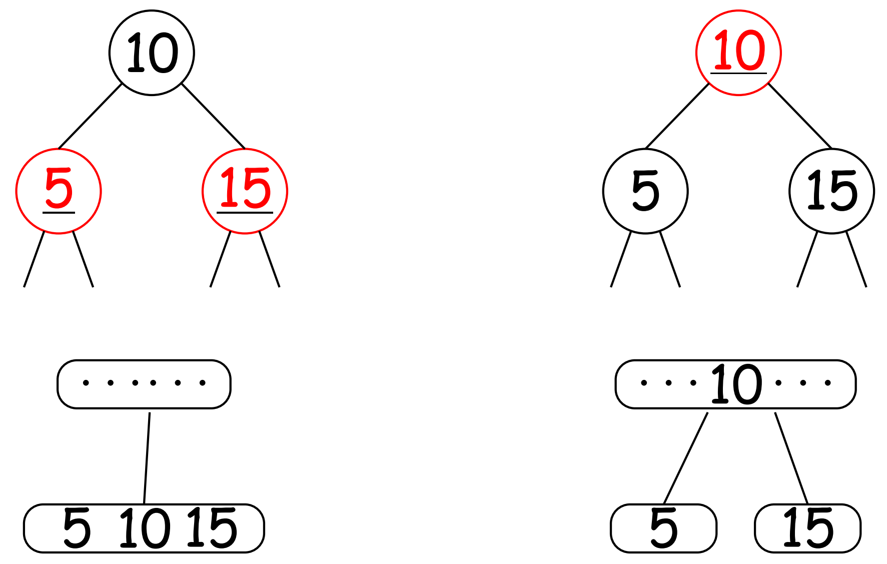
- For our purposes, we'll augment the general rotation algorithms with some recoloring.
- Transfer the color from the original root to the new root, and color the original root red. Examples:



- Neither of these changes the number of black nodes along any path between the root and the leaves.

Splitting by Recoloring

- Our algorithms will temporarily create nodes with too many children, and then split them up.
- A simple recoloring allows us to split nodes. We'll call it `colorFlip`:



- Here, key 10 joins the parent node, splitting the original.

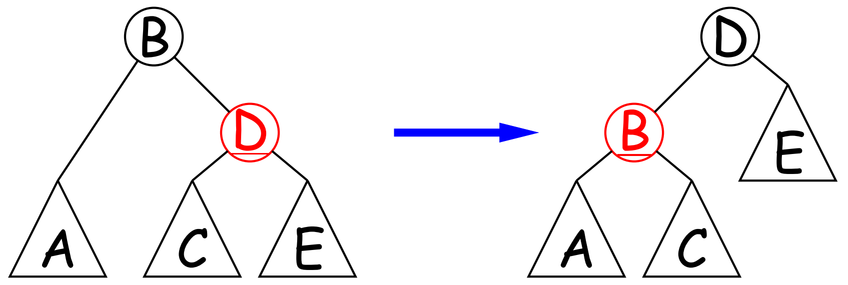
The Algorithm (Sedgewick)

- We posit a binary-tree type `RBTree`: basically ordinary BST nodes plus color.
- Insertion is the same as for ordinary BSTs, but we add some fixups to restore the red-black properties.

```
RBTree insert(RBTree tree, KeyType key) {
    if (tree == null)
        return new RBTree(key, null, null, RED);
    int cmp = key.compareTo(tree.label());
    else if (cmp < 0) tree.setLeft(insert(tree.left(), key));
    else              tree.setRight(insert(tree.right(), key));

    return fixup(tree);     // Only line that's all new!
}
```

Fixing Up the Tree

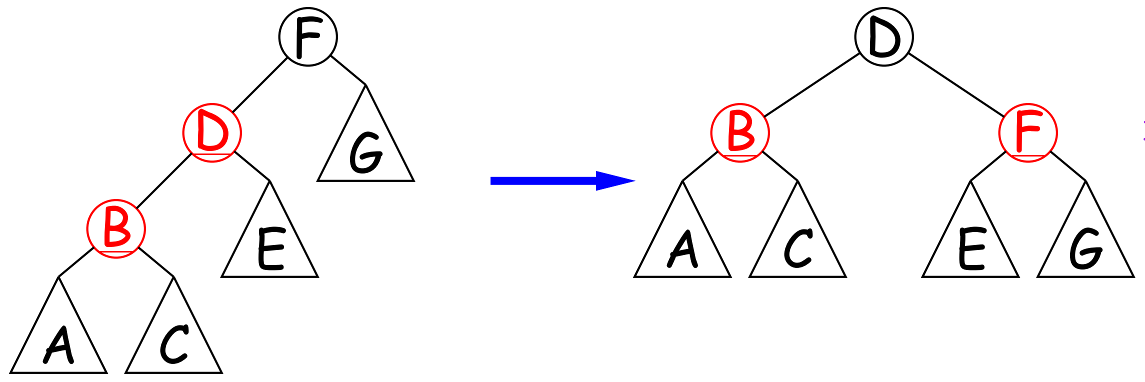
- As we return back up the BST, we restore the left-leaning red-black properties, and limit ourselves to red-black trees that correspond to (2,3) trees by applying the following (in order) to each node:
- Fixup 1: Convert right-leaning trees to left-leaning:



```
if (tree.right().isRed()
    && tree.left().isBlack()) {
       tree.rotateLeft();
}
```

Sometimes, node B will be red, so that both B and D end up red. This is fixed by...

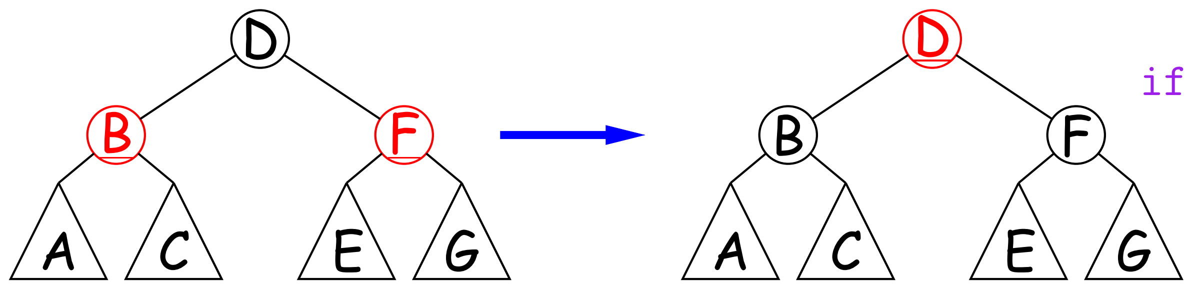
- Fixup 2: Rotate linked red nodes into a normal 4-node (temporarily).



```
if (tree.left().isRed() &&
    tree.left().left().isRed())
        tree.rotateRight();
```

Fixing Up the Tree (II)

- Fixup 3: Break up 4-nodes into 3-nodes or 2-nodes.

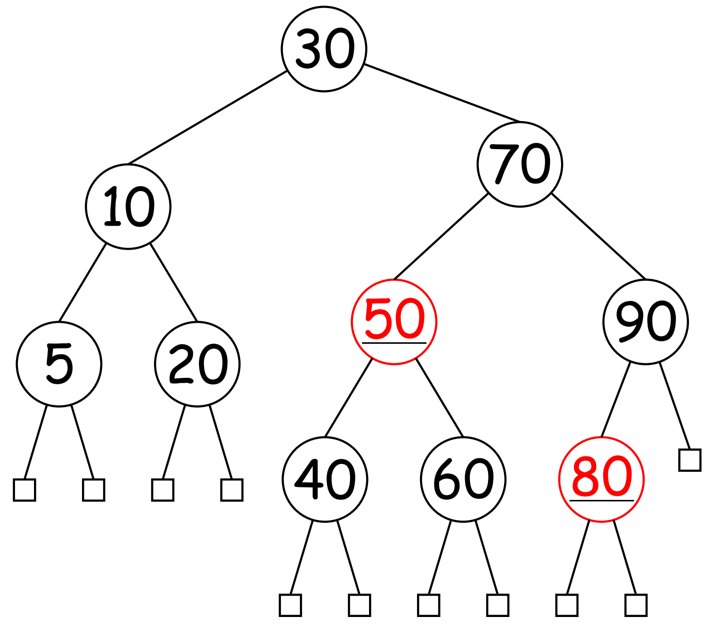


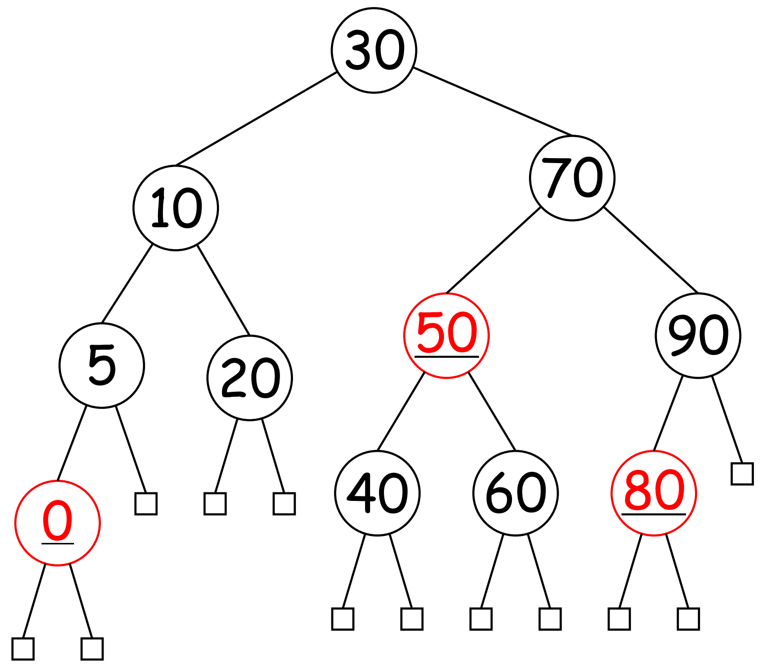
```
if (tree.left().isRed() &&
    tree.right().isRed())
        colorFlip(tree);
```

- Fixup 4: As a result of other fixups, or of insertion into the empty tree, the root may end up red, so color the root black after the rest of insertion and fixups are finished. (Not part of the fixup function; just done at the end).

Example of Left-Leaning 2-3 Red-Black Insertion

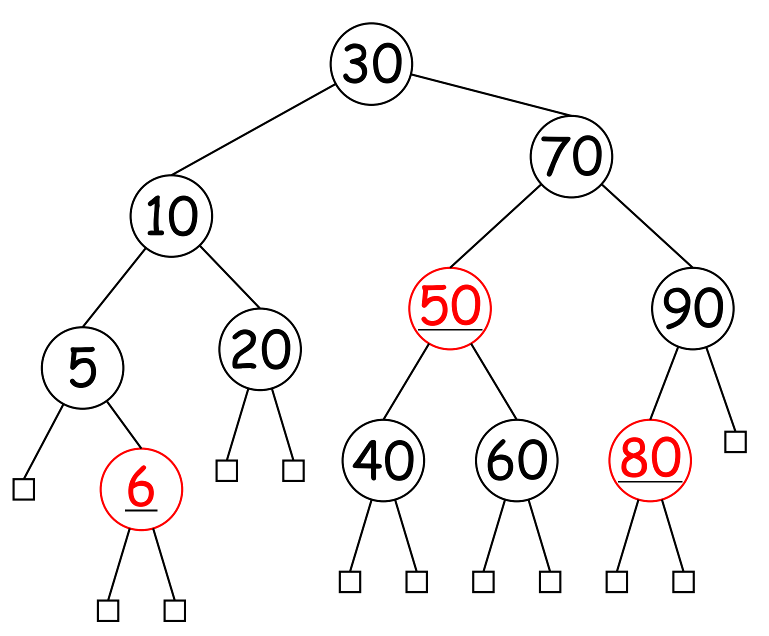
- Insert 0 into initial tree on left. No fixups needed.

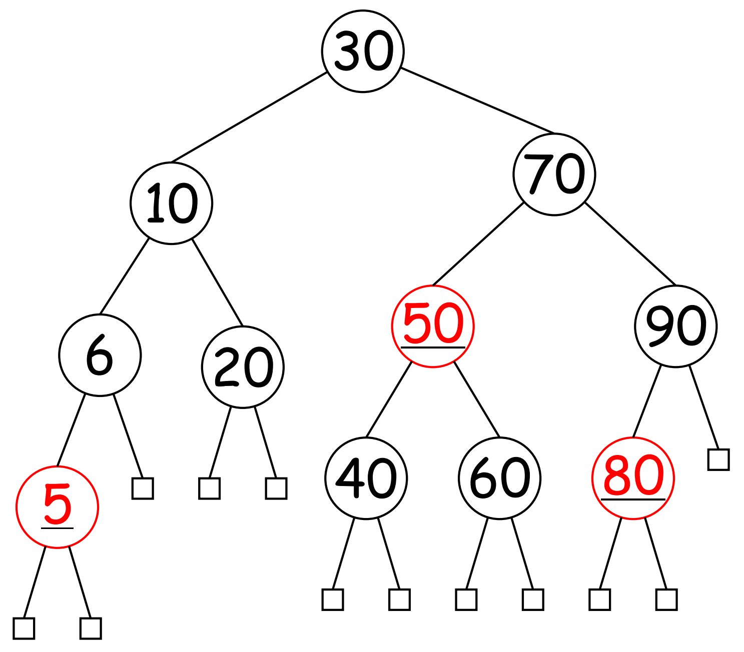




Insertion Example (II)

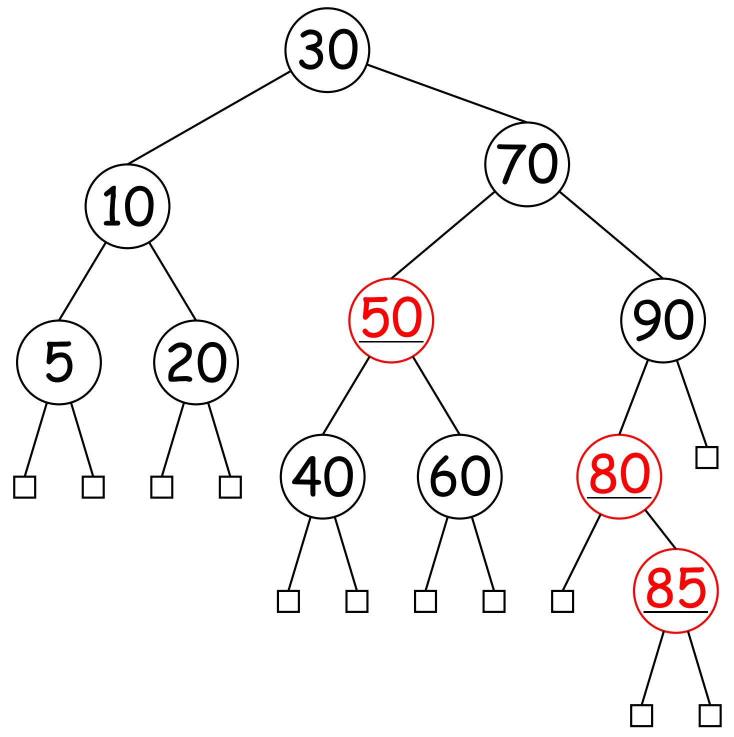
- Instead of 0, let's insert 6, leading to the tree on the left. This is right-leaning, so apply Fixup 1:

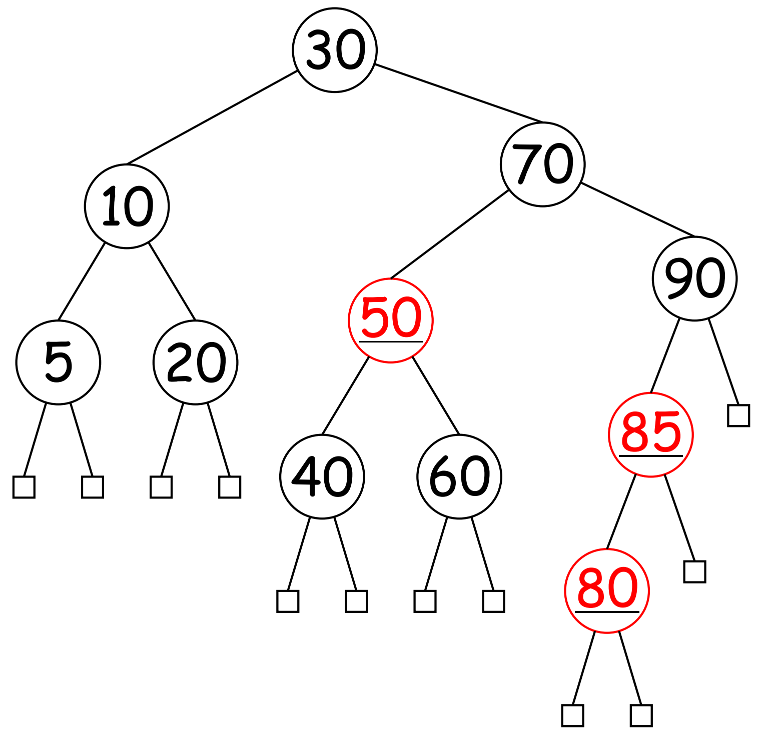




Insertion Example (III)

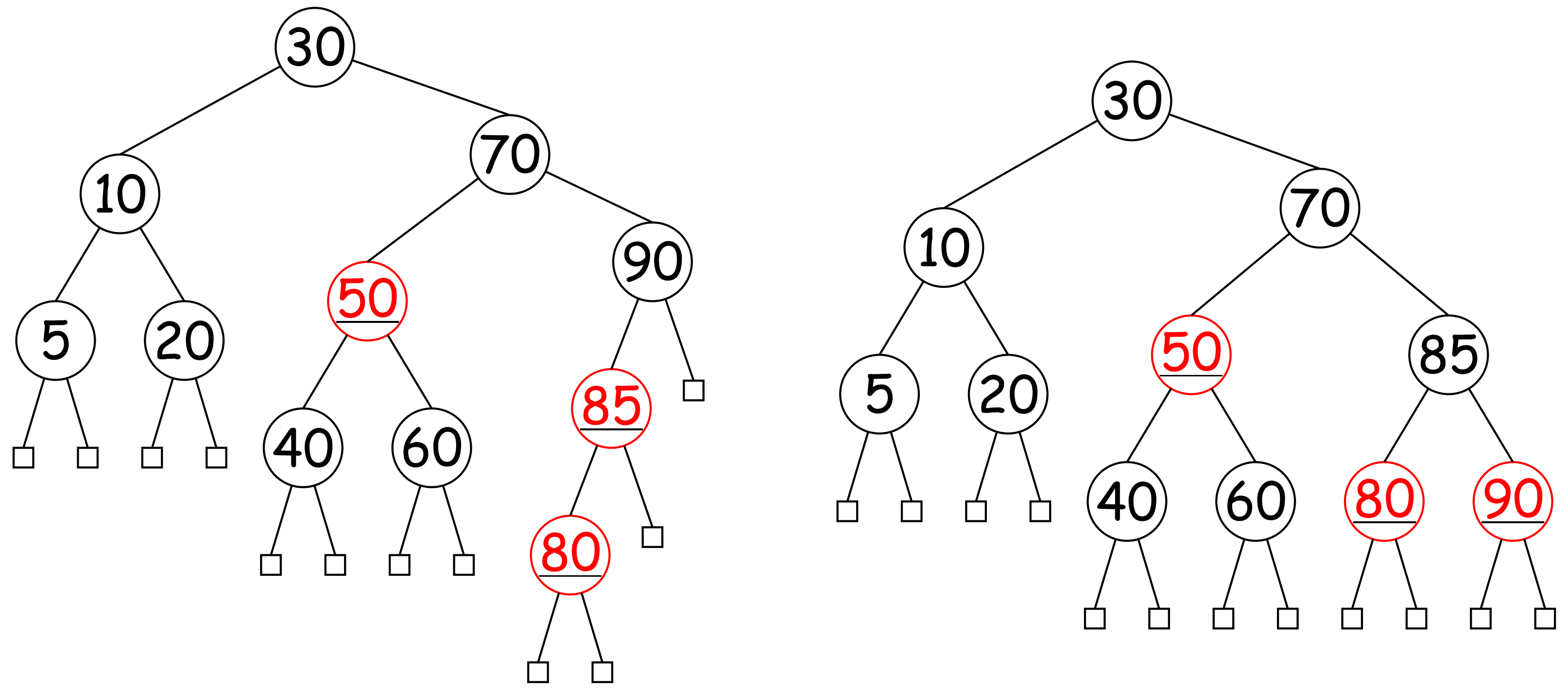
- Now consider inserting 85. We need fixup 1 first.





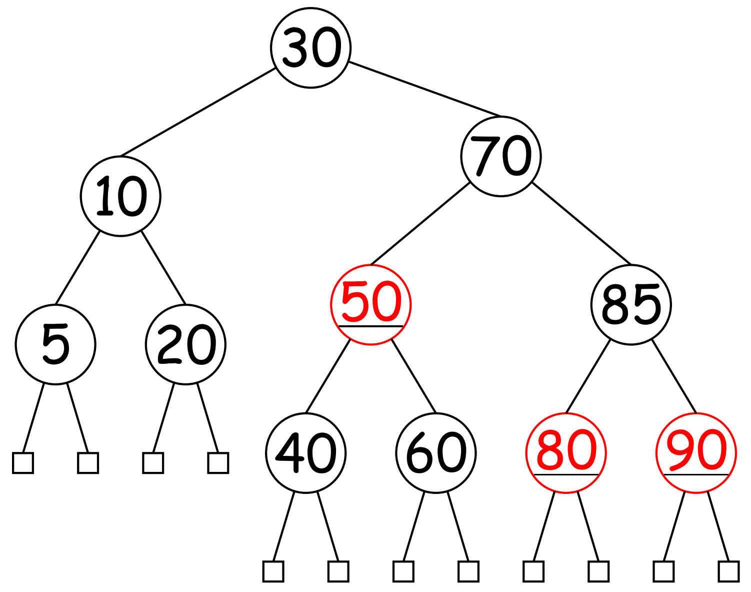
Insertion Example (IIIa)

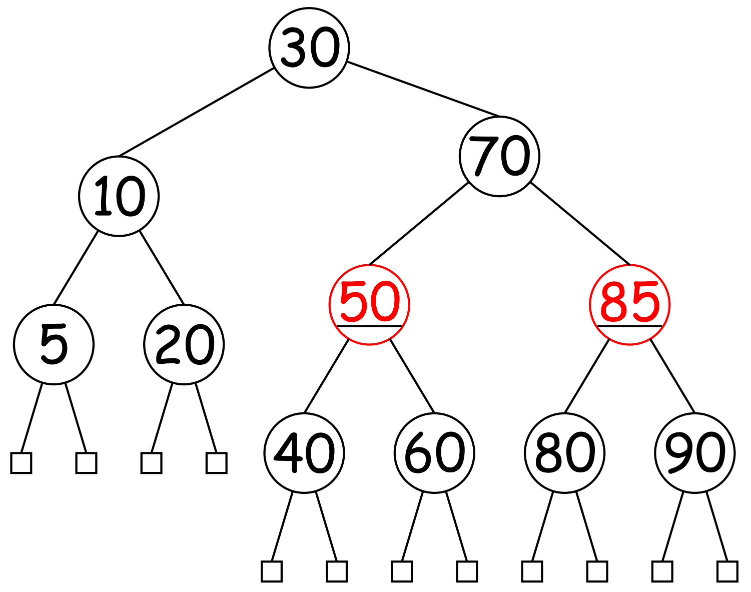
- Now apply fixup 2.



Insertion Example (IIIb)

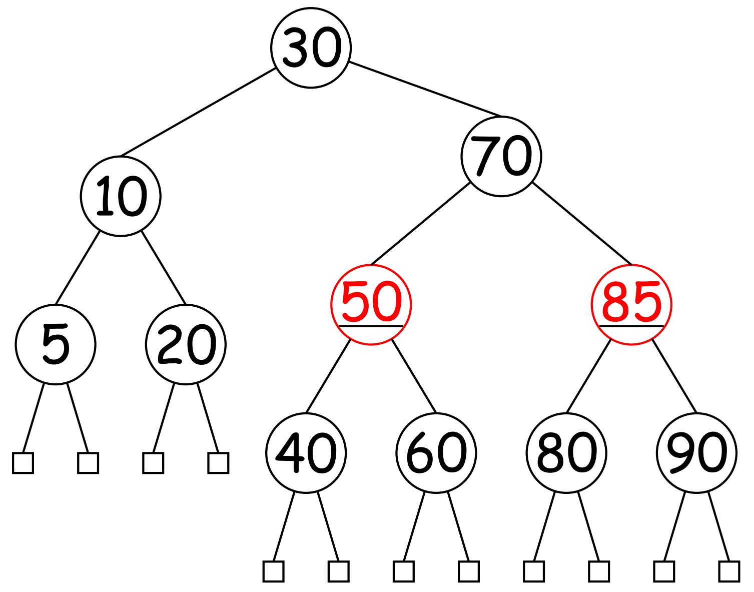
- This gives us a 4-node, so apply fixup 3.

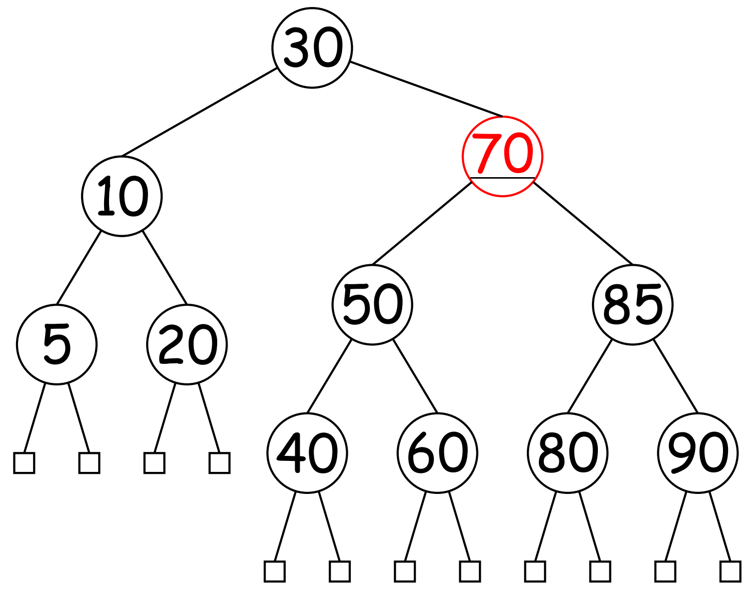




Insertion Example (IIIc)

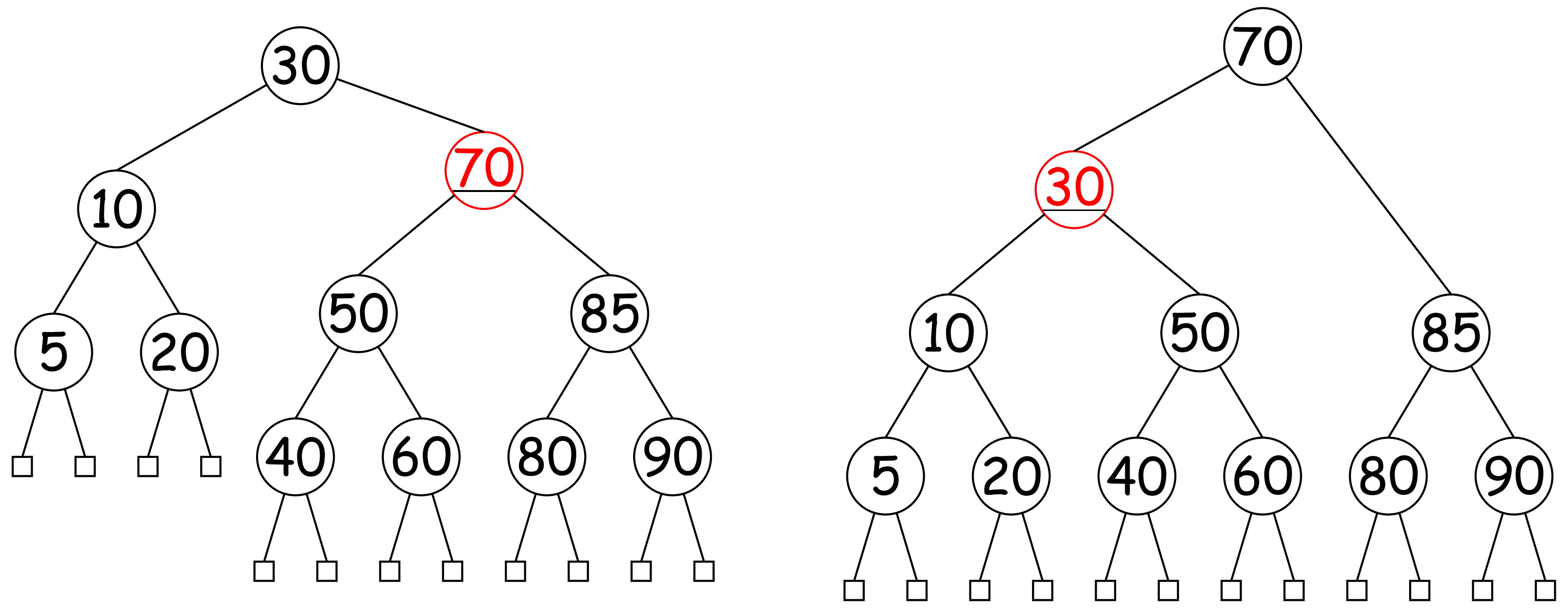
- This gives us another 4-node, so apply fixup 3 again.





Insertion Example (IIId)

- This gives us a right-leaning tree, so apply fixup 1.



CS61B Lecture #31

Today:

- More balanced search structures (*DS(IJ)*, Chapter 9

Coming Up:

- Pseudo-random Numbers (*DS(IJ)*, Chapter 11)

Really Efficient Use of Keys: the Trie

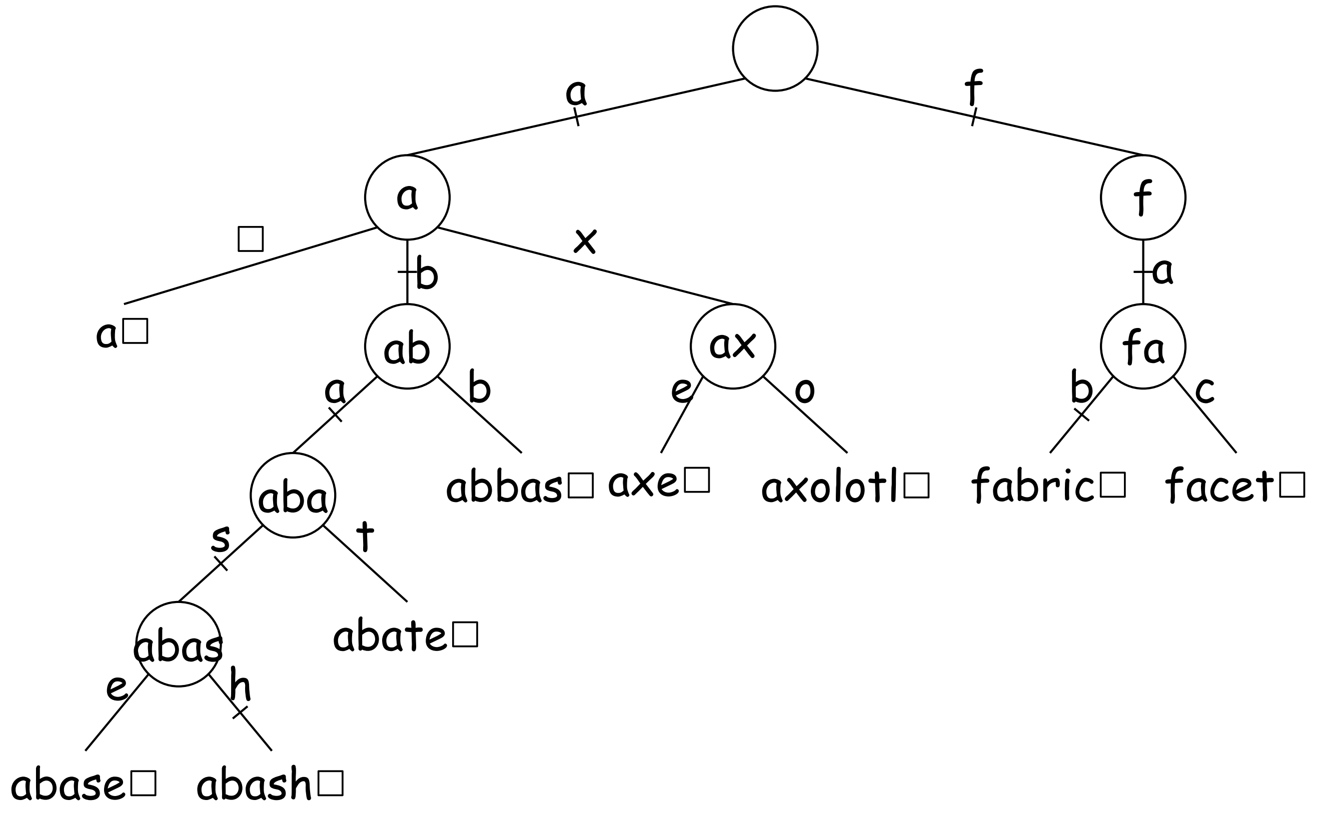
- Haven't said much about cost of comparisons.
- For strings, worst case is length of string.
- Therefore should throw extra factor of key length, L, into costs:
  - $\Theta(M)$ comparisons really means $\Theta(ML)$ operations.
  - So to look for key X, keep looking at same chars of X M times.
- Can we do better? Can we get search cost to be $O(L)$?

Idea: Make a *multi-way decision tree,* with one decision per character of key.

The Trie: Example

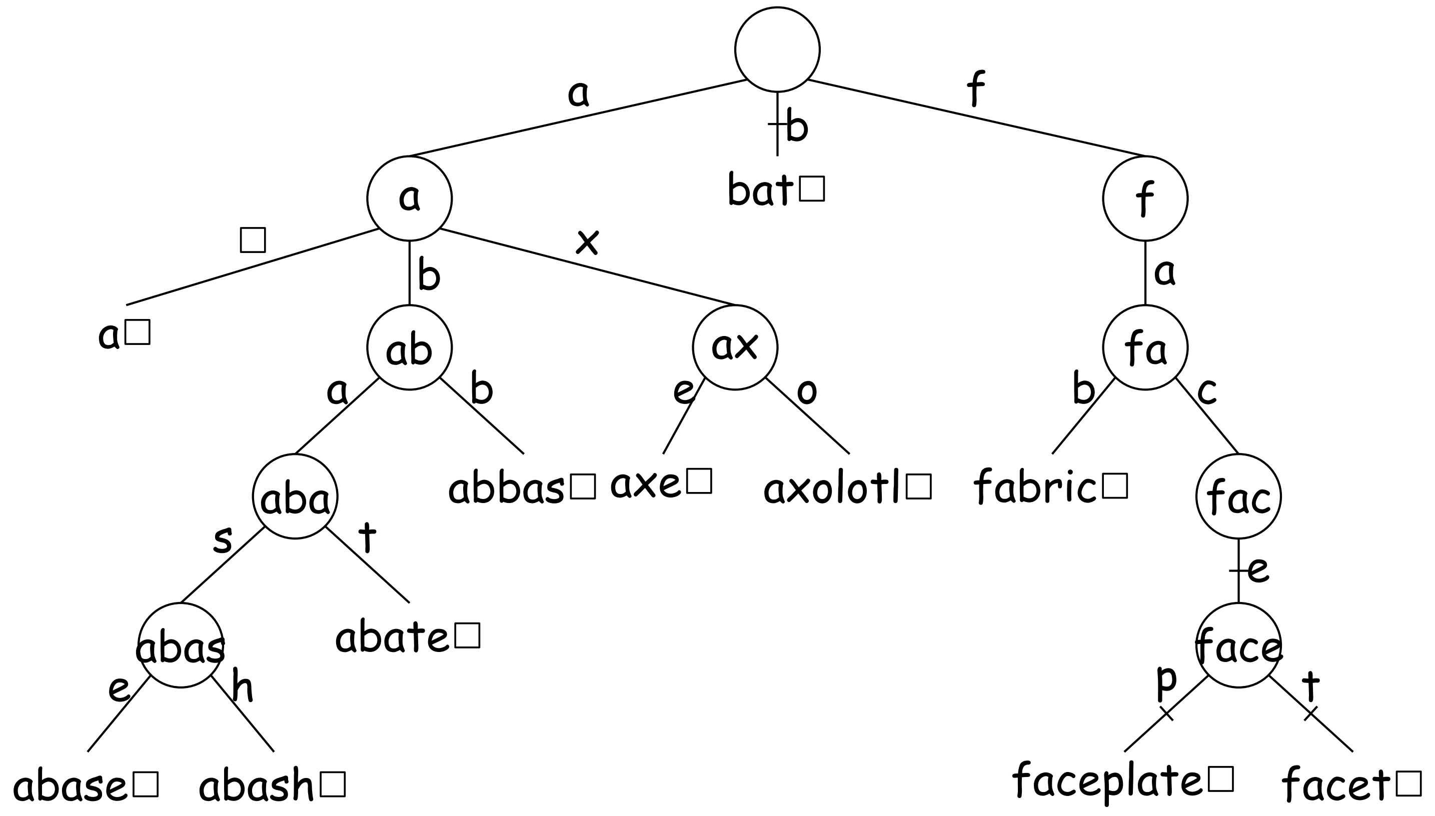
- Set of keys

  {a, abase, abash, abate, abbas, axolotl, axe, fabric, facet}
- Ticked lines show paths followed for "abash" and "fabric"
- Each internal node corresponds to a possible prefix.
- Characters in path to node = that prefix.



Adding Item to a Trie

- Result of adding `bat` and `faceplate`.
- New edges ticked.



A Side-Trip: Scrunching

- For speed, obvious implementation for internal nodes is array indexed by character.
- Gives $O(L)$ performance, L length of search key.
- [Looks as if independent of N, number of keys. Is there a dependence?]
- **Problem:** arrays are *sparsely populated* by non-null values—waste of space.

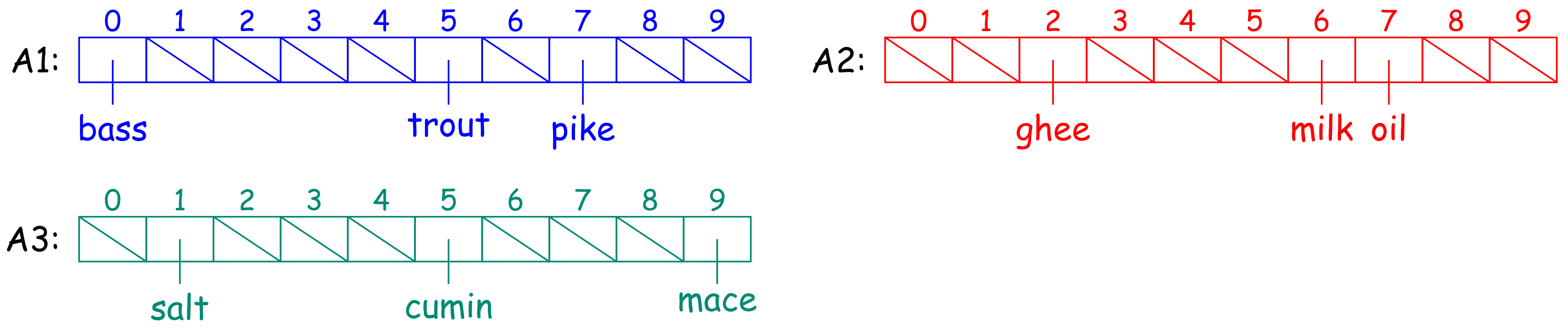
Idea: Put the arrays on top of each other!

- Use null (0, empty) entries of one array to hold non-null elements of another.
- Use extra markers to tell which entries belong to which array.

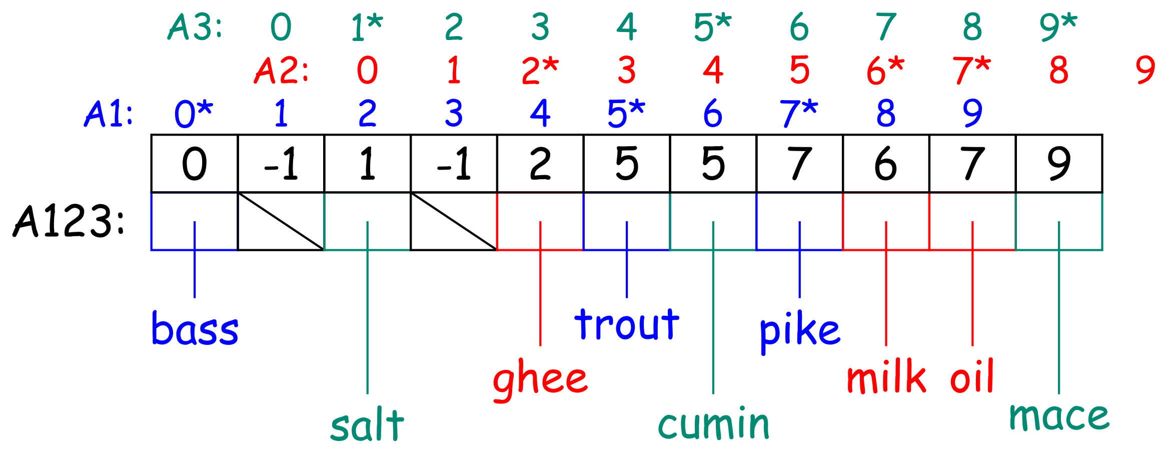
Scrunching Example

Small example: (unrelated to Tries on preceding slides)

- Three leaf arrays, each indexed 0..9



- Now overlay them, but keep track of original index of each item:



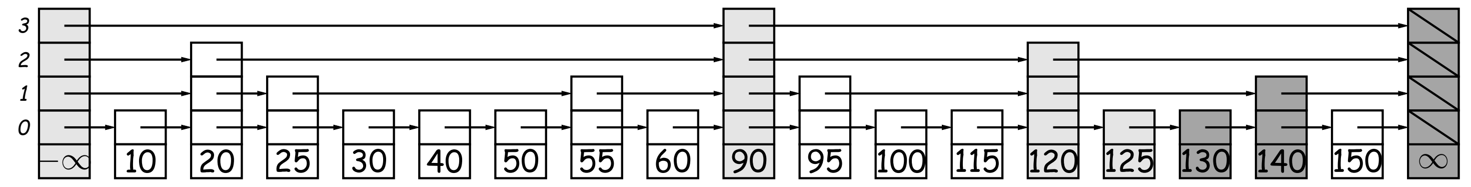
A3: 0 1* 2 3 4 5* 6 7 8 9*
A2: 0 1 2* 3 4 5 6* 7* 8 9
A1: 0* 1 2 3 4 5* 6 7* 8 9
0 -1 1 -1 2 5 5 7 6 7 9
A123:
bass
salt
ghee
trout
cumin
pike
milk
oil
mace

Practicum

- The scrunching idea is cute, but
  - Not so good if we want to expand our trie.
  - A bit complicated.
  - Actually more useful for representing large, sparse, fixed tables with many rows and columns.
- Furthermore, number of children in trie tends to drop drastically when one gets a few levels down from the root.
- So in practice, might as well use linked lists to represent set of node's children...
- ...but use arrays for the first few levels, which are likely to have more children.

Probabilistic Balancing: Skip Lists

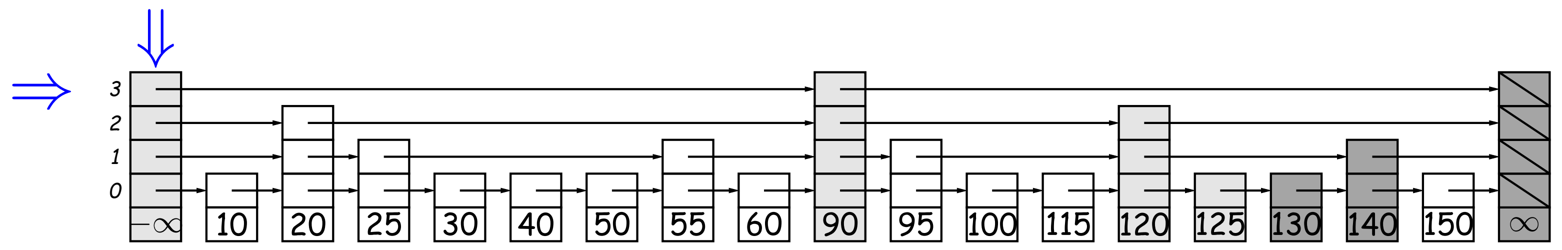
- A *skip list* can be thought of as a kind of n-ary search tree in which we choose to put the keys at "random" heights.
- More often thought of as an ordered list in which one can skip large segments.
- Typical example:



- To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.
- In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.
- Heights of the nodes were chosen randomly so that there are about 1/2 as many nodes that are $> k$ high as there are that are k high.
- Makes searches fast *with high probability.*

Probabilistic Balancing: Skip Lists

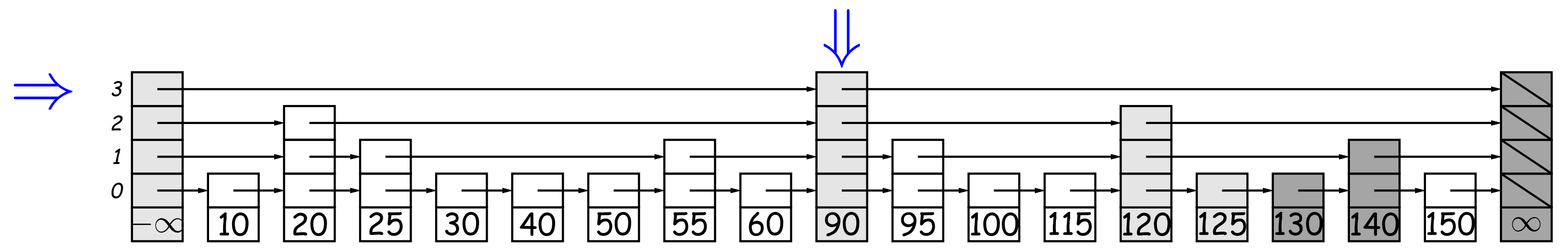
- A *skip list* can be thought of as a kind of n-ary search tree in which we choose to put the keys at "random" heights.
- More often thought of as an ordered list in which one can skip large segments.
- Typical example:



- To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.
- In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.
- Heights of the nodes were chosen randomly so that there are about 1/2 as many nodes that are $> k$ high as there are that are k high.
- Makes searches fast *with high probability.*

Probabilistic Balancing: Skip Lists

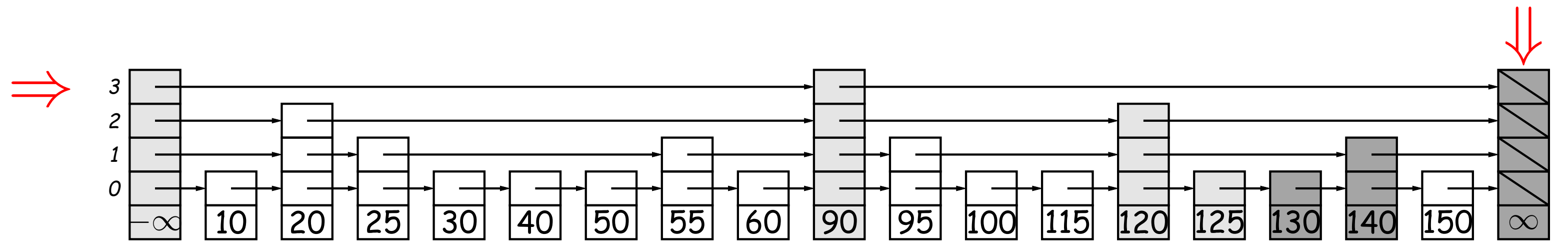
- A *skip list* can be thought of as a kind of n-ary search tree in which we choose to put the keys at "random" heights.
- More often thought of as an ordered list in which one can skip large segments.
- Typical example:



- To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.
- In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.
- Heights of the nodes were chosen randomly so that there are about 1/2 as many nodes that are $> k$ high as there are that are k high.
- Makes searches fast *with high probability.*

Probabilistic Balancing: Skip Lists

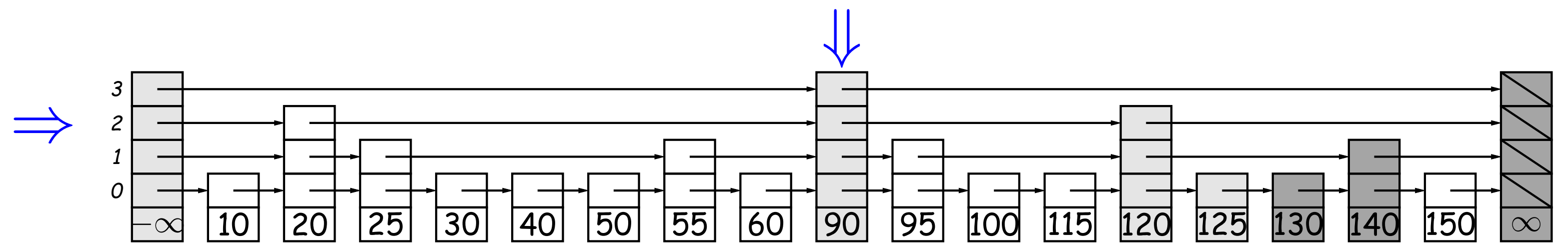
- A *skip list* can be thought of as a kind of n-ary search tree in which we choose to put the keys at “random” heights.
- More often thought of as an ordered list in which one can skip large segments.
- Typical example:



- To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.
- In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.
- Heights of the nodes were chosen randomly so that there are about 1/2 as many nodes that are $> k$ high as there are that are k high.
- Makes searches fast *with high probability.*

Probabilistic Balancing: Skip Lists

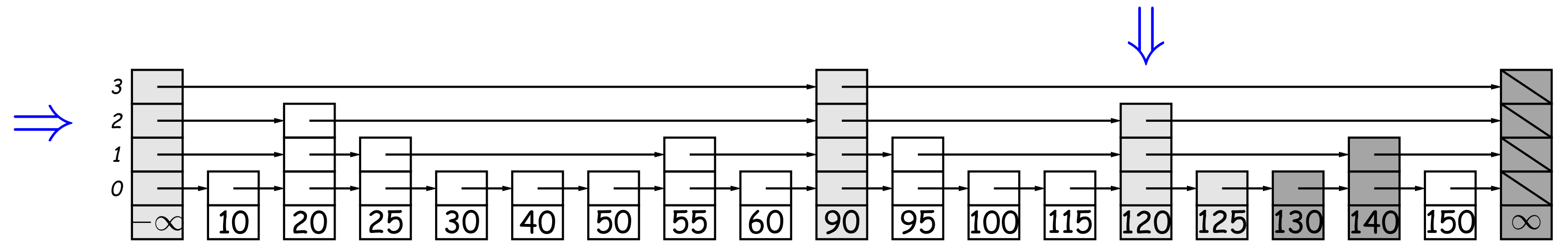
- A *skip list* can be thought of as a kind of n-ary search tree in which we choose to put the keys at "random" heights.
- More often thought of as an ordered list in which one can skip large segments.
- Typical example:



- To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.
- In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.
- Heights of the nodes were chosen randomly so that there are about 1/2 as many nodes that are $> k$ high as there are that are k high.
- Makes searches fast *with high probability.*

Probabilistic Balancing: Skip Lists

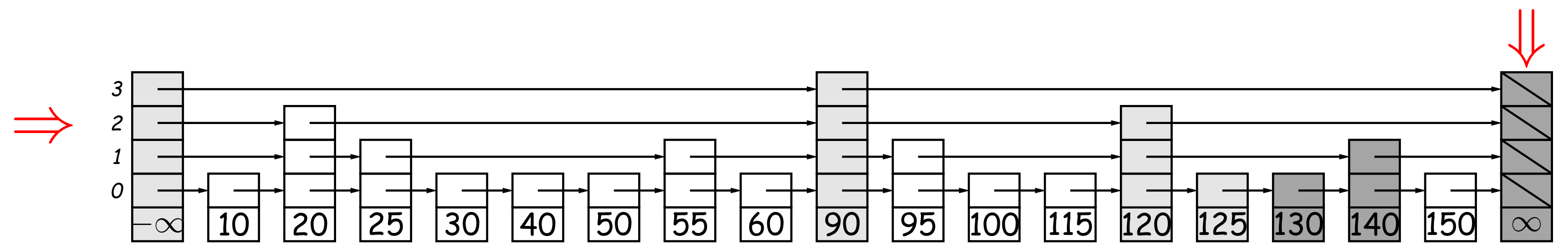
- A *skip list* can be thought of as a kind of n-ary search tree in which we choose to put the keys at “random” heights.
- More often thought of as an ordered list in which one can skip large segments.
- Typical example:



- To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.
- In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.
- Heights of the nodes were chosen randomly so that there are about 1/2 as many nodes that are $> k$ high as there are that are k high.
- Makes searches fast *with high probability.*

Probabilistic Balancing: Skip Lists

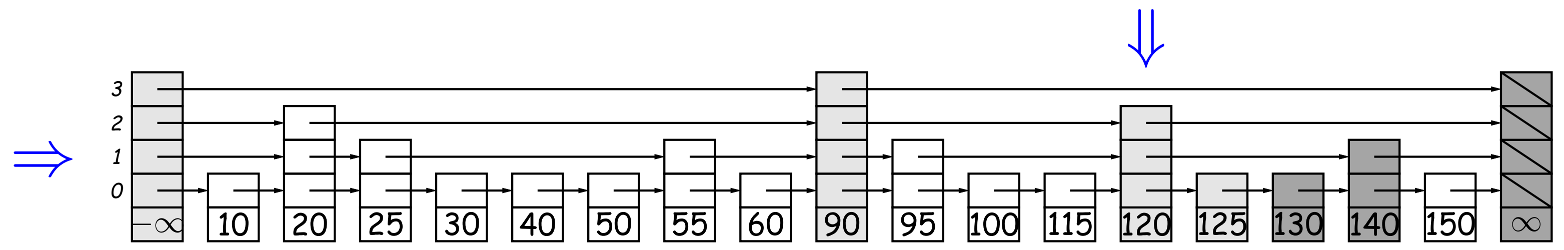
- A *skip list* can be thought of as a kind of n-ary search tree in which we choose to put the keys at "random" heights.
- More often thought of as an ordered list in which one can skip large segments.
- Typical example:



- To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.
- In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.
- Heights of the nodes were chosen randomly so that there are about 1/2 as many nodes that are $> k$ high as there are that are k high.
- Makes searches fast *with high probability.*

Probabilistic Balancing: Skip Lists

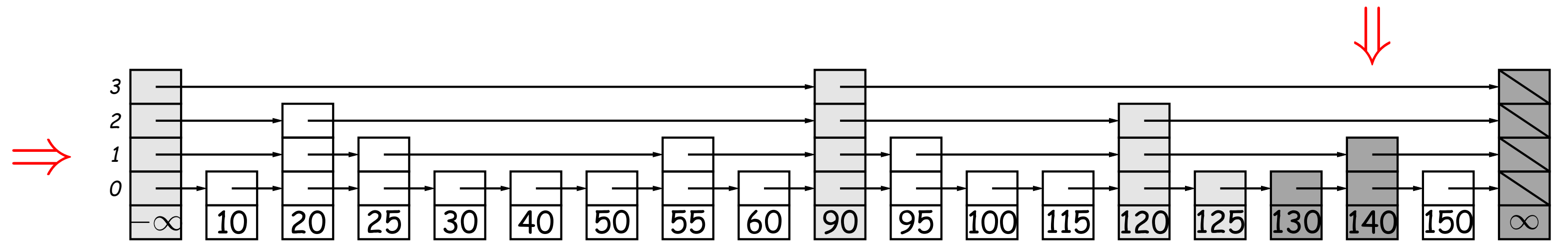
- A *skip list* can be thought of as a kind of n-ary search tree in which we choose to put the keys at "random" heights.
- More often thought of as an ordered list in which one can skip large segments.
- Typical example:



- To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.
- In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.
- Heights of the nodes were chosen randomly so that there are about 1/2 as many nodes that are $> k$ high as there are that are k high.
- Makes searches fast *with high probability.*

Probabilistic Balancing: Skip Lists

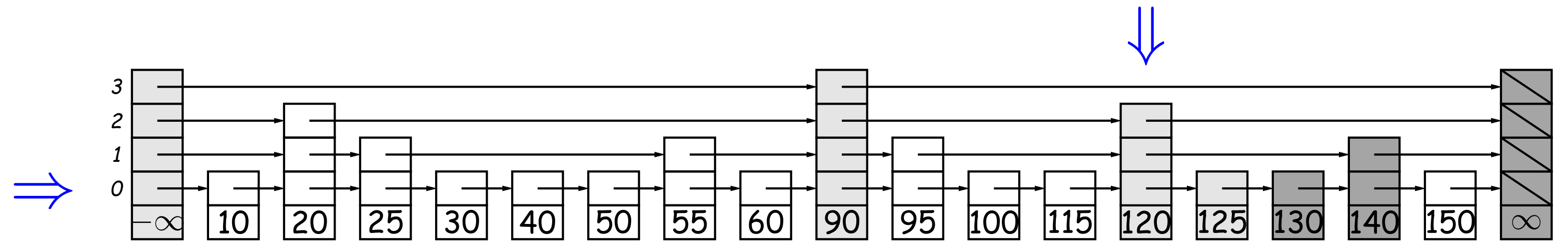
- A *skip list* can be thought of as a kind of n-ary search tree in which we choose to put the keys at "random" heights.
- More often thought of as an ordered list in which one can skip large segments.
- Typical example:



- To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.
- In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.
- Heights of the nodes were chosen randomly so that there are about 1/2 as many nodes that are $> k$ high as there are that are k high.
- Makes searches fast *with high probability.*

Probabilistic Balancing: Skip Lists

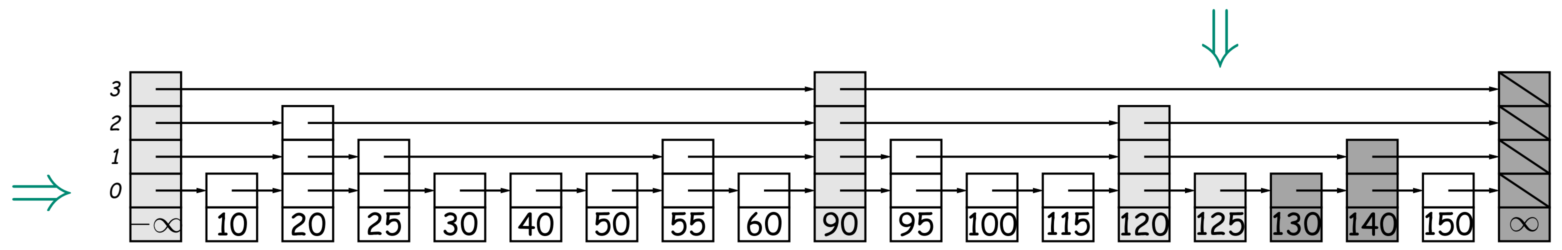
- A *skip list* can be thought of as a kind of n-ary search tree in which we choose to put the keys at "random" heights.
- More often thought of as an ordered list in which one can skip large segments.
- Typical example:



- To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.
- In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.
- Heights of the nodes were chosen randomly so that there are about 1/2 as many nodes that are $> k$ high as there are that are k high.
- Makes searches fast *with high probability.*

Probabilistic Balancing: Skip Lists

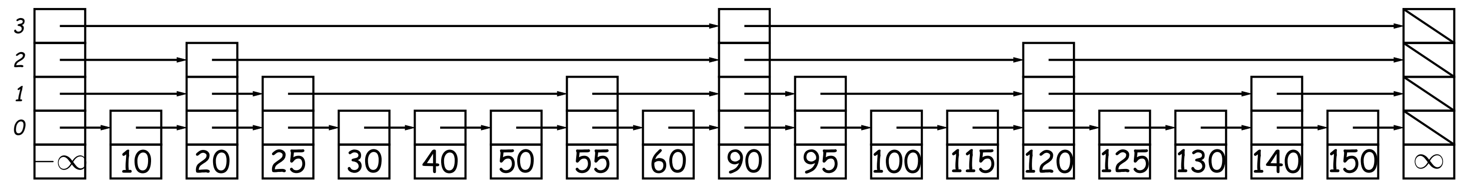
- A *skip list* can be thought of as a kind of n-ary search tree in which we choose to put the keys at "random" heights.
- More often thought of as an ordered list in which one can skip large segments.
- Typical example:



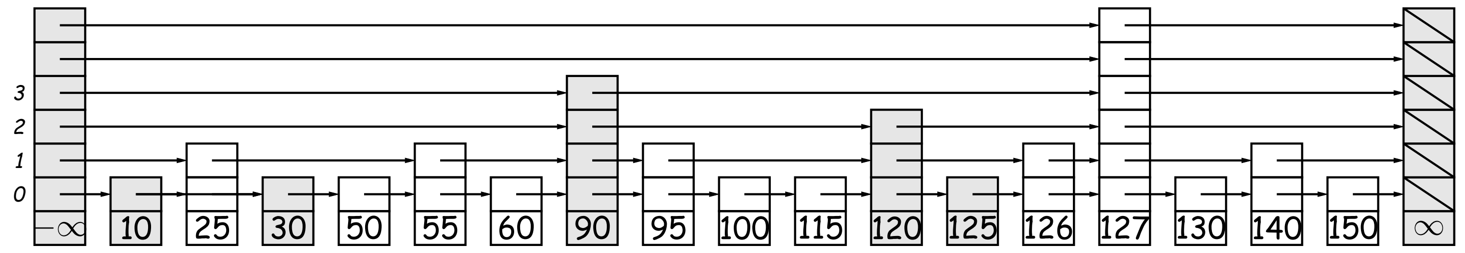
- To search, start at top layer on left, search until next step would overshoot, then go down one layer and repeat.
- In list above, we search for 125 and 127. Gray nodes are looked at; darker gray nodes are overshoots.
- Heights of the nodes were chosen randomly so that there are about 1/2 as many nodes that are $> k$ high as there are that are k high.
- Makes searches fast *with high probability.*

Example: Adding and deleting

- Starting from initial list:



- In any order, we add 126 and 127 (choosing random heights for them), and remove 20 and 40:

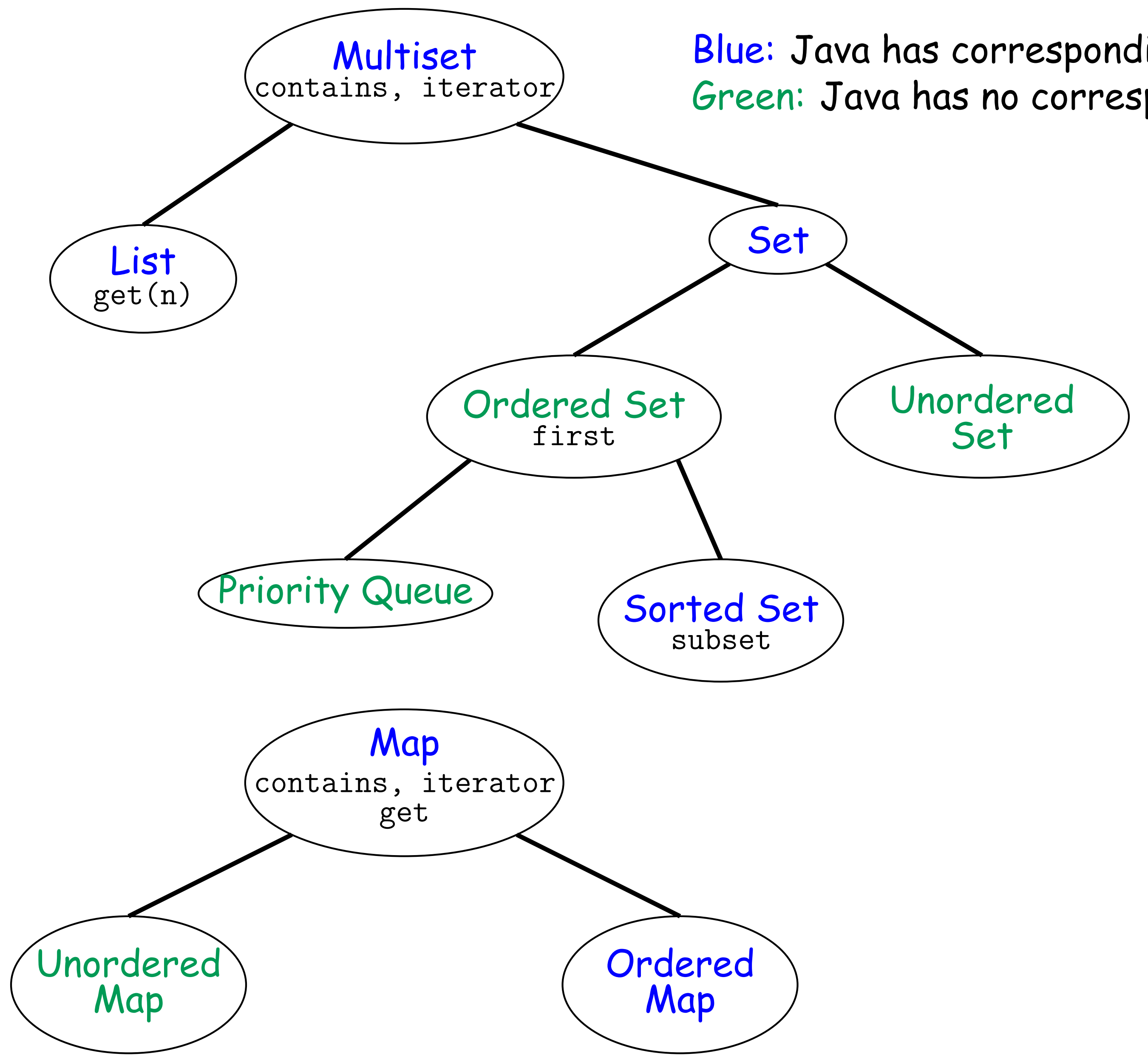


- Shaded nodes here have been modified.

Summary

- Balance in search trees allows us to realize $\Theta(\lg N)$ performance.
- B-trees, red-black trees:
  - Give $\Theta(\lg N)$ performance for searches, insertions, deletions.
  - B-trees good for external storage. Large nodes minimize # of I/O operations
- Tries:
  - Give $\Theta(B)$ performance for searches, insertions, and deletions, where B is length of key being processed.
  - But hard to manage space efficiently.
- *Interesting idea:* scrunched arrays share space.
- Skip lists:
  - Give probable $\Theta(\lg N)$ performace for searches, insertions, deletions
  - Easy to implement.
  - Presented for *interesting ideas:* probabilistic balance, randomized data structures.

Summary of Collection Abstractions



Blue: Java has corresponding interface
Green: Java has no corresponding interface

Data Structures that Implement Abstractions

Multiset

- List: arrays, linked lists, circular buffers
- Set
  - OrderedSet
    - Priority Queue: heaps
    - Sorted Set: binary search trees, red-black trees, B-trees, sorted arrays or linked lists
  - Unordered Set: hash table

Map

- Unordered Map: hash table
- Ordered Map: red-black trees, B-trees, sorted arrays or linked lists

Corresponding Classes in Java

Multiset (Collection)

- List: ArrayList, LinkedList, Stack, ArrayBlockingQueue, ArrayDeque
- Set
  - OrderedSet
    - Priority Queue: PriorityQueue
    - Sorted Set (SortedSet): TreeSet
  - Unordered Set: HashSet

Map

- Unordered Map: HashMap
- Ordered Map (SortedMap): TreeMap

CS61B Lecture #32

Today:

- Pseudo-random Numbers (Chapter 11)
- What use are random sequences?
- What *are* "random sequences"?
- Pseudo-random sequences.
- How to get one.
- Relevant Java library classes and methods.
- Random permutations.

Why Random Sequences?

- Choose statistical samples
- Simulations
- Random algorithms
- Cryptography:
  - Choosing random keys
  - Generating streams of random bits (e.g., SSL xor's your data with a regeneratable, pseudo-random bit stream that only you and the recipient can generate).
- And, of course, games

What Is a “Random Sequence”?

- How about: “a sequence where all numbers occur with equal frequency”?
  - Like 1, 2, 3, 4, . . . ?
- Well then, how about: “an unpredictable sequence where all numbers occur with equal frequency?”
  - Like 0, 0, 0, 1, 1, 2, 2, 2, 2, 2, 3, 4, 4, 0, 1, 1, 1,. . . ?
- Besides, what is wrong with 0, 0, 0, 0, . . . anyway? Can’t that occur by random selection?

Pseudo-Random Sequences

- Even if definable, a "truly" random sequence is difficult for a computer (or human) to produce.
- For most purposes, need only a sequence that satisfies certain statistical properties, even if deterministic.
- Sometimes (e.g., cryptography) need sequence that is *hard* or *impractical* to predict.
- *Pseudo-random sequence:* deterministic sequence that passes some given set of statistical tests.
- For example, look at lengths of *runs:* increasing or decreasing contiguous subsequences.
- Unfortunately, statistical criteria to be used are quite involved. For details, see Knuth.

Generating Pseudo-Random Sequences

- Not as easy as you might think.
- Seemingly complex jumbling methods can give rise to bad sequences.
- *Linear congruential method* is a simple method used by Java:

$$X_0 = \textit{arbitrary seed}$$
$$X_i = (aX_{i-1} + c) \bmod m, \quad i > 0$$

- Usually, m is large power of 2.
- For best results, want $a \equiv 5 \bmod 8$, and a, c, m with no common factors.
- This gives generator with a *period of m* (length of sequence before repetition), and reasonable *potency* (measures certain dependencies among adjacent X_i.)
- Also want bits of a to "have no obvious pattern" and pass certain other tests (see Knuth).
- Java uses $a = 25214903917$, $c = 11$, $m = 2^{48}$, to compute 48-bit pseudo-random numbers. It's good enough for many purposes, but not *cryptographically secure.*

What Can Go Wrong (I)?

- Short periods, many impossible values: E.g., $a,\ c,\ m$ even.
- Obvious patterns. E.g., just using lower 3 bits of X_i in Java's 48-bit generator, to get integers in range 0 to 7. By properties of modular arithmetic,

$$\begin{aligned} X_i \bmod 8 &= (25214903917 X_{i-1} + 11 \bmod 2^{48}) \bmod 8 \\ &= (5(X_{i-1} \bmod 8) + 3) \bmod 8 \end{aligned}$$

  so we have a period of 8 on this generator; sequences like

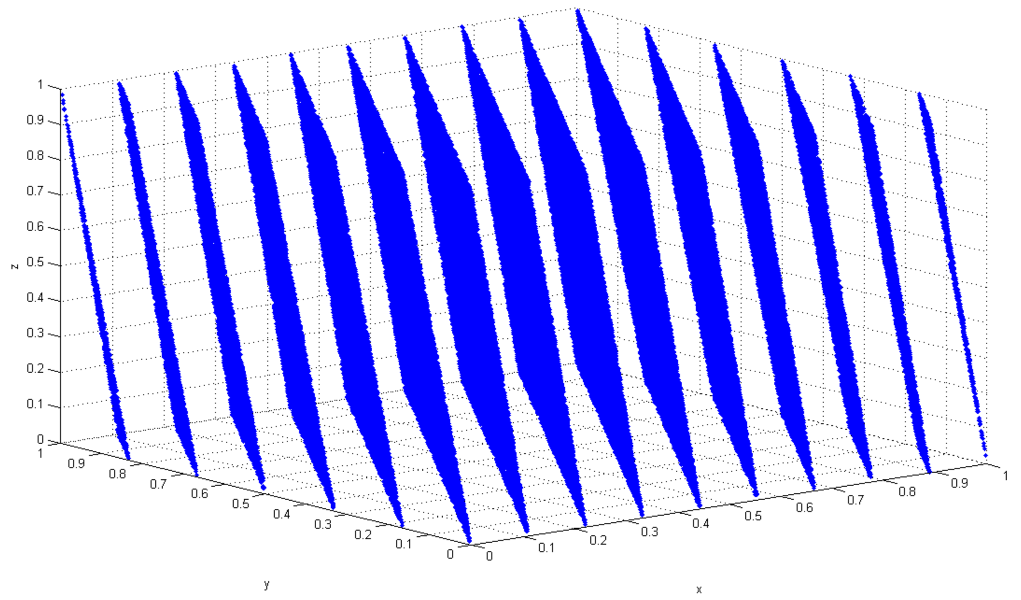
$$0, 1, 3, 7, 1, 2, 7, 1, 4, \ldots$$

  are impossible. This is why Java doesn't give you the raw 48 bits.

What Can Go Wrong (II)?

Bad potency leads to bad correlations.

- The infamous IBM generator RANDU: $c = 0$, $a = 65539$, $m = 2^{31}$.
- When RANDU is used to make 3D points: $(X_i/S, X_{i+1}/S, X_{i+2}/S)$, where S scales to a unit cube, ...
- ...points will be arranged in parallel planes with voids between. So "random points" won't ever get near many points in the cube:



[Credit: Luis Sanchez at English Wikipedia - Transferred from en.wikipedia to Commons by sevela.p., CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=3832343]

Additive Generators

- Additive generator:

$$X_n = \begin{cases} \textit{arbitary value}, & n < 55 \\ (X_{n-24} + X_{n-55}) \bmod 2^e, & n \geq 55 \end{cases}$$

- Other choices than 24 and 55 possible.
- This one has period of $2^f(2^{55} - 1)$, for some $f < e$.
- Simple implementation with circular buffer:

```
i = (i+1) % 55;
X[i] += X[(i+31) % 55];  // Why +31 (55-24) instead of -24?
return X[i];   /* modulo 2^32 */
```

- where `X[0 .. 54]` is initialized to some "random" initial seed values.

Cryptographic Pseudo-Random Number Generators

- The simple form of linear congruential generators means that one can predict future values after seeing relatively few outputs.
- Not good if you want *unpredictable* output (think on-line games involving money or randomly generated keys for encrypting your web traffic.)
- A *cryptographic pseudo-random number generator (CPRNG)* has the properties that
  - Given k bits of a sequence, no polynomial-time algorithm can guess the next bit with better than 50% accuracy.
  - Given the current state of the generator, it is also infeasible to reconstruct the bits it generated in getting to that state.

Cryptographic Pseudo-Random Number Generator Example

- Start with a good *block cipher*—an encryption algorithm that encrypts blocks of N bits (not just one byte at a time as for Enigma). AES is an example.
- As a seed, provide a key, K, and an initialization value I.
- The jth pseudo-random number is now $E(K, I + j)$, where $E(x, y)$ is the encryption of message y using key x.

Adjusting Range and Distribution

- Given raw sequence of numbers, X_i, from above methods in range (e.g.) 0 to 2^{48}, how to get uniform random integers in range 0 to $n-1$?
- If $n = 2^k$, is easy: use top k bits of next X_i (bottom k bits not as "random")
- For other n, be careful of slight biases at the ends. For example, if we compute $X_i/(2^{48}/n)$ using all integer division, and if $(2^{48}/n)$ gets rounded down, then you can get n as a result (which you don't want).
- If you try to fix that by computing $(2^{48}/(n-1))$ instead, the probability of getting $n-1$ will be wrong.

Adjusting Range (II)

- To fix the bias problems when n does not evenly divide 2^{48}, Java throws out values after the largest multiple of n that is less than 2^{48}:

```
/** Random integer in the range 0 .. n-1, n>0. */
int nextInt(int n) {
  long X = next random long ($0 \le X < 2^{48}$);
  if (n is $2^k$ for some $k$)
      return top $k$ bits of X;

  int MAX = largest multiple of n that is $< 2^{48}$;
  while ($X_i$ >= MAX)
        X = next random long ($0 \le X < 2^{48}$);
  return $X_i$ / (MAX/n);
}
```

Arbitrary Bounds

- How to get arbitrary range of integers (L to U)?
- To get random `float`, x in range $0 \leq x < d$, compute

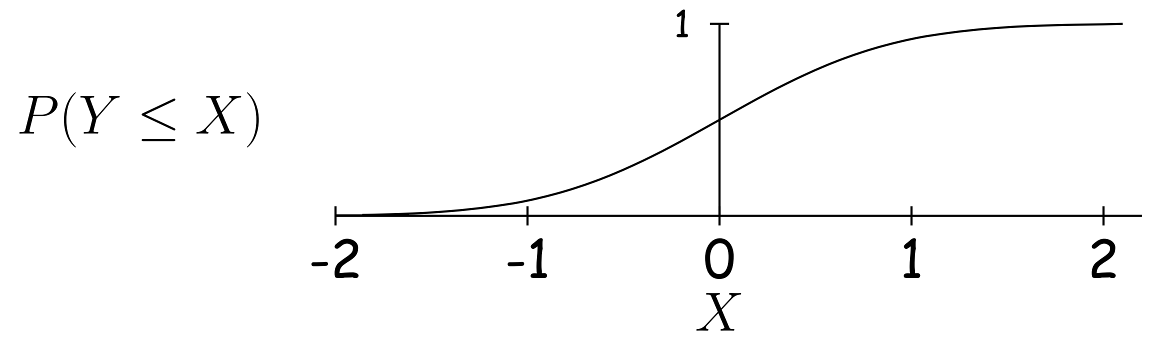
```
return d*nextInt(1<<24) / (1<<24);
```

- Random `double` a bit more complicated: need two integers to get enough bits.

```
long bigRand = ((long) nextInt(1<<26) << 27) + (long) nextInt(1<<27);
return d * bigRand / (1L << 53);
```

Generalizing: Other Distributions

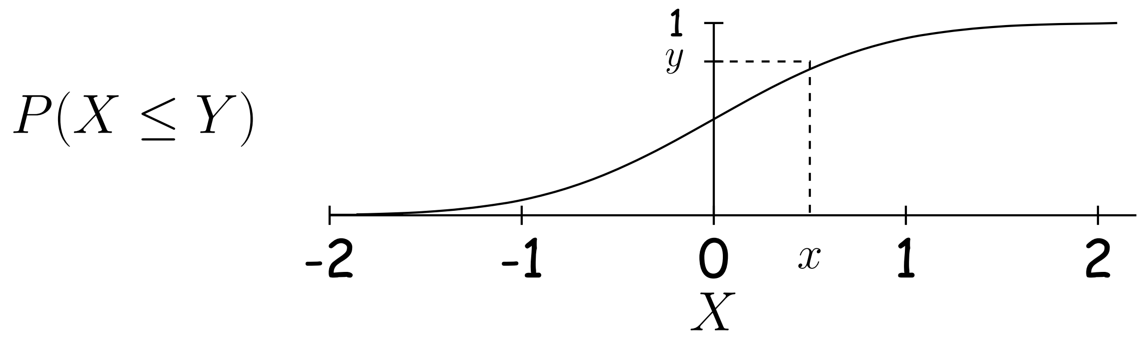
- Suppose we have some desired probability distribution function, and want to get random numbers that are distributed according to that distribution. How can we do this?
- Example: the normal distribution:



- Curve is the desired probability distribution. $P(Y \leq X)$ is the probability that random variable Y is $\leq X$.

Other Distributions

Solution: Choose y uniformly between 0 and 1, and the corresponding x will be distributed according to P.



Java Classes

- `Math.random()`: random double in $[0..1)$.
- Class `java.util.Random`: a random number generator with constructors:

  Random() generator with "random" seed (based on time).

  Random(seed) generator with given starting value (reproducible).
- Methods

  next(k) k-bit random integer

  nextInt(n) `int` in range $[0..n)$.

  nextLong() random 64-bit integer.

  nextBoolean(), nextFloat(), nextDouble() Next random values of other primitive types.

  nextGaussian() normal distribution with mean 0 and standard deviation 1 ("bell curve").
- `Collections.shuffle`(L, R) for list R and `Random` R permutes L randomly (using R).

Shuffling

- A *shuffle* is a random permutation of some sequence.
- Obvious dumb technique for sorting N-element list:
  - Generate N random numbers
  - Attach each to one of the list elements
  - Sort the list using random numbers as keys.
- Can do quite a bit better:

```
void shuffle(List L, Random R) {
   for (int i = L.size(); i > 0; i -= 1)
       swap element i-1 of L with element R.nextInt(i) of L;
}
```

- Example:

Swap items	0	1	2	3	4	5
Start	A♣	2♣	3♣	A♡	2♡	3♡
$5 \Longleftrightarrow 1$	A♣	3♡	3♣	A♡	2♡	2♣
$4 \Longleftrightarrow 2$	A♣	3♡	2♡	A♡	3♣	2♣

Swap items	0	1	2	3	4	5
$3 \Longleftrightarrow 3$	A♣	3♡	2♡	A♡	3♣	2♣
$2 \Longleftrightarrow 0$	2♡	3♡	A♣	A♡	3♣	2♣
$1 \Longleftrightarrow 0$	3♡	2♡	A♣	A♡	3♣	2♣

Random Selection

- Same technique would allow us to select N items from list:

```
/** Permute L and return sublist of K>=0 randomly
 *  chosen elements of L, using R as random source. */
List select(List L, int k, Random R) {
  for (int i = L.size(); i+k > L.size(); i -= 1)
    swap element i-1 of L with element
      R.nextInt(i) of L;
  return L.sublist(L.size()-k, L.size());
}
```

- Not terribly efficient for selecting random sequence of K distinct integers from $[0..N)$, with $K \ll N$.

Alternative Selection Algorithm (Floyd)

```
/** Random sequence of K distinct integers
 *  from 0..N-1, 0<=K<=N. */
IntList selectInts(int N, int K, Random R)
{
  IntList S = new IntList();

  for (int i = N-K; i < N; i += 1) {
    // All values in S are < i
    int s = R.randInt(i+1); // 0 <= s <= i < N
    if (s == S.get(j) for some j)
      // Insert value i (which can't be there
      // yet) after the s (i.e., at a random
      // place other than the front)
      S.add(j+1, i);
    else
      // Insert random value s at front
      S.add(0, s);
  }
  return S;
}
```

Example

i	s	S
5	4	[4]
6	2	[2, 4]
7	5	[5, 2, 4]
8	5	[5, 8, 2, 4]
9	4	[5, 8, 2, 4, 9]

`selectRandomIntegers(10, 5, R)`

CS61B Lecture #33

Today's Readings: Graph Structures: *DSIJ*, Chapter 12

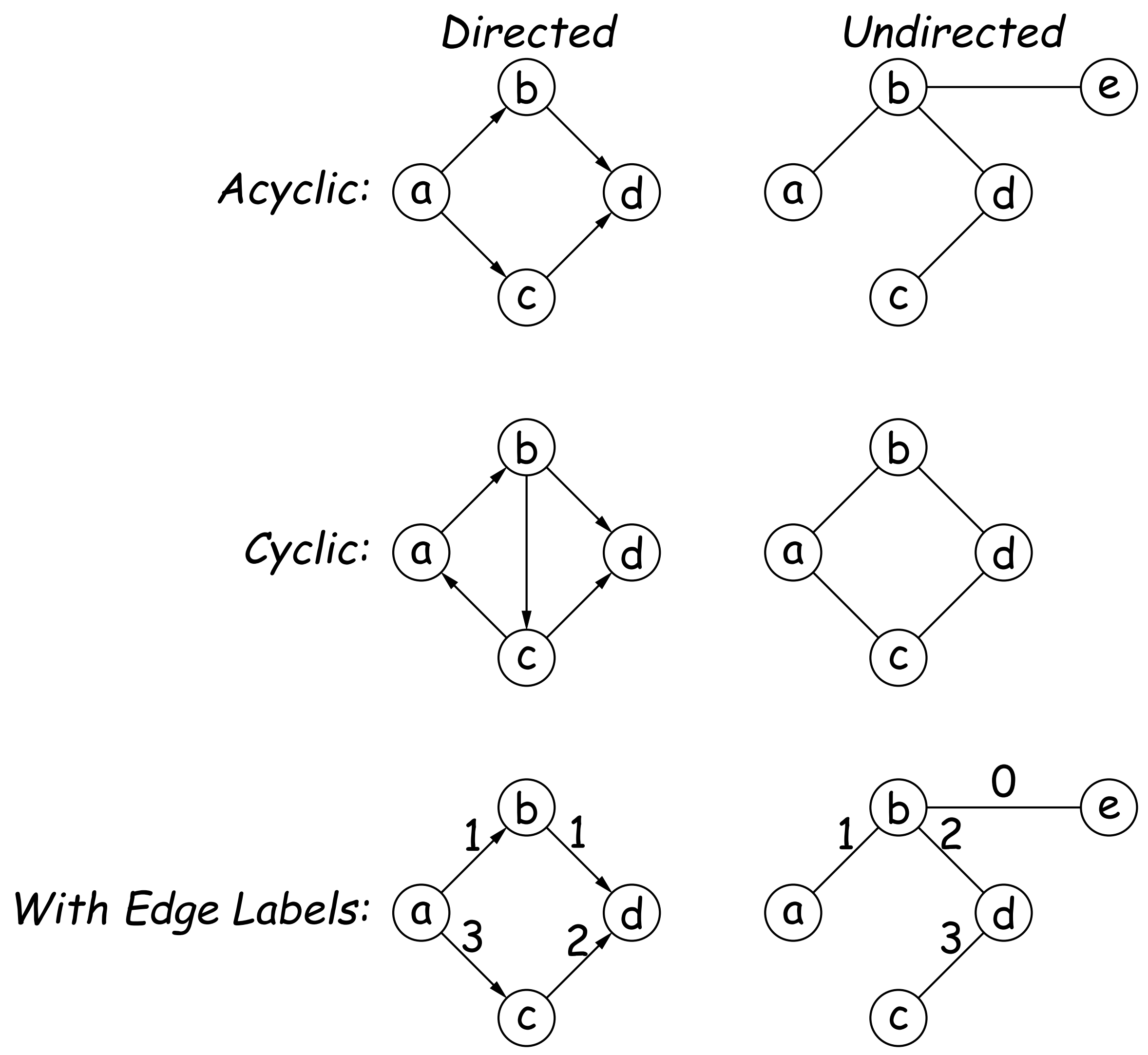
Why Graphs?

- For expressing non-hierarchically related items
- Examples:
  - Networks: pipelines, roads, assignment problems
  - Representing processes: flow charts, Markov models
  - Representing partial orderings: PERT charts, makefiles

Some Terminology

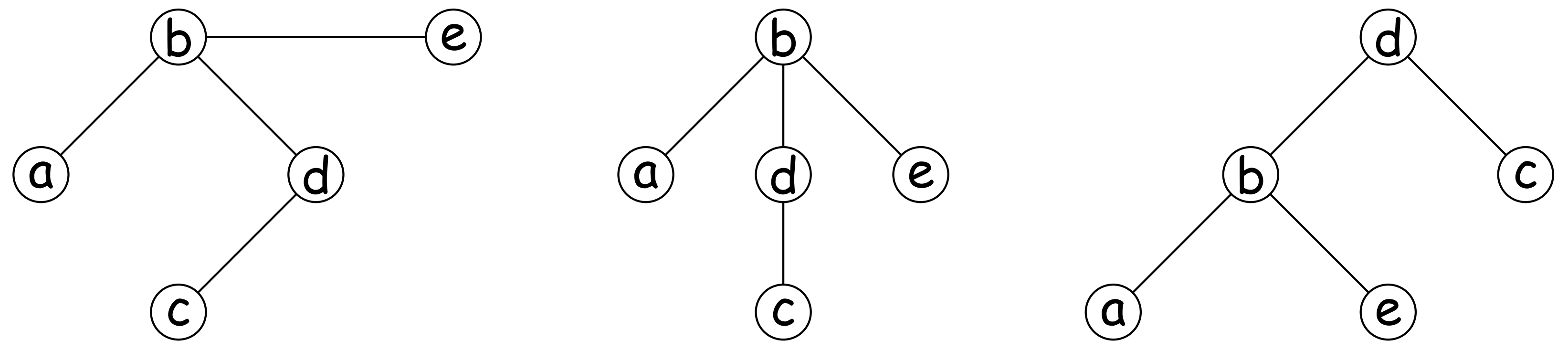
- A *graph* consists of
  - A set of *nodes* (aka *vertices*)
  - A set of *edges:* pairs of nodes.
  - Nodes with an edge between are *adjacent*.
  - Depending on problem, nodes or edges may have *labels* (or *weights*)
- Typically call node set $V = \{v_0, \ldots\}$, and edge set E.
- If the edges have an order (first, second), they are *directed edges,* and we have a *directed graph (digraph),* otherwise an *undirected graph*.
- Edges are *incident* to their nodes.
- Directed edges *exit* one node and *enter* the next.
- A *cycle* is a path without repeated edges leading from a node back to itself (following arrows if directed).
- A graph is *cyclic* if it has a cycle, else *acyclic*. Abbreviation: Directed Acyclic Graph—*DAG*.

Some Pictures



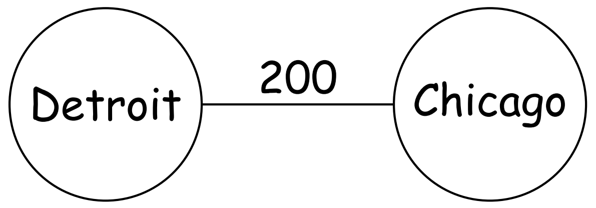
Trees are Graphs

- A graph is *connected* if there is a (possibly directed) path between every pair of nodes.
- That is, if one node of the pair is *reachable* from the other.
- A DAG is a (rooted) tree iff connected, and every node but the root has exactly one parent.
- A connected, acyclic, undirected graph is also called a *free tree.* Free: we're free to pick the root; e.g.,

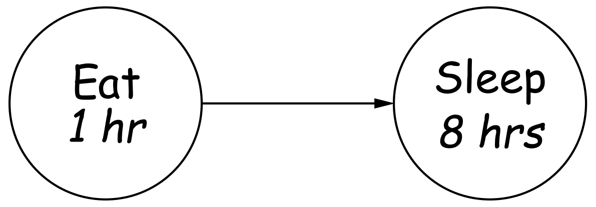


Examples of Use

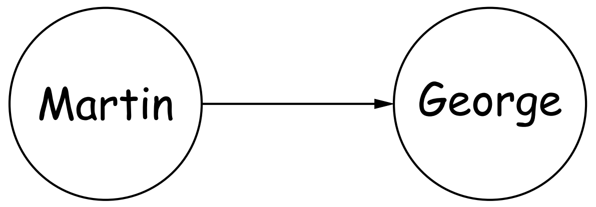
- Edge = Connecting road, with length.



- Edge = Must be completed before; Node label = time to complete.

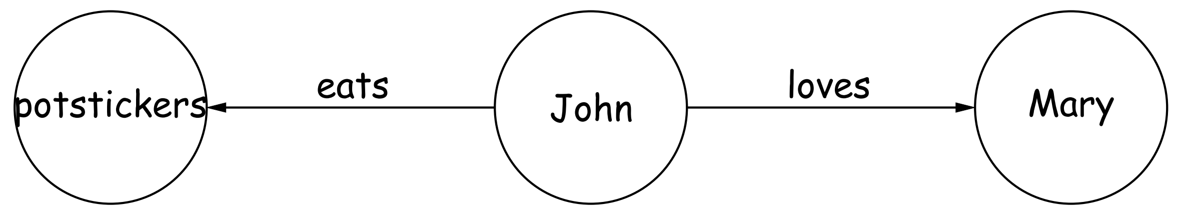


- Edge = Begat

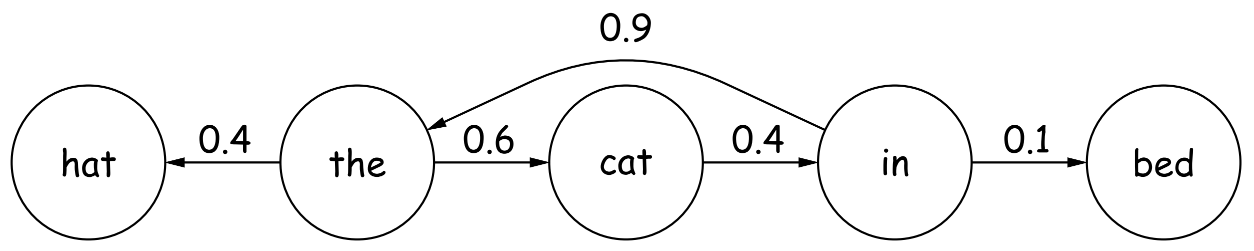


More Examples

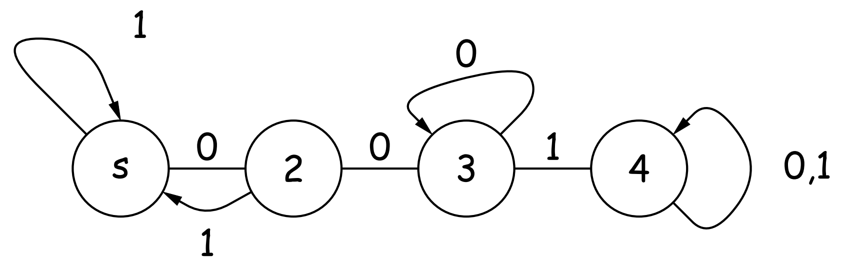
- Edge = some relationship



- Edge = next state might be (with probability)

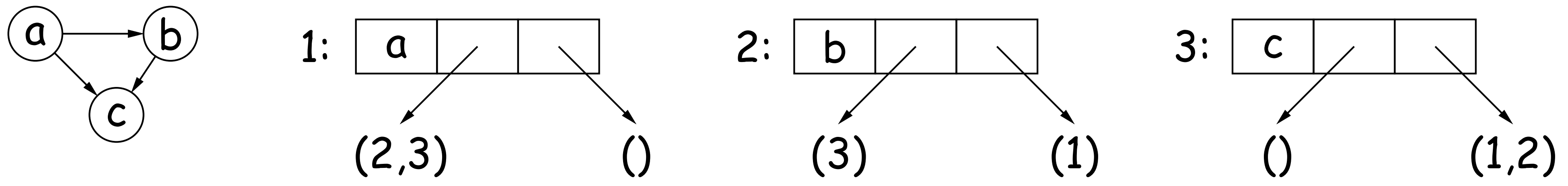


- Edge = next state in state machine, label is triggering input. (Start at s. Being in state 4 means “there is a substring ‘001’ somewhere in the input”.)



Representation

- Often useful to number the nodes, and use the numbers in edges.
- *Edge list representation:* each node contains some kind of list (e.g., linked list or array) of its successors (and possibly predecessors).



- *Edge sets:* Collection of all edges. For graph above:

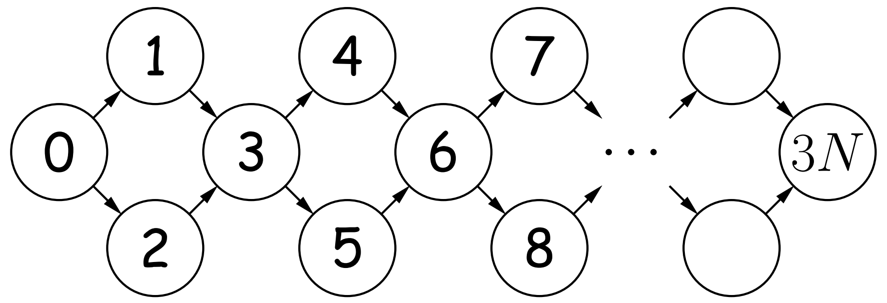
$$\{(1,2),(1,3),(2,3)\}$$

- *Adjacency matrix:* Represent connection with matrix entry:

$$\begin{array}{c|ccc} & \mathit{1} & \mathit{2} & \mathit{3} \\ \hline \mathit{1} & 0 & 1 & 1 \\ \mathit{2} & 0 & 0 & 1 \\ \mathit{3} & 0 & 0 & 0 \end{array}$$

Traversing a Graph

- Many algorithms on graphs depend on traversing all or some nodes.
- Can't quite use recursion because of cycles.
- Even in acyclic graphs, can get combinatorial explosions:



Treat 0 as the root and do recursive traversal down the two edges out of each node: $\Theta(2^N)$ operations!

- So typically try to visit each node constant # of times (e.g., once).

Recursive Depth-First Traversal of a Graph

- Can fix looping and combinatorial problems using the “bread-crumb” method used in earlier lectures for a maze.
- That is, *mark* nodes as we traverse them and don’t traverse previously marked nodes.
- Makes sense to talk about *preorder* and *postorder*, as for trees.

```
void preorderTraverse(Graph G, Node v)
{
   if (v is unmarked) {
     mark(v);
     visit v;
     for (Edge(v, w) ∈ G)
       traverse(G, w);
   }
}
```

```
void postorderTraverse(Graph G, Node v)
{
   if (v is unmarked) {
     mark(v);
     for (Edge(v, w) ∈ G)
       traverse(G, w);
     visit v;
   }
}
```

Recursive Depth-First Traversal of a Graph (II)

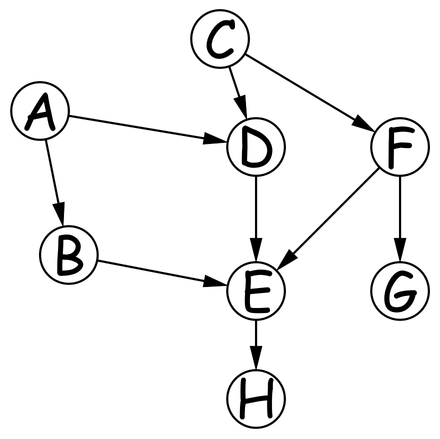
- We are often interested in traversing *all* nodes of a graph, not just those reachable from one node.
- So we can repeat the procedure as long as there are unmarked nodes.

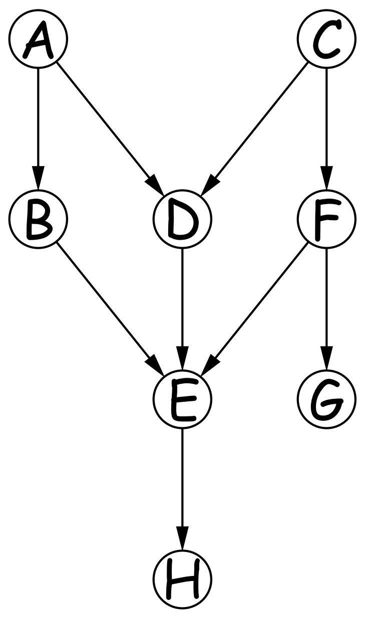
```
void preorderTraverse(Graph G) {
    for (v ∈ nodes of G) {
        preorderTraverse(G, v);
    }
}

void postorderTraverse(Graph G) {
    for (v ∈ nodes of G) {
        postorderTraverse(G, v);
    }
}
```

Topological Sorting

Problem: Given a DAG, find a linear order of nodes consistent with the edges.

- That is, order the nodes $v_0,\ v_1,\ \ldots$ such that v_k is never reachable from $v_{k'}$ if $k' > k$.
- Gmake does this. Also PERT charts.

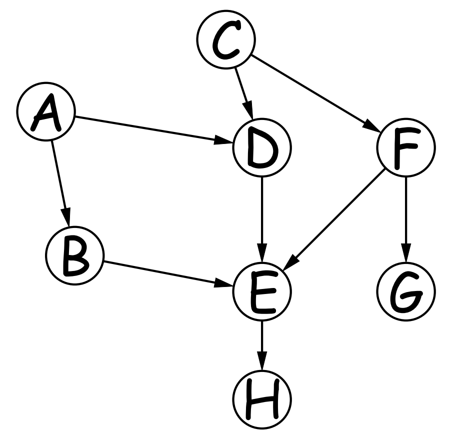


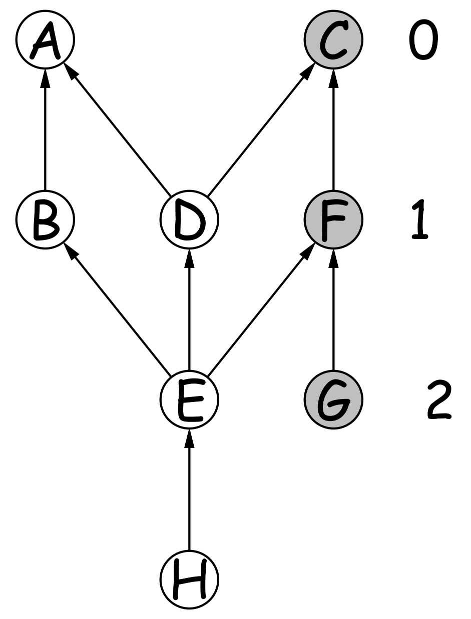


A	C	C
C	A	F
B	F	G
D	D	A
F	B	B
E	G	D
G	E	E
H	H	H

Sorting and Depth First Search

- **Observation**: Suppose we *reverse the links* on our graph.
- If we do a recursive DFS on the reverse graph, starting from node H, for example, we will find all nodes that must come *before* H.
- When the search reaches a node in the reversed graph and there are no successors, we know that it is safe to put that node first.
- In general, a *postorder* traversal of the reversed graph visits nodes only after all predecessors have been visited.





Numbers show post-order traversal order starting from G: everything that must come before G.

General Graph Traversal Algorithm

```
COLLECTION_OF_VERTICES fringe;

fringe = INITIAL_COLLECTION;
while (!fringe.isEmpty()) {
  Vertex v = fringe.REMOVE_HIGHEST_PRIORITY_ITEM();

  if (!MARKED(v)) {
    MARK(v);
    VISIT(v);
    For each edge(v,w) {
      if (NEEDS_PROCESSING(w))
        Add w to fringe;
    }
  }
}
```

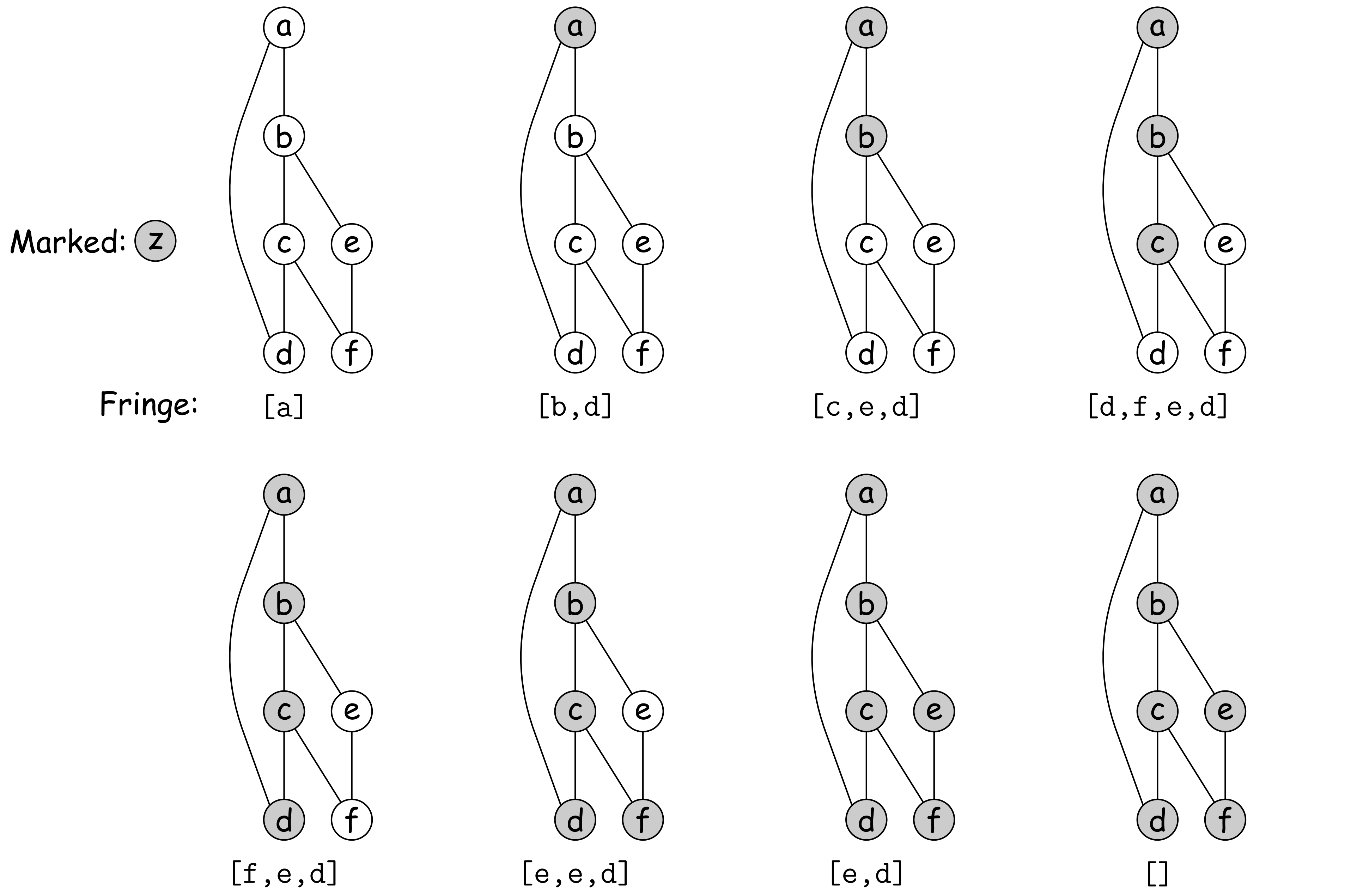
Replace *COLLECTION_OF_VERTICES*, *INITIAL_COLLECTION*, etc. with various types, expressions, or methods to different graph algorithms.

Example: Depth-First Traversal

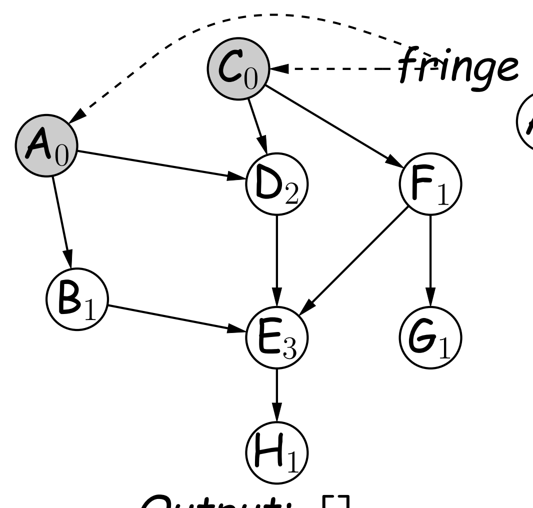
Problem: Visit every node reachable from v once, visiting nodes further from start first.

```
Stack<Vertex> fringe;

fringe = stack containing {v};
while (!fringe.isEmpty()) {
  Vertex v = fringe.pop();

  if (!marked(v)) {
    mark(v);
    VISIT(v);
    For each edge(v,w) {
      if (!marked(w))
        fringe.push(w);
    }
  }
}
```

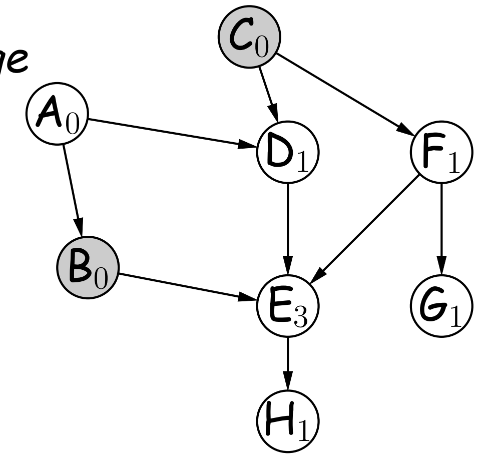
Depth-First Traversal Illustrated



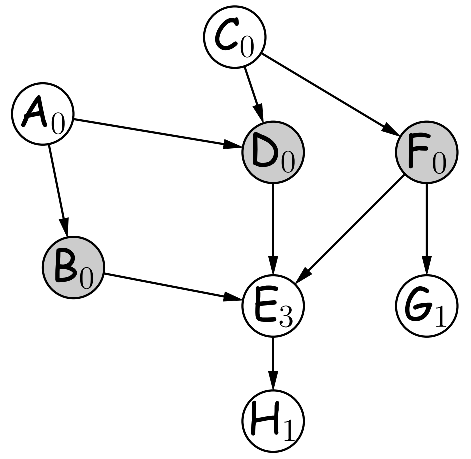
Topological Sort in Action



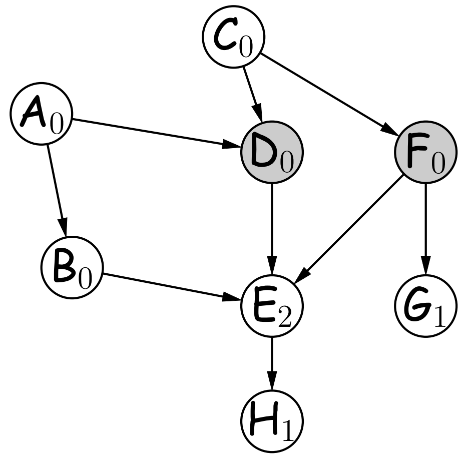
Output: `[]`



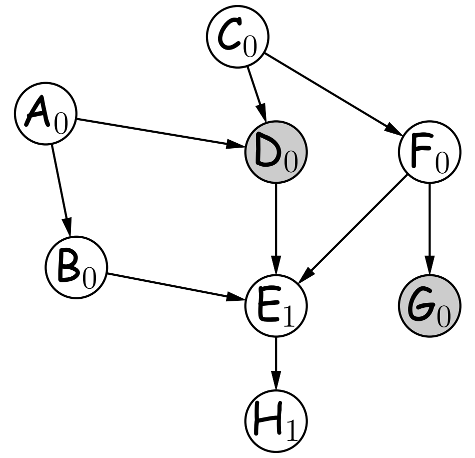
`[A]`



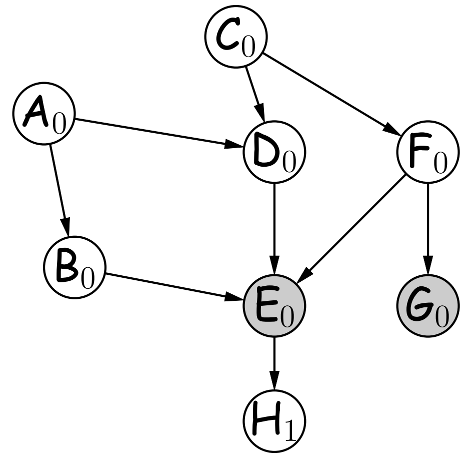
`[A,C]`



`[A,C,B]`

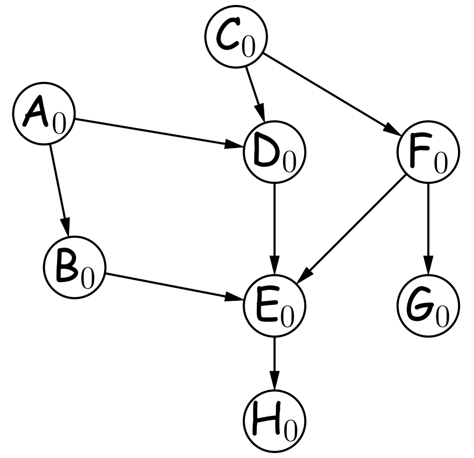


`[A,C,B,F]`



`[A,C,B,F,D]`

…



`[A,C,B,F,D,E,G,H]`

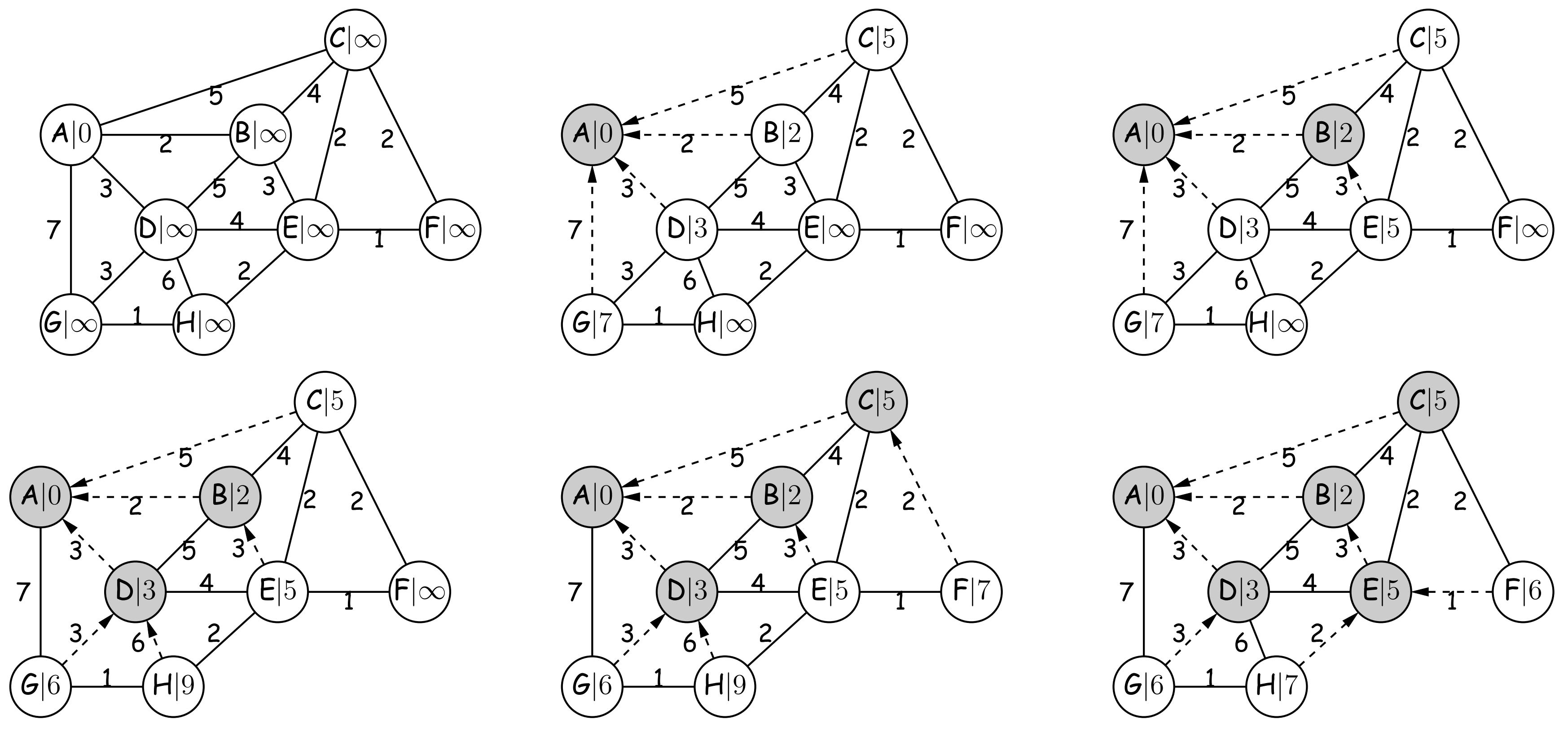
Shortest Paths: Dijkstra's Algorithm

Problem: Given a graph (directed or undirected) with non-negative edge weights, compute shortest paths from given source node, s, to all nodes.

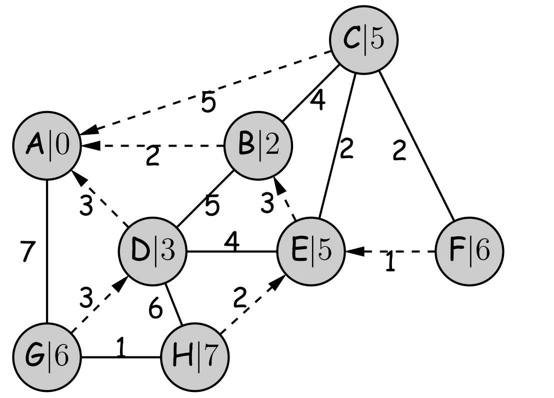
- "Shortest" = sum of weights along path is smallest.
- For each node, keep estimated distance from s, ...
- ...and of preceding node in shortest path from s.

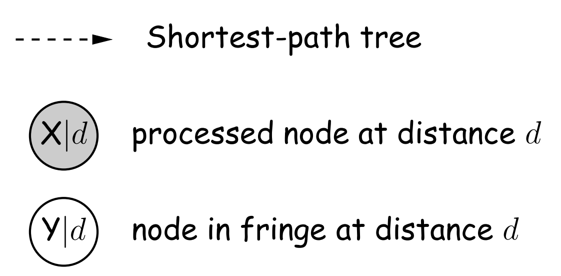
```
PriorityQueue<Vertex> fringe;
For each node v { v.dist() = ∞; v.back() = null; }
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (!fringe.isEmpty()) {
  Vertex v = fringe.removeFirst();

  For each edge(v,w) {
    if (v.dist() + weight(v,w) < w.dist())
      { w.dist() = v.dist() + weight(v,w); w.back() = v; }
  }
}
```

Example



Final result:




Shortest-path tree
$X|d$ processed node at distance d
$Y|d$ node in fringe at distance d

CS61B Lecture #34

- **Today:** A* search, Minimum spanning trees, union-find.

Point-to-Point Shortest Path

- Dijkstra's algorithm gives you shortest paths from a particular given vertex to all others in a graph.
- But suppose you're only interested in getting to a particular vertex?
- Because the algorithm finds paths in order of length, you *could* simply run it and stop when you get to the vertex you want.
- But, this can be really wasteful.
- For example, to travel by road from Denver to a destination on lower Fifth Avenue in New York City is about 1750 miles (says Google).
- But traveling from Denver to the Gourmet Ghetto in Berkeley is about 1650 miles.
- So, we'd explore much of California, Nevada, Arizona, etc. before we found our destination, even though these are all in the wrong direction!
- Situation even worse when graph is infinite, generated on the fly.

A* Search

- We're looking for a path from vertex Denver to the desired NYC vertex.
- Suppose that we had a *heuristic guess*, $h(V)$, of the length of a path from any vertex V to NYC.
- And suppose that instead of visiting vertices in the fringe in order of their shortest known path to Denver, we order by the sum of that distance plus a *heuristic estimate* of the remaining distance to NYC: $d(\text{Denver}, V) + h(V)$.
- In other words, we look at places that are reachable from places where we already know the shortest path to Denver and choose those that look like they will result in the shortest trip to NYC, guessing at the remaining distance.
- If the estimate is good, then we don't look at, say, Grand Junction (250 miles west by road), because it's in the wrong direction.
- The resulting algorithm is *A* search*.
- But for it to work, we must be careful about the heuristic.

Admissible Heuristics for A* Search

- If our heuristic estimate for the distance to NYC is too high (i.e., larger than the actual path by road), then we may get to NYC without ever examining points along the shortest route.
- For example, if our heuristic decided that the midwest was literally the middle of nowhere, and $h(C) = 2000$ for C any city in Michigan or Indiana, we'd only find a path that detoured south through Kentucky.
- So to be *admissible,* $h(C)$ must never overestimate $d(C, \text{NYC})$, the minimum path distance from C to NYC.
- On the other hand, $h(C) = 0$ will work (what is the result?), but yield a non-optimal algorithm.

Consistency

- Suppose that we estimate $h(\text{Chicago}) = 700$, and $h(\text{Springfield, IL}) = 200$, where $d(\text{Chicago, Springfield}) = 200$.
- So by driving 200 miles to Springfield, we guess that we are suddenly 500 miles closer to NYC.
- This is admissible, since both estimates are low, but it will mess up our algorithm.
- Specifically, will require that we put processed nodes back into the fringe, in case our estimate was wrong.
- So (in this course, anyway) we also require *consistent heuristics:* $h(A) \leq h(B) + d(A, B)$, as for the triangle inequality.
- All consistent heuristics are admissible (why?).
- For project 3, distance "as the crow flies" is a good $h(\cdot)$ in the trip application.
- Demo of A* search (and others) is in `cs61b-software` and on the instructional machines as `graph-demo`.

Summary of Shortest Paths

- Dijkstra's algorithm finds a *shortest-path tree* computing giving (backwards) shortest paths in a weighted graph from a given starting node to all other nodes.
- Time required =
  - Time to remove V nodes from priority queue +
  - Time to update all neighbors of each of these nodes and add or reorder them in queue ($E \lg E$)
  - $\in \Theta(V \lg V + E \lg V) = \Theta((V + E) \lg V)$
- A* search searches for a shortest path to a *particular* target node.
- Same as Dijkstra's algorithm, except:
  - Stop when we take target from queue.
  - Order queue by estimated distance to start + heuristic guess of remaining distance ($h(v) = d(v, \text{target})$)
  - Heuristic must not overestimate distance and obey triangle inequality ($d(a, b) + d(b, c) \geq d(a, c)$).

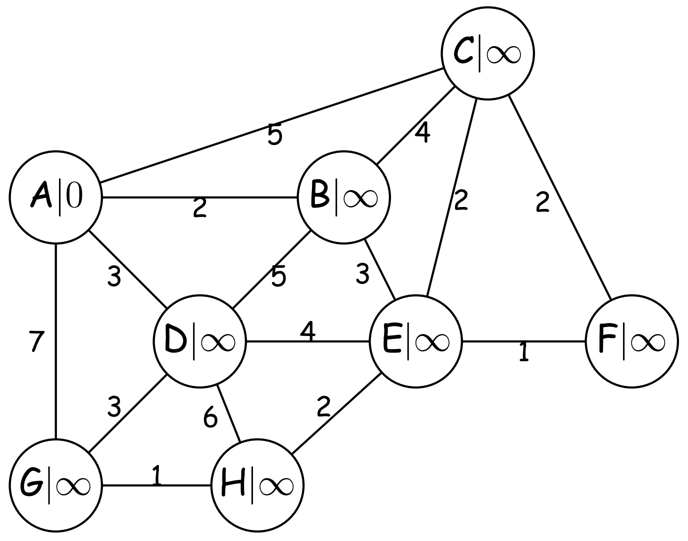
Minimum Spanning Trees

- **Problem:** Given a set of places and distances between them (assume always positive), find a set of connecting roads of minimum total length that allows travel between any two.
- The routes you get will not necessarily be shortest paths.
- Easy to see that such a set of connecting roads and places must form a tree, because removing one road in a cycle still allows all to be reached.

Minimum Spanning Trees by Prim's Algorithm

- Idea is to grow a tree starting from an arbitrary node.
- At each step, add the shortest edge connecting some node already in the tree to one that isn't yet.
- Why must this work?

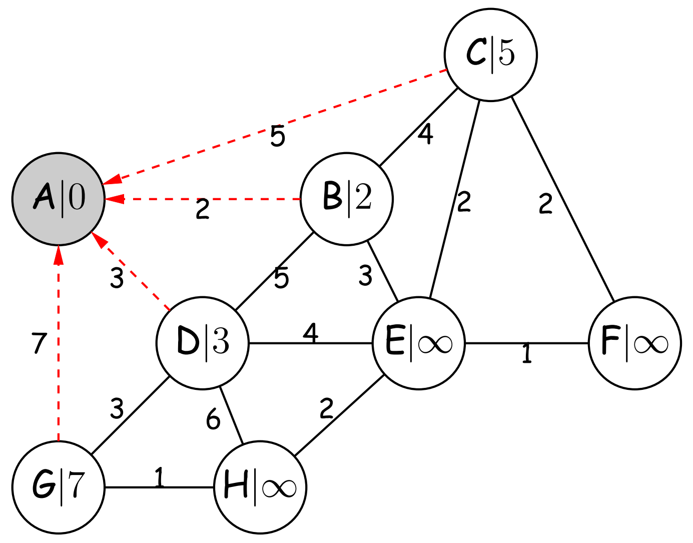
```
PriorityQueue fringe;
For each node v { v.dist() = ∞; v.parent() = null; }
Choose an arbitrary starting node, s;
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (!fringe.isEmpty()) {
  Vertex v = fringe.removeFirst();

  For each edge(v,w) {
    if (w ∈ fringe && weight(v,w) < w.dist())
      { w.dist() = weight(v, w); w.parent() = v; }
  }
}
```



Minimum Spanning Trees by Prim's Algorithm

- Idea is to grow a tree starting from an arbitrary node.
- At each step, add the shortest edge connecting some node already in the tree to one that isn't yet.
- Why must this work?

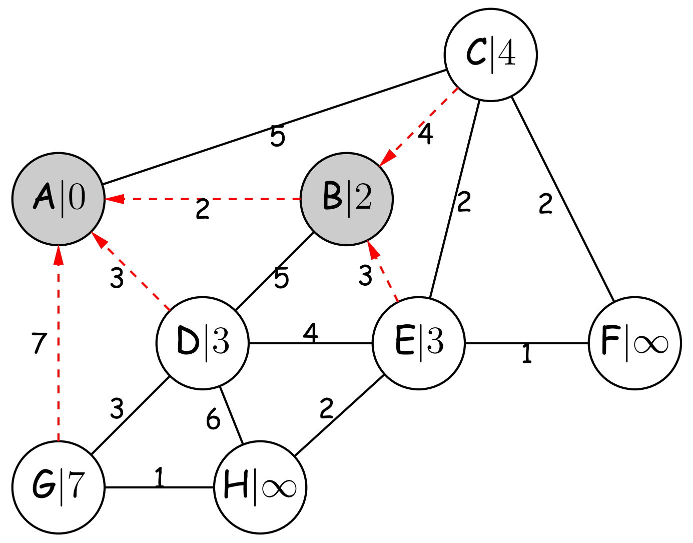
```
PriorityQueue fringe;
For each node v { v.dist() = ∞; v.parent() = null; }
Choose an arbitrary starting node, s;
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (!fringe.isEmpty()) {
  Vertex v = fringe.removeFirst();

  For each edge(v,w) {
    if (w ∈ fringe && weight(v,w) < w.dist())
      { w.dist() = weight(v, w); w.parent() = v; }
  }
}
```



Minimum Spanning Trees by Prim's Algorithm

- Idea is to grow a tree starting from an arbitrary node.
- At each step, add the shortest edge connecting some node already in the tree to one that isn't yet.
- Why must this work?

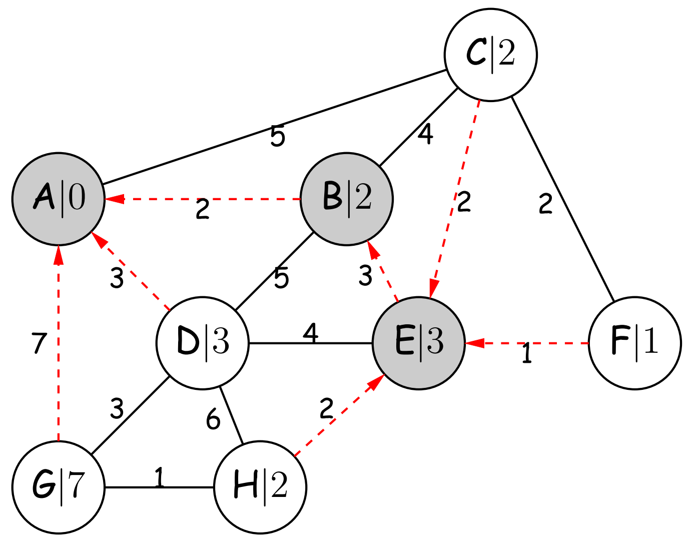
```
PriorityQueue fringe;
For each node v { v.dist() = ∞; v.parent() = null; }
Choose an arbitrary starting node, s;
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (!fringe.isEmpty()) {
  Vertex v = fringe.removeFirst();

  For each edge(v,w) {
    if (w ∈ fringe && weight(v,w) < w.dist())
      { w.dist() = weight(v, w); w.parent() = v; }
  }
}
```



Minimum Spanning Trees by Prim's Algorithm

- Idea is to grow a tree starting from an arbitrary node.
- At each step, add the shortest edge connecting some node already in the tree to one that isn't yet.
- Why must this work?

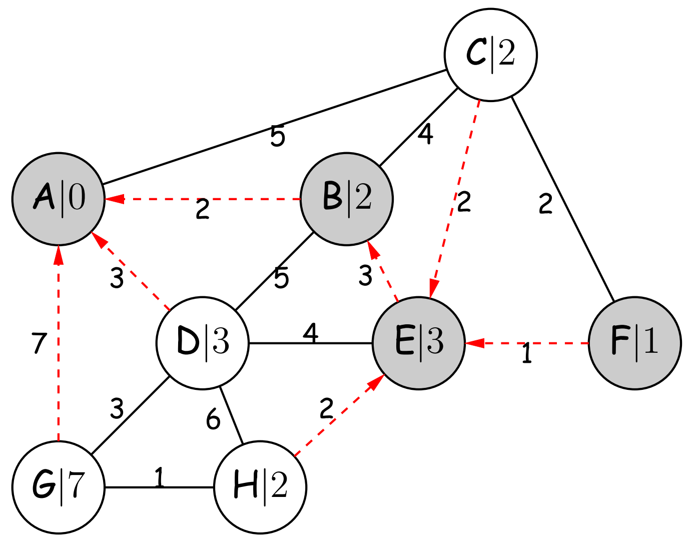
```
PriorityQueue fringe;
For each node v { v.dist() = ∞; v.parent() = null; }
Choose an arbitrary starting node, s;
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (!fringe.isEmpty()) {
  Vertex v = fringe.removeFirst();

  For each edge(v,w) {
    if (w ∈ fringe && weight(v,w) < w.dist())
      { w.dist() = weight(v, w); w.parent() = v; }
  }
}
```



Minimum Spanning Trees by Prim's Algorithm

- Idea is to grow a tree starting from an arbitrary node.
- At each step, add the shortest edge connecting some node already in the tree to one that isn't yet.
- Why must this work?

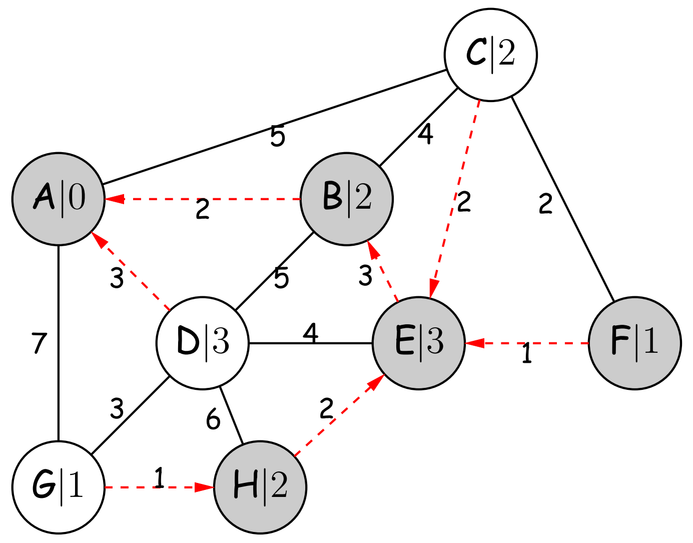
```
PriorityQueue fringe;
For each node v { v.dist() = ∞; v.parent() = null; }
Choose an arbitrary starting node, s;
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (!fringe.isEmpty()) {
  Vertex v = fringe.removeFirst();

  For each edge(v,w) {
    if (w ∈ fringe && weight(v,w) < w.dist())
      { w.dist() = weight(v, w); w.parent() = v; }
  }
}
```



Minimum Spanning Trees by Prim's Algorithm

- Idea is to grow a tree starting from an arbitrary node.
- At each step, add the shortest edge connecting some node already in the tree to one that isn't yet.
- Why must this work?

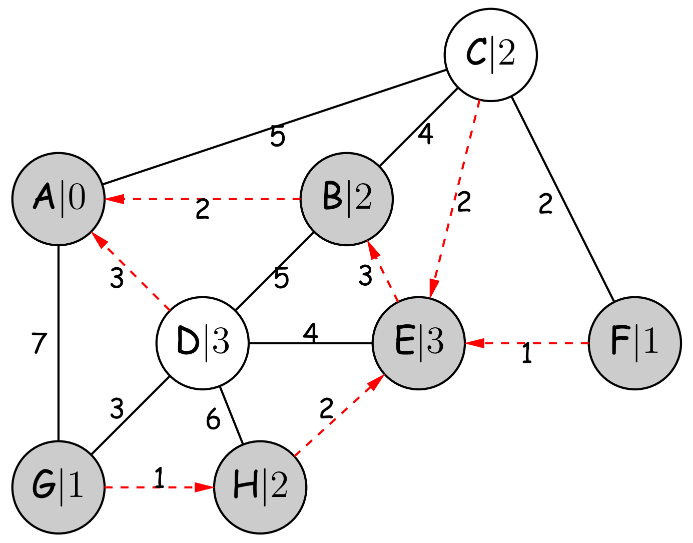
```
PriorityQueue fringe;
For each node v { v.dist() = ∞; v.parent() = null; }
Choose an arbitrary starting node, s;
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (!fringe.isEmpty()) {
  Vertex v = fringe.removeFirst();

  For each edge(v,w) {
    if (w ∈ fringe && weight(v,w) < w.dist())
      { w.dist() = weight(v, w); w.parent() = v; }
  }
}
```



Minimum Spanning Trees by Prim's Algorithm

- Idea is to grow a tree starting from an arbitrary node.
- At each step, add the shortest edge connecting some node already in the tree to one that isn't yet.
- Why must this work?

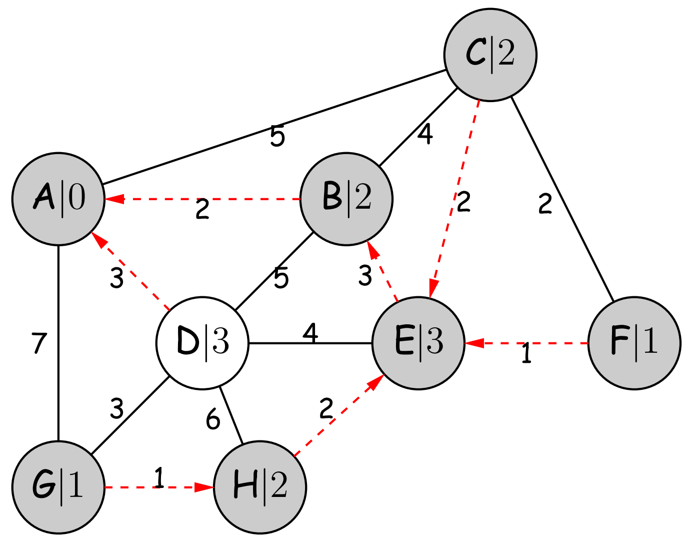
```
PriorityQueue fringe;
For each node v { v.dist() = ∞; v.parent() = null; }
Choose an arbitrary starting node, s;
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (!fringe.isEmpty()) {
  Vertex v = fringe.removeFirst();

  For each edge(v,w) {
    if (w ∈ fringe && weight(v,w) < w.dist())
      { w.dist() = weight(v, w); w.parent() = v; }
  }
}
```



Minimum Spanning Trees by Prim's Algorithm

- Idea is to grow a tree starting from an arbitrary node.
- At each step, add the shortest edge connecting some node already in the tree to one that isn't yet.
- Why must this work?

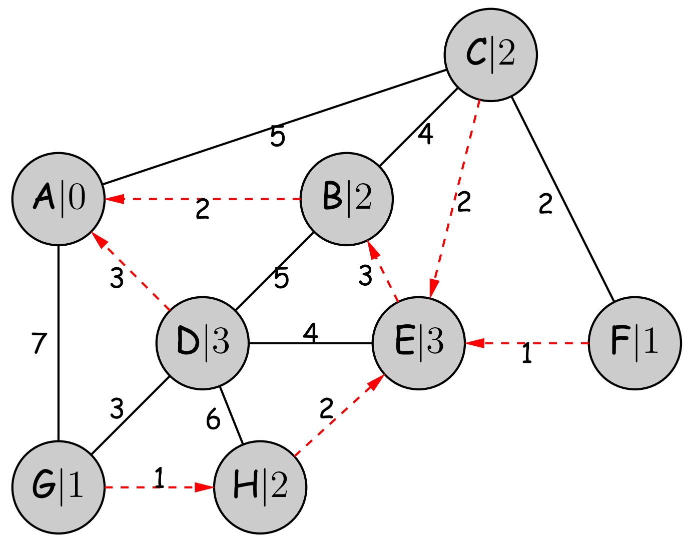
```
PriorityQueue fringe;
For each node v { v.dist() = ∞; v.parent() = null; }
Choose an arbitrary starting node, s;
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (!fringe.isEmpty()) {
  Vertex v = fringe.removeFirst();

  For each edge(v,w) {
    if (w ∈ fringe && weight(v,w) < w.dist())
      { w.dist() = weight(v, w); w.parent() = v; }
  }
}
```



Minimum Spanning Trees by Prim's Algorithm

- Idea is to grow a tree starting from an arbitrary node.
- At each step, add the shortest edge connecting some node already in the tree to one that isn't yet.
- Why must this work?

```
PriorityQueue fringe;
For each node v { v.dist() = ∞; v.parent() = null; }
Choose an arbitrary starting node, s;
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (!fringe.isEmpty()) {
  Vertex v = fringe.removeFirst();

  For each edge(v,w) {
    if (w ∈ fringe && weight(v,w) < w.dist())
      { w.dist() = weight(v, w); w.parent() = v; }
  }
}
```



Minimum Spanning Trees by Kruskal's Algorithm

- Observation: the shortest edge in a graph can always be part of a minimum spanning tree.
- In fact, if we have a bunch of subtrees of a MST, then the shortest edge that connects two of them can be part of a MST, combining the two subtrees into a bigger one.
- So,...

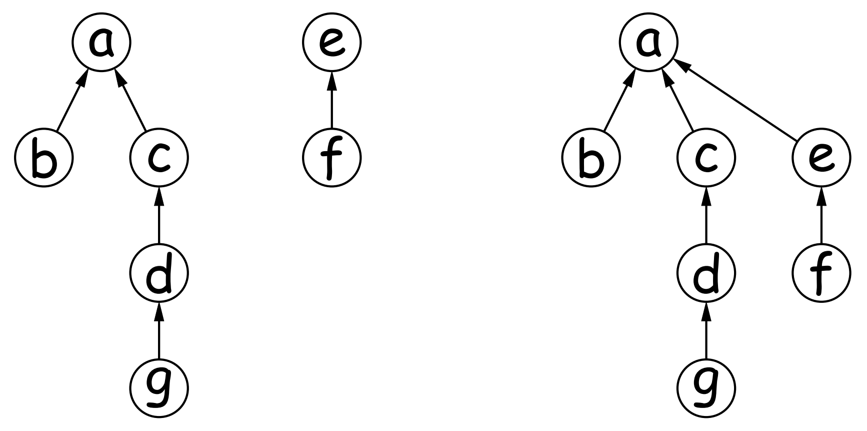
```
Create one (trivial) subtree for each node in the graph;
MST = {};

for each edge(v,w), in increasing order of weight {
   if ( (v,w) connects two different subtrees ) {
      Add (v,w) to MST;
      Combine the two subtrees into one;
   }
}
```

Union Find

- Kruskal's algorithm required that we have a set of sets of nodes with two operations:
  - *Find* which of the sets a given node belongs to.
  - Replace two sets with their *union,* reassigning all the nodes in the two original sets to this union.
- Obvious thing to do is to store a set number in each node, making finds fast.
- Union requires changing the set number in one of the two sets being merged; the smaller is better choice.
- This means an individual union can take $\Theta(N)$ time.
- Can union be fast?

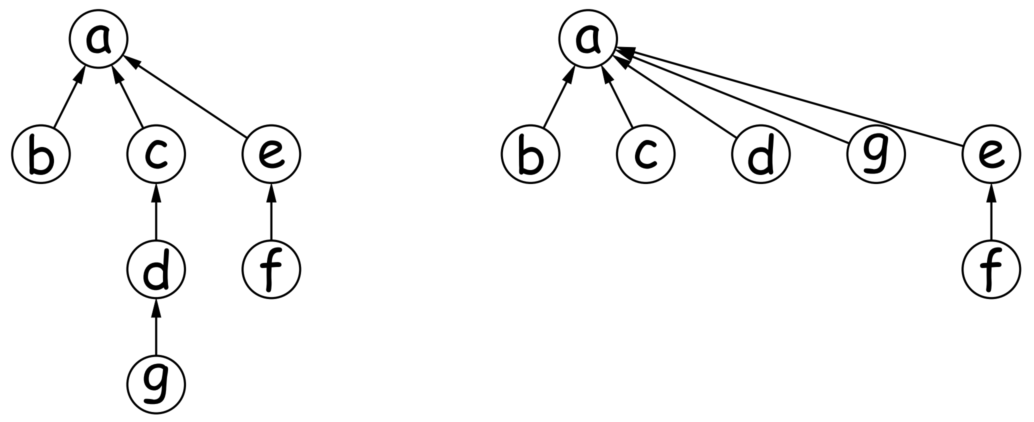
A Clever Trick

- Let's choose to represent a set of nodes by *one* arbitrary representative node in that set.
- Let every node contain a pointer to another node in the same set.
- Arrange for each pointer to represent the *parent* of a node in a tree that has the representative node as its root.
- To find what set a node is in, follow parent pointers.
- To union two such trees, make one root point to the other (choose the root of the larger tree as the union representative).



Path Compression

- This makes unioning really fast, but the find operation potentially slow ($\Omega(\lg N)$).
- So use the following trick: whenever we do a *find* operation, *compress* the path to the root, so that subsequent finds will be faster.
- That is, make each of the nodes in the path point directly to the root.
- Now union is very fast, and sequence of unions and finds each have very, *very* nearly constant amortized time.
- Example: find 'g' in last tree (result of compression on right):



Lecture #35

Today

- Dynamic programming and memoization.
- Anatomy of Git.

Dynamic Programming

- A puzzle (D. Garcia):
  - Start with a list with an even number of non-negative integers.
  - Each player in turn takes either the leftmost number or the rightmost.
  - Idea is to get the largest possible sum.
- Example: starting with (6, 12, 0, 8), you (as first player) should take the 8. Whatever the second player takes, you also get the 12, for a total of 20.
- Assuming your opponent plays perfectly (i.e., to get as much as possible), how can you maximize your sum?
- Can solve this with exhaustive game-tree search.

Obvious Program

- Recursion makes it easy, again:

```
int bestSum(int[] V) {
  int total, i, N = V.length;
  for (i = 0, total = 0; i < N; i += 1) total += V[i];
  return bestSum(V, 0, N-1, total);
}

/** The largest sum obtainable by the first player in the choosing
 *  game on the list V[LEFT .. RIGHT], assuming that TOTAL is the
 *  sum of all the elements in V[LEFT .. RIGHT].  */
int bestSum(int[] V, int left, int right, int total) {
  if (left > right)
    return 0;
  else {
    int L = total - bestSum(V, left+1, right, total-V[left]);
    int R = total - bestSum(V, left, right-1, total-V[right]);
    return Math.max(L, R);
 }
}
```

- Time cost is $C(0) = 1,\ C(N) = 2C(N-1)$; so $C(N) \in \Theta(2^N)$

Still Another Idea from CS61A

- The problem is that we are recomputing intermediate results many times.
- Solution: *memoize* the intermediate results. Here, we pass in an $N \times N$ array ($N =$ `V.length`) of memoized results, initialized to -1.

```
int bestSum(int[] V, int left, int right, int total, int[][] memo) {
  if (left > right)
    return 0;
  else if (memo[left][right] == -1) {
    int L = total - bestSum(V, left+1, right, total-V[left], memo);
    int R = total - bestSum(V, left, right-1, total-V[right], memo);
    memo[left][right] = Math.max(L, R);
  }
  return memo[left][right];
 }
}
```

- Now the number of recursive calls to `bestSum` must be $O(N^2)$, for $N =$ the length of V, an enormous improvement from $\Theta(2^N)$!

Iterative Version

- I prefer the recursive version, but the usual presentation of this idea—known as *dynamic programming*—is iterative:

```
int bestSum(int[] V) {
  int[][] memo = new int[V.length][V.length];
  int[][] total = new int[V.length][V.length];
  for (int i = 0; i < V.length; i += 1)
    memo[i][i] = total[i][i] = V[i];
  for (int k = 1; k < V.length; k += 1)
    for (int i = 0; i < V.length-k-1; i += 1) {
      total[i][i+k] = V[i] + total[i+1][i+k];
      int L = total[i][i+k] - memo[i+1][i+k];
      int R = total[i][i+k] - memo[i][i+k-1];
      memo[i][i+k] = Math.max(L, R);
    }
  return memo[0][V.length-1];
}
```

- That is, we figure out ahead of time the order in which the memoized version will fill in `memo`, and write an explicit loop.
- Save the time needed to check whether result exists.
- But I say, why bother unless it's necessary to save space?

Longest Common Subsequence

- **Problem:** Find length of the longest string that is a subsequence of each of two other strings.
- **Example:** Longest common subsequence of
  "`sally␣sells␣sea␣shells␣by␣the␣seashore`" and
  "`sarah␣sold␣salt␣sellers␣at␣the␣salt␣mines`"
  is
  "`sa␣sl␣sa␣sells␣␣the␣sae`" (length 23)
- Similarity testing, for example.
- Obvious recursive algorithm:

```
/** Length of longest common subsequence of S0[0..k0-1]
 *  and S1[0..k1-1] (pseudo Java) */
static int lls(String S0, int k0, String S1, int k1) {
  if (k0 == 0 || k1 == 0) return 0;
  if (S0[k0-1] == S1[k1-1]) return 1 + lls(S0, k0-1, S1, k1-1);
  else return Math.max(lls(S0, k0-1, S1, k1), lls(S0, k0, S1, k1-1);
}
```

- Exponential, but obviously memoizable.

Memoized Longest Common Subsequence

```
/** Length of longest common subsequence of S0[0..k0-1]
 *  and S1[0..k1-1] (pseudo Java) */
static int lls(String S0, int k0, String S1, int k1) {
  int[][] memo = new int[k0+1][k1+1];
  for (int[] row : memo) Arrays.fill(row, -1);
  return lls(S0, k0, S1, k1, memo);
}

private static int lls(String S0, int k0, String S1, int k1, int[][] memo) {
  if (k0 == 0 || k1 == 0) return 0;
  if (memo[k0][k1] == -1) {
    if (S0[k0-1] == S1[k1-1])
      memo[k0][k1] = 1 + lls(S0, k0-1, S1, k1-1, memo);
    else
      memo[k0][k1] = Math.max(lls(S0, k0-1, S1, k1, memo),
                              lls(S0, k0, S1, k1-1, memo));
  }
  return memo[k0][k1];
}
```

Q: How fast will the memoized version be?

Memoized Longest Common Subsequence

```
/** Length of longest common subsequence of S0[0..k0-1]
 *  and S1[0..k1-1] (pseudo Java) */
static int lls(String S0, int k0, String S1, int k1) {
  int[][] memo = new int[k0+1][k1+1];
  for (int[] row : memo) Arrays.fill(row, -1);
  return lls(S0, k0, S1, k1, memo);
}

private static int lls(String S0, int k0, String S1, int k1, int[][] memo) {
  if (k0 == 0 || k1 == 0) return 0;
  if (memo[k0][k1] == -1) {
    if (S0[k0-1] == S1[k1-1])
      memo[k0][k1] = 1 + lls(S0, k0-1, S1, k1-1, memo);
    else
      memo[k0][k1] = Math.max(lls(S0, k0-1, S1, k1, memo),
                              lls(S0, k0, S1, k1-1, memo));
  }
  return memo[k0][k1];
}
```

Q: How fast will the memoized version be? $\Theta(k_0 \cdot k_1)$

Git: A Case Study in System and Data-Structure Design

- Git is a distributed version-control system, apparently the most popular of these currently.
- Conceptually, it stores snapshots (*versions*) of the files and directory structure of a project, keeping track of their relationships, authors, dates, and log messages.
- It is *distributed,* in that there can be many copies of a given repository, each supporting indepenent development, with machinery to transmit and reconcile versions between repositories.
- Its operation is extremely fast (as these things go).

A Little History

- Developed by Linus Torvalds and others in the Linux community when the developer of their previous, propietary VCS (Bitkeeper) withdrew the free version.
- Initial implementation effort seems to have taken about 2–3 months, in time for the 2.6.12 Linux kernel release in June, 2005.
- As for the name, according to Wikipedia,

  > Torvalds has quipped about the name Git, which is British English slang meaning "unpleasant person". Torvalds said: "I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'." The man page describes Git as "the stupid content tracker."

- Initially, was a collection of basic primitives (now called "plumbing") that could be scripted to provide desired functionality.
- Then, higher-level commands ("porcelain") built on top of these to provide a convenient user interface.

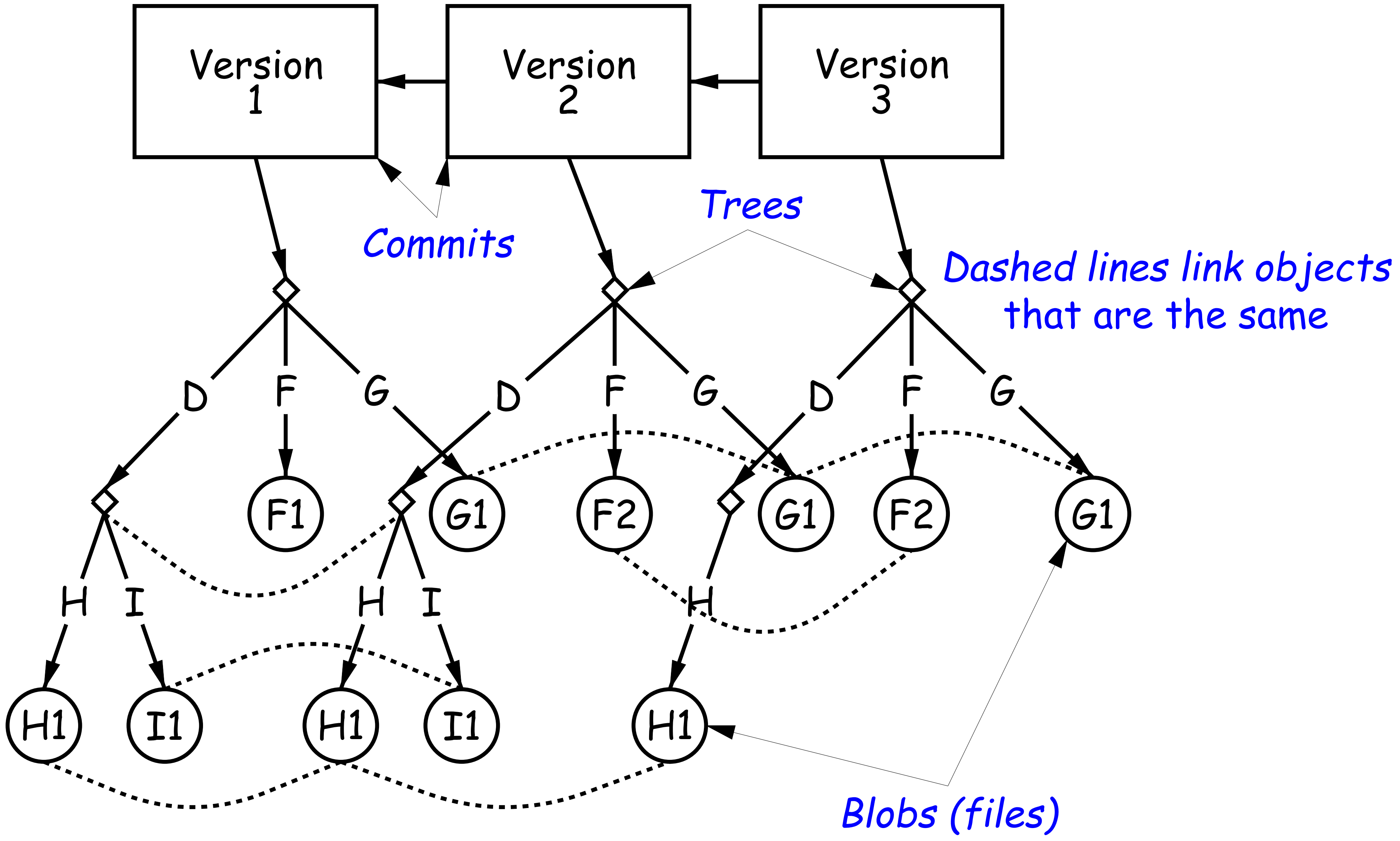
Major User-Level Features (I)

- Abstraction is of a graph of versions or snapshots (called *commits*) of a complete project.
- The graph structure reflects ancestory: which versions came from which.
- Each commit contains
  - A directory tree of files (like a Unix directory).
  - Information about who committed and when.
  - Log message.
  - Pointers to commit (or commits, if there was a merge) from which the commit was derived.

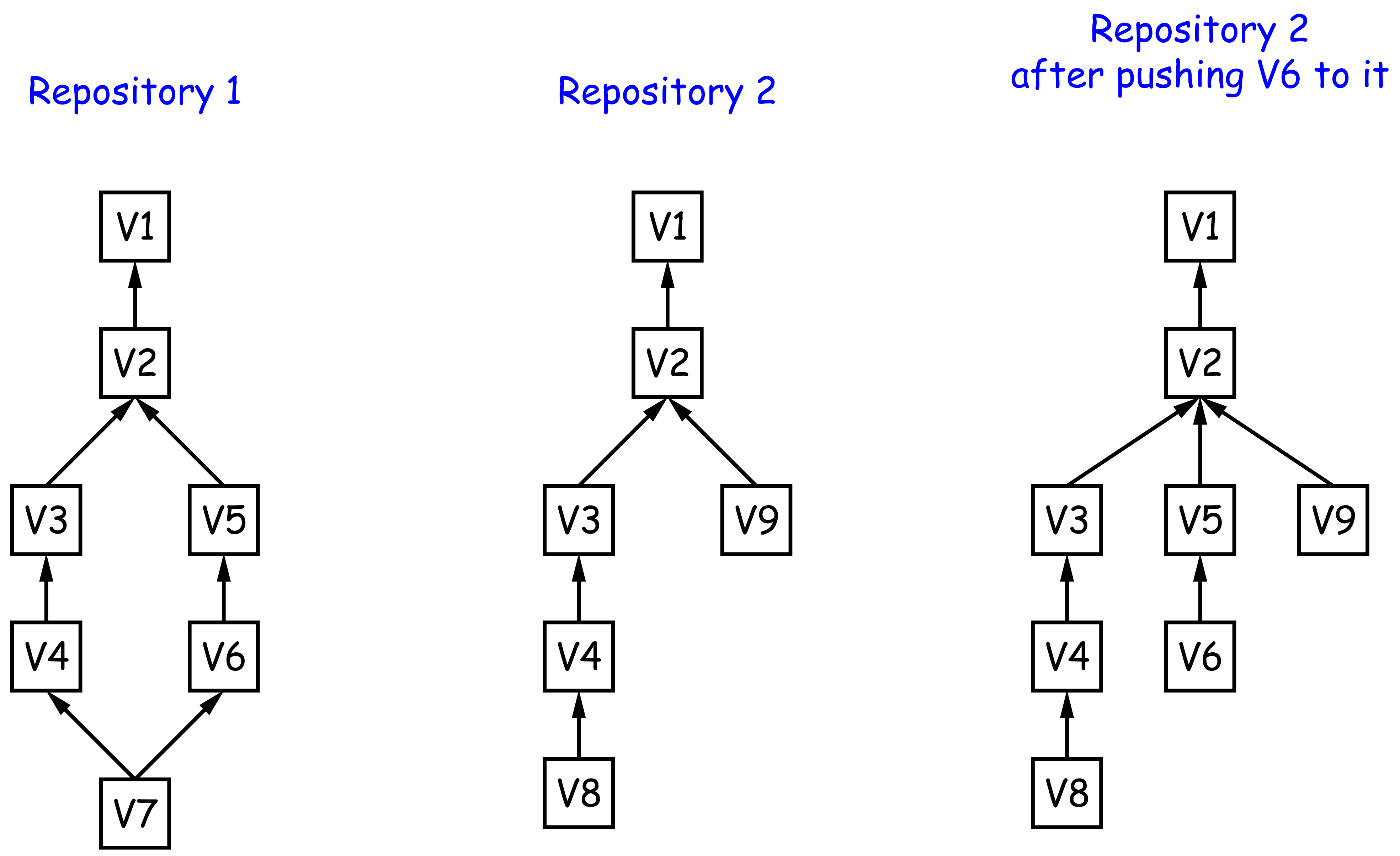
Conceptual Structure

- Main internal components consist of four types of *object:*
  - *Blobs:* basically hold contents of files.
  - *Trees:* directory structures of files.
  - *Commits:* Contain references to trees and additional information (committer, date, log message).
  - *Tags:* References to commits or other objects, with additional information, intended to identify releases, other important versions, or various useful information. (Won't mention further today).

Commits, Trees, Files



Version Histories in Two Repositories



Major User-Level Features (II)

- Each commit has a name that uniquely identifies it to all versions.
- Repositories can transmit collections of versions to each other.
- Transmitting a commit from repository A to repository B requires only the transmission of those objects (files or directory trees) that B does not yet have (allowing speedy updating of repositories).
- Repositories maintain named *branches*, which are simply identifiers of particular commits that are updated to keep track of the most recent commits in various lines of development.
- Likewise, *tags* are essentially named pointers to particular commits. Differ from branches in that they are not usually changed.

Internals

- Each Git repository is contained in a directory.
- Repository may either be *bare* (just a collection of objects and metadata), or may be included as part of a working directory.
- The data of the repository is stored in various *objects* corresponding to files (or other "leaf" content), trees, and commits.
- To save space, data in files is *compressed*.
- Git can *garbage-collect* the objects from time to time to save additional space.

The Pointer Problem

- Objects in Git are files. How should we represent pointers between them?
- Want to be able to *transmit* objects from one repository to another with different contents. How do you transmit the pointers?
- Only want to transfer those objects that are missing in the target repository. How do we know which those are?
- Could use a counter in each repository to give each object there a unique name. But how can that work consistently for two independent repositories?

Content-Addressable File System

- Could use some way of naming objects that is universal.
- We use the names, then, as pointers.
- Solves the "Which objects don't you have?" problem in an obvious way.
- Conceptually, what is invariant about an object, regardless of repository, is its *contents*.
- But can't use the contents as the name for obvious reasons.
- Idea: Use a *hash of the contents* as the address.
- Problem: That doesn't work!
- Brilliant Idea: Use it anyway!!

How A Broken Idea Can Work

- The idea is to use a hash function that is so unlikely to have a collision that we can ignore that possibility.
- *Cryptographic Hash Functions* have relevant property.
- Such a function, f, is designed to withstand cryptoanalytic attacks. In particular, should have
  - *Pre-image resistance:* given $h = f(m)$, should be computationally infeasible to find such a message m.
  - *Second pre-image resistance:* given message m_1, should be infeasible to find $m_2 \neq m_1$ such that $f(m_1) = f(m_2)$.
  - *Collision resistance:* should be difficult to find *any* two messages $m_1 \neq m_2$ such that $f(m_1) = f(m_2)$.
- With these properties, scheme of using hash of contents as name is extremely unlikely to fail, even when system is used maliciously.

SHA1

- Git uses *SHA1* (Secure Hash Function 1).
- Can play around with this using the `hashlib` module in Python3.
- All object names in Git are therefore 160-bit hash codes of contents, in hex.
- E.g. a recent commit in the shared CS61B repository could be fetched (if needed) with

```
git checkout e59849201956766218a3ad6ee1c3aab37dfec3fe
```

CS61B Lecture #36

Today:

- A Brief Side Trip: Enumeration types.
- *DSIJ*, Chapter 10, *HFJ*, pp. 489–516.
  - Threads
  - Communication between threads
  - Synchronization
  - Mailboxes

Side Trip into Java: Enumeration Types

- Problem: Need a type to represent something that has a few, named, discrete values.
- In the purest form, the only necessary operations are `==` and `!=`; the only property of a value of the type is that it differs from all others.
- In older versions of Java, used named integer constants:

```
interface Pieces {
  int BLACK_PIECE = 0,    // Fields in interfaces are static final.
      BLACK_KING = 1,
      WHITE_PIECE = 2,
      WHITE_KING = 3,
      EMPTY = 4;
}
```

- C and C++ provide *enumeration types* as a shorthand, with syntax like this:

```
enum Piece { BLACK_PIECE, BLACK_KING, WHITE_PIECE, WHITE_KING, EMPTY };
```

- But since all these values are basically **ints**, accidents can happen.

Enum Types in Java

- New version of Java allows syntax like that of C or C++, but with more guarantees:

```
public enum Piece {
  BLACK_PIECE, BLACK_KING, WHITE_PIECE, WHITE_KING, EMPTY
}
```

- Defines `Piece` as a new reference type, a special kind of class type.
- The names `BLACK_PIECE`, etc., are static, final *enumeration constants* (or *enumerals*) of type `PIECE`.
- They are automatically initialized, and are the only values of the enumeration type that exist (illegal to use **new** to create an enum value.)
- Can safely use ==, and also `switch` statements:

```
boolean isKing(Piece p) {
  switch (p) {
    case BLACK_KING: case WHITE_KING: return true;
    default: return false;
  }
}
```

Making Enumerals Available Elsewhere

- Enumerals like `BLACK_PIECE` are static members of a class, not classes.
- Therefore, unlike C or C++, their declarations are not automatically visible outside the enumeration class definition.
- So, in other classes, must write `Piece.BLACK_PIECE`, which can get annoying.
- However, with version 1.5, Java has *static imports:* to import all static definitions of class `checkers.Piece` (including enumerals), you write

  ```
  import static checkers.Piece.*;
  ```

  among the import clauses.
- Alas, *cannot* use this for enum classes in the anonymous package.

Operations on Enum Types

- Order of declaration of enumeration constants significant: `.ordinal()` gives the position (numbering from 0) of an enumeration value. Thus, `Piece.BLACK_KING.ordinal()` is 1.
- The array `Piece.values()` gives all the possible values of the type. Thus, you can write:

```
for (Piece p : Piece.values())
    System.out.printf("Piece value #%d is %s%n", p.ordinal(), p);
```

- The static function `Piece.valueOf` converts a String into a value of type `Piece`. So `Piece.valueOf("EMPTY") == EMPTY`.

Fancy Enum Types

- Enums are classes. You can define all the extra fields, methods, and constructors you want.
- Constructors are used only in creating enumeration constants. The constructor arguments follow the constant name:

```
enum Piece {
  BLACK_PIECE(BLACK, false, "b"), BLACK_KING(BLACK, true, "B"),
  WHITE_PIECE(WHITE, false, "w"), WHITE_KING(WHITE, true, "W"),
  EMPTY(null, false, " ");

  private final Side color;
  private final boolean isKing;
  private final String textName;

  Piece(Side color, boolean isKing, String textName) {
    this.color = color; this.isKing = isKing; this.textName = textName;
  }

  Side color() { return color; }
  boolean isKing() { return isKing; }
  String textName() { return textName; }
}
```

Threads

- So far, all our programs consist of single sequence of instructions.
- Each such sequence is called a *thread* (for “thread of control”) in Java.
- Java supports programs containing *multiple* threads, which (conceptually) run concurrently.
- Actually, on a uniprocessor, only one thread at a time actually runs, while others wait, but this is largely invisible.
- To allow program access to threads, Java provides the type `Thread` in `java.lang`. Each `Thread` contains information about, and controls, one thread.
- Simultaneous access to data from two threads can cause chaos, so are also constructs for controlled communication, allowing threads to *lock* objects, to *wait* to be notified of events, and to *interrupt* other threads.

But Why?

- Typical Java programs always have > 1 thread: besides the main program, others clean up garbage objects, receive signals, update the display, other stuff.
- When programs deal with asynchronous events, is sometimes convenient to organize into subprograms, one for each independent, related sequence of events.
- Threads allow us to insulate one such subprogram from another.
- GUIs often organized like this: application is doing some computation or I/O, another thread waits for mouse clicks (like 'Stop'), another pays attention to updating the screen as needed.
- Large servers like search engines may be organized this way, with one thread per request.
- And, of course, sometimes we *do* have a real multiprocessor.

Java Mechanics

- To specify the actions “walking” and “chewing gum”:

```
class Chewer1 implements Runnable {
    public void run()
      { while (true) ChewGum(); }
}
class Walker1 implements Runnable {
    public void run()
      { while (true) Walk(); }
}
```

```
// Walk and chew gum
Thread chomp
  = new Thread(new
Chewer1());
Thread clomp
  = new Thread(new
Walker1());
chomp.start(); clomp.start();
```

- Concise Alternative (uses fact that Thread implements Runnable):

```
class Chewer2 extends Thread {
  public void run()
    { while (true) ChewGum(); }
}
class Walker2 extends Thread {
  public void run()
    { while (true) Walk(); }
}
```

```
Thread chomp = new Chewer2(),
       clomp = new Walker2();
chomp.start();
clomp.start();
```

Avoiding Interference

- When one thread has data for another, one must wait for the other to be ready.
- Likewise, if two threads use the same data structure, generally only one should modify it at a time; other must wait.
- E.g., what would happen if two threads simultaneously inserted an item into a linked list at the same point in the list?
- A: Both could conceivably execute

  ```
  p.next = new ListCell(x, p.next);
  ```

  with the *same* values of `p` and `p.next`; one insertion is lost.
- Can arrange for only one thread at a time to execute a method on a particular object with either of the following equivalent definitions:

```
void f(...) {
  synchronized (this) {
    body of f
  }
}
```

```
synchronized void f(...) {
  body of f
}
```

Communicating the Hard Way

- Communicating data is tricky: the faster party must wait for the slower.
- Obvious approaches for sending data from thread to thread don't work:

```
class DataExchanger {
   Object value = null;
   Object receive() {
      Object r; r = null;
      while (r == null)
        { r = value; }
      value = null;
      return r;
   }
   void deposit(Object data) {
      while (value != null) { }
      value = data;
   }
}
```

```
DataExchanger exchanger
  = new DataExchanger();

--------------------------------

// thread1 sends to thread2 with
exchanger.deposit("Hello!");

--------------------------------

// thread2 receives from thread1 with
msg = (String) exchanger.receive();
```

- BAD: One thread can monopolize machine while waiting; two threads executing `deposit` or `receive` simultaneously cause chaos.

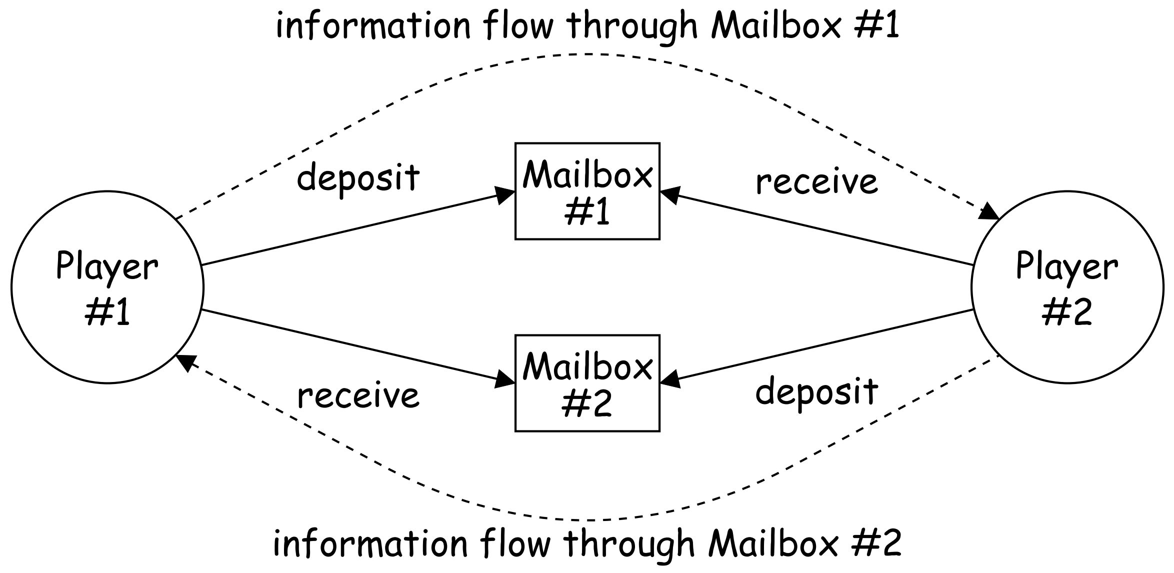
Primitive Java Facilities

- `wait` method on Object makes thread wait (not using processor) until notified by `notifyAll`, unlocking the Object while it waits.
- Example, `ucb.util.mailbox` has something like this (simplified):

```
interface Mailbox {
  void deposit(Object msg) throws InterruptedException;
  Object receive() throws InterruptedException;
}

class QueuedMailbox implements Mailbox {
  private List<Object> queue = new LinkedList<Object>();

  public synchronized void deposit(Object msg) {
    queue.add(msg);
    this.notifyAll(); // Wake any waiting receivers
  }

  public synchronized Object receive() throws InterruptedException {
    while (queue.isEmpty()) wait();
    return queue.remove(0);
  }
}
```

Message-Passing Style

- Use of Java primitives very error-prone. Wait until CS162.
- Mailboxes are higher-level, and allow the following program structure:


information flow through Mailbox #1
deposit
Mailbox #1
receive
Player #1
Player #2
receive
Mailbox #2
deposit
information flow through Mailbox #2

- Where each Player is a thread that looks like this:

```
while (! gameOver()) {
  if (myMove())
    outBox.deposit(computeMyMove(lastMove));
  else
    lastMove = inBox.receive();
}
```

More Concurrency

- Previous example can be done other ways, but mechanism is very flexible.
- E.g., suppose you want to think during opponent's move:

```
while (!gameOver()) {
  if (myMove())
    outBox.deposit(computeMyMove(lastMove));
  else {
    do {
      thinkAheadALittle();
      lastMove = inBox.receiveIfPossible();
    } while (lastMove == null);
}
```

- `receiveIfPossible` (written `receive(0)` in our actual package) doesn't wait; returns `null` if no message yet, perhaps like this:

```
public synchronized Object receiveIfPossible()
   throws InterruptedException {
  if (queue.isEmpty())
    return null;
  return queue.remove(0);
}
```

Coroutines

- A *coroutine* is a kind of synchronous thread that explicitly hands off control to other coroutines so that only one executes at a time, like Python generators. Can get similar effect with threads and mailboxes.
- Example: recursive inorder tree iterator:

```
class TreeIterator extends Thread {
  Tree root; Mailbox r;
  TreeIterator(Tree T, Mailbox r) {
    this.root = T; this.dest = r;
  }
  public void run() {
    traverse(root);
    r.deposit(End marker);
  }
  void traverse(Tree t) {
    if (t == null) return;
    traverse(t.left);
    r.deposit(t.label);
    traverse(t.right);
  }
}
```

```
void treeProcessor(Tree T) {
  Mailbox m = new QueuedMailbox();
  new TreeIterator(T, m).start();
  while (true) {
     Object x = m.receive();
     if (x is end marker)
        break;
     do something with x;
  }
}
```

Use In GUIs

- Jave runtime library uses a special thread that does nothing but wait for *events* like mouse clicks, pressed keys, mouse movement, etc.
- You can designate an object of your choice as a *listener;* which means that Java's event thread calls a method of that object whenever an event occurs.
- As a result, your program can do work while the GUI continues to respond to buttons, menus, etc.
- Another special thread does all the drawing. You don't have to be aware when this takes place; just ask that the thread wake up whenever you change something.

Highlights of a GUI Component

```java
/** A widget that draws multi-colored lines indicated by mouse. */
class Lines extends JComponent implements MouseListener {
  private List<Point> lines = new ArrayList<Point>();

  Lines() {  // Main thread calls this to create one
    setPreferredSize(new Dimension(400, 400));
    addMouseListener(this);
  }
  public synchronized void paintComponent(Graphics g) { // Paint thread
    g.setColor(Color.white);   g.fillRect(0, 0, 400, 400);
    int x, y;   x = y = 200;
    Color c = Color.black;
    for (Point p : lines)
      g.setColor(c); c = chooseNextColor(c);
      g.drawLine(x, y, p.x, p.y); x = p.x; y = p.y;
    }
  }
  public synchronized void mouseClicked(MouseEvent e) // Event thread
    { lines.add(new Point(e.getX(), e.getY())); repaint(); }
  ...
}
```

Interrupts

- An *interrupt* is an event that disrupts the normal flow of control of a program.
- In many systems, interrupts can be totally *asynchronous*, occurring at arbitrary points in a program, the Java developers considered this unwise; arranged that interrupts would occur only at controlled points.
- In Java programs, one thread can interrupt another to inform it that something unusual needs attention:

```
otherThread.interrupt();
```

- But `otherThread` does not receive the interrupt until it waits: methods `wait`, `sleep` (wait for a period of time), `join` (wait for thread to terminate), and mailbox deposit and receive.
- Interrupt causes these methods to throw `InterruptedException`, so typical use is like this:

```
try {
  msg = inBox.receive();
} catch (InterruptedException e) { HandleEmergency(); }
```

Remote Mailboxes (A Side Excursion)

- RMI: Remote Method Interface allows one program to refer to objects in another program.
- We use it to allow mailboxes in one program be received from or deposited into in another.
- To use this, you define an *interface* to the remote object:

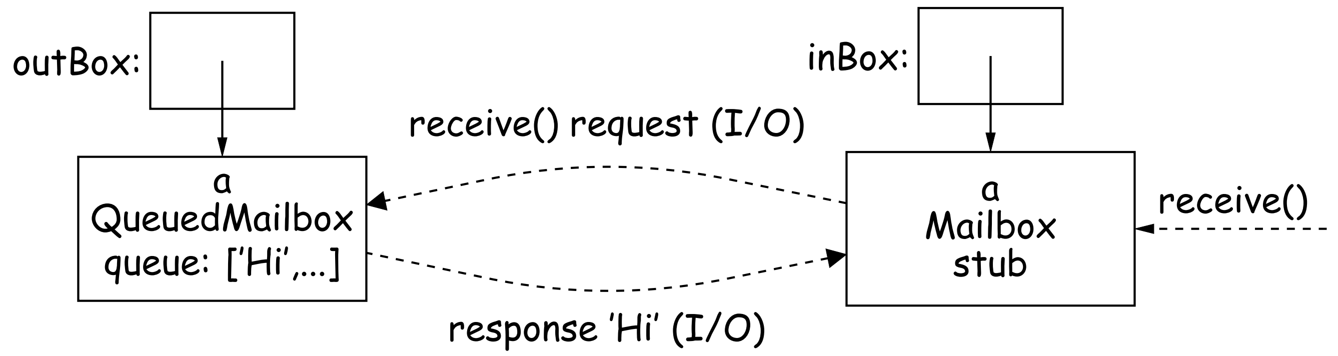
```
import java.rmi.*;
interface Mailbox extends Remote {
  void deposit(Object msg)
    throws InterruptedException, RemoteException;
  Object receive()
    throws InterruptedException, RemoteException;
  ...
}
```

- On machine that actually will contain the object, you define

```
class QueuedMailbox ... implements Mailbox {
    Same implementation as before, roughly
}
```

Remote Objects Under the Hood

```
// On machine #1:                  // On Machine #2:
Mailbox outBox                      Mailbox inBox
  = new QueuedMailbox();             = get outBox from machine #1
```



- Because Mailbox is an interface, hides fact that on Machine #2 doesn't actually have direct access to it.
- Requests for method calls are relayed by I/O to machine that has real object.
- Any argument or return type OK if it also implements Remote or can be *serialized*—turned into stream of bytes and back, as can primitive types and String.
- Because I/O involved, expect failures, hence every method can throw RemoteException (subtype of IOException).

Lecture #37

Today: A little side excursion into nitty-gritty stuff: Storage management.

Scope and Lifetime

- *Scope* of a declaration is portion of program text to which it applies (is *visible*).
  - Need not be contiguous.
  - In Java, is static: independent of data.
- *Lifetime* or *extent* of storage is portion of program execution during which it exists.
  - Always contiguous
  - Generally dynamic: depends on data
- Classes of extent:
  - *Static:* entire duration of program
  - *Local* or *automatic:* duration of call or block execution (local variable)
  - *Dynamic:* From time of allocation statement (**new**) to deallocation, if any.

Explicit vs. Automatic Freeing

- Java has no explicit means to free dynamic storage.
- However, when no expression in any thread can possibly be influenced by or change an object, it might as well not exist:

```
IntList wasteful()
{
  IntList c = new IntList(3, new IntList(4, null));
  return c.tail;
  // variable c now deallocated, so no way
  // to get to first cell of list
}
```

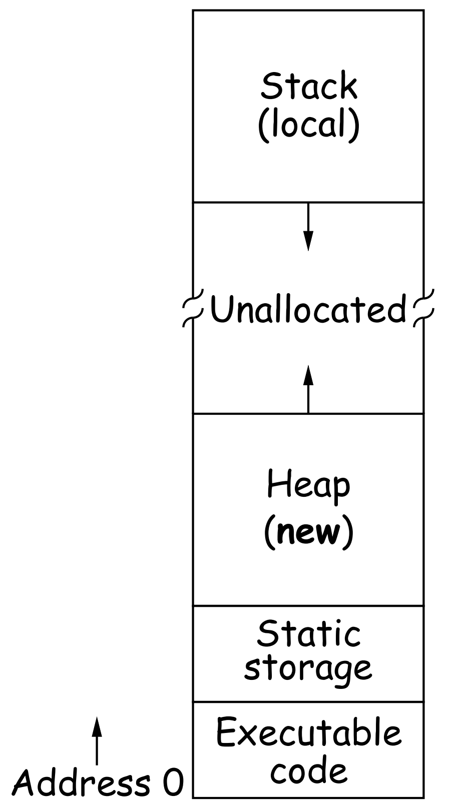
- At this point, Java runtime, like Scheme's, recycles the object `c` pointed to: *garbage collection*.

Under the Hood: Allocation

- Java pointers (references) are represented as integer addresses.
- Corresponds to machine's own practice.
- In Java, cannot convert integers $\leftrightarrow$ pointers,
- But crucial parts of Java runtime implemented in C, or sometimes machine code, where you can.
- Crude allocator in C:

```
char store[STORAGE_SIZE];  // Allocated array
size_t remainder = STORAGE_SIZE;

/** A pointer to a block of at least N bytes of storage */
void* simpleAlloc(size_t n) { // void*: pointer to anything
  if (n > remainder) ERROR();
  remainder = (remainder - n) & ~0x7;  // Make multiple of 8
  return (void*) (store + remainder);
}
```

Example of Storage Layout: Unix



- OS gives way to turn chunks of unallocated region into heap.
- Happens automatically for stack.

Explicit Deallocating

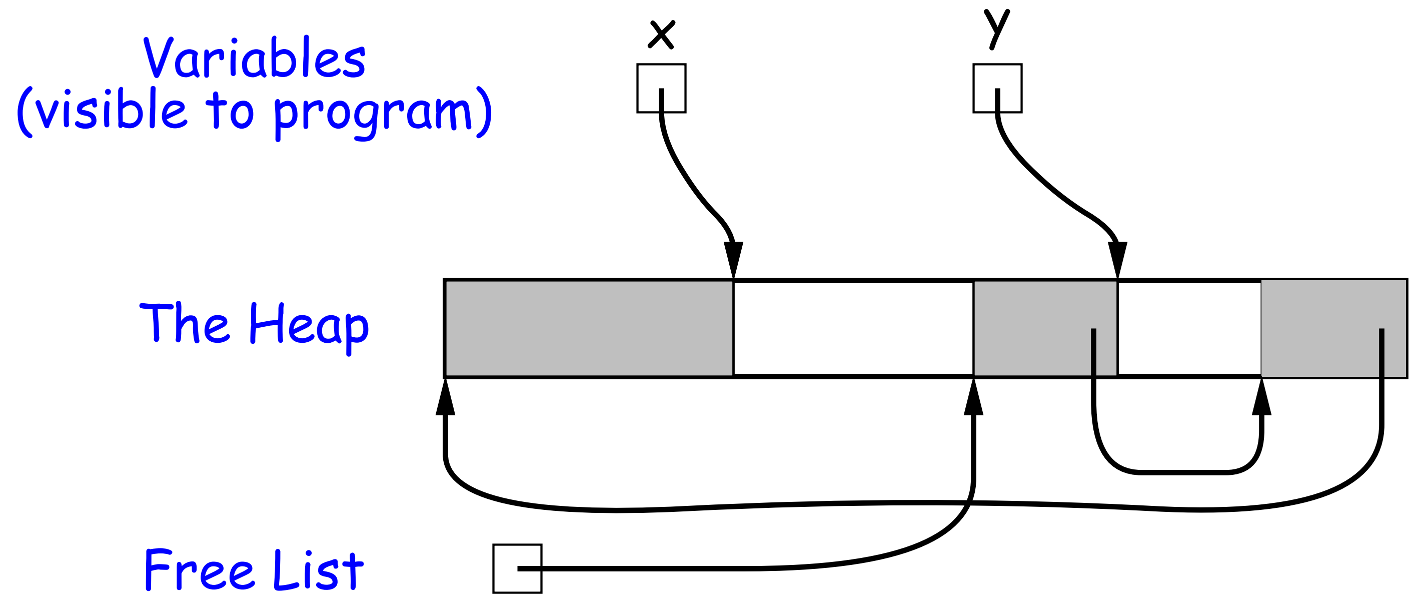
- C/C++ normally require explicit deallocation, because of
  - Lack of run-time information about what is array
  - Possibility of converting pointers to integers.
  - Lack of run-time information about *unions:*

```
union Various {
  int Int;
  char* Pntr;
  double Double;
} X;  // X is either an int, char*, or double
```

- Java avoids all three problems; automatic collection possible.
- Explicit freeing can be somewhat faster, but rather error-prone:
  - Memory corruption
  - Memory leaks

Free Lists

- Explicit allocator grabs chunks of storage from OS and gives to applications.
- Or gives recycled storage, when available.
- When storage is freed, added to a *free list* data structure to be recycled.
- Used both for explicit freeing and some kinds of automatic garbage collection.

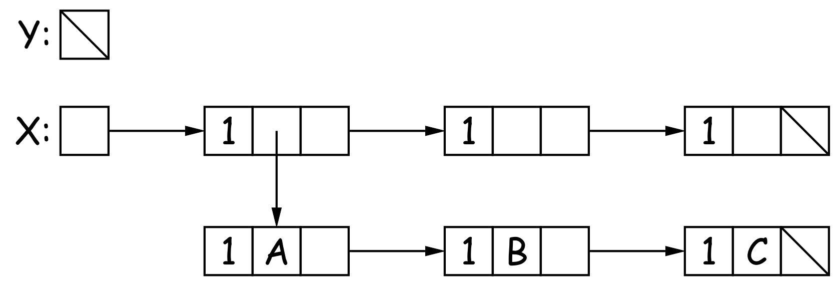


Free List Strategies

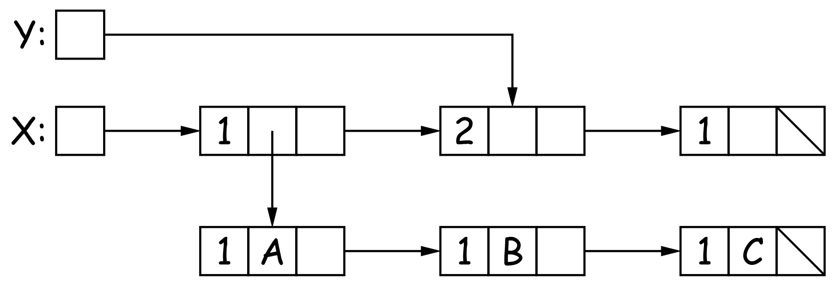
- Memory requests generally come in multiple sizes.
- Not all chunks on the free list are big enough, and one may have to search for a chunk and break it up if too big.
- Various strategies to find a chunk that fits have been used:
  - *Sequential fits:*
    - Link blocks in LIFO or FIFO order, or sorted by address.
    - Coalesce adjacent blocks.
    - Search for *first fit* on list, *best fit* on list, or *next fit* on list after last-chosen chunk.
  - *Segregated fits:* separate free lists for different chunk sizes.
  - *Buddy systems:* A kind of segregated fit where some newly adjacent free blocks of one size are easily detected and combined into bigger chunks.
- Coalescing blocks reduces *fragmentation* of memory into lots of little scattered chunks.

Garbage Collection: Reference Counting

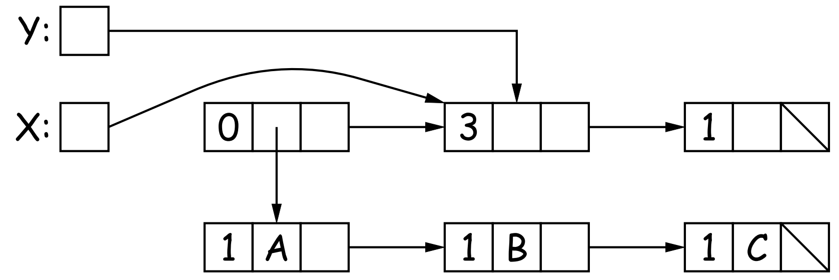
- Idea: Keep count of number of pointers to each object. Release when count goes to 0.

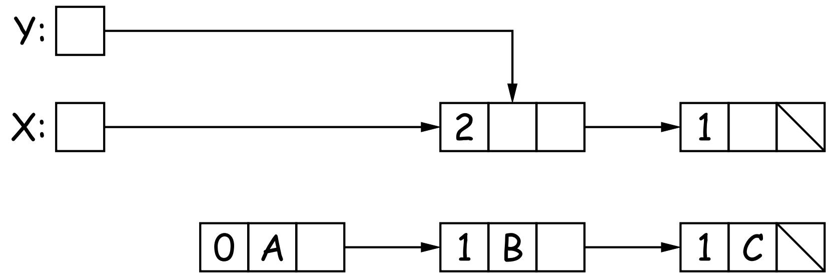


`Y = X.tail;`

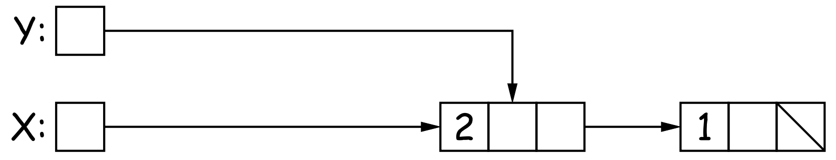


`X = Y;`



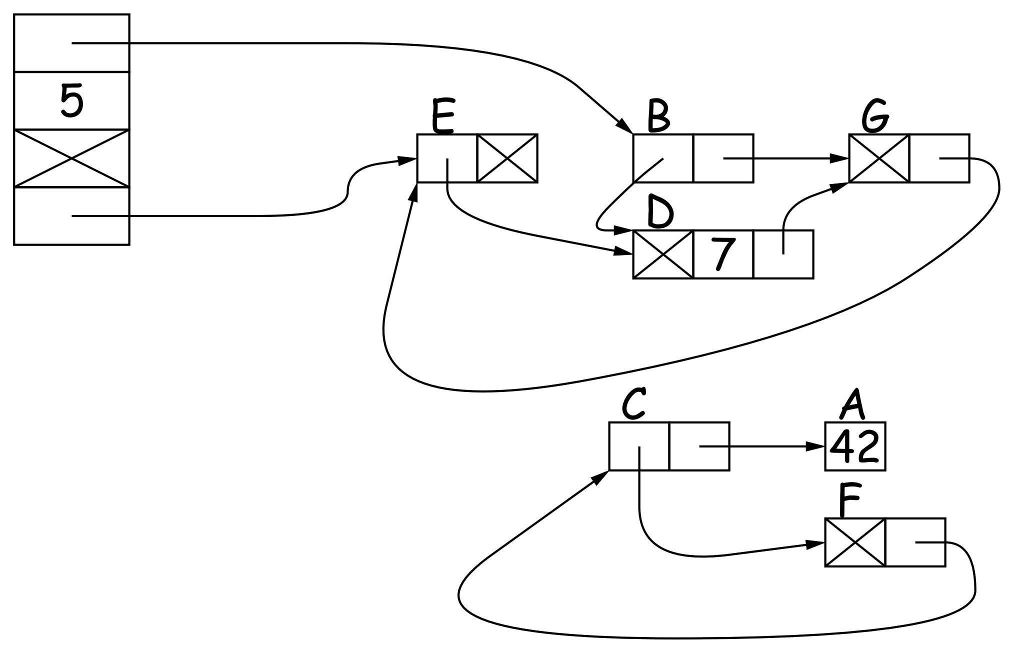


. . . etc., until:

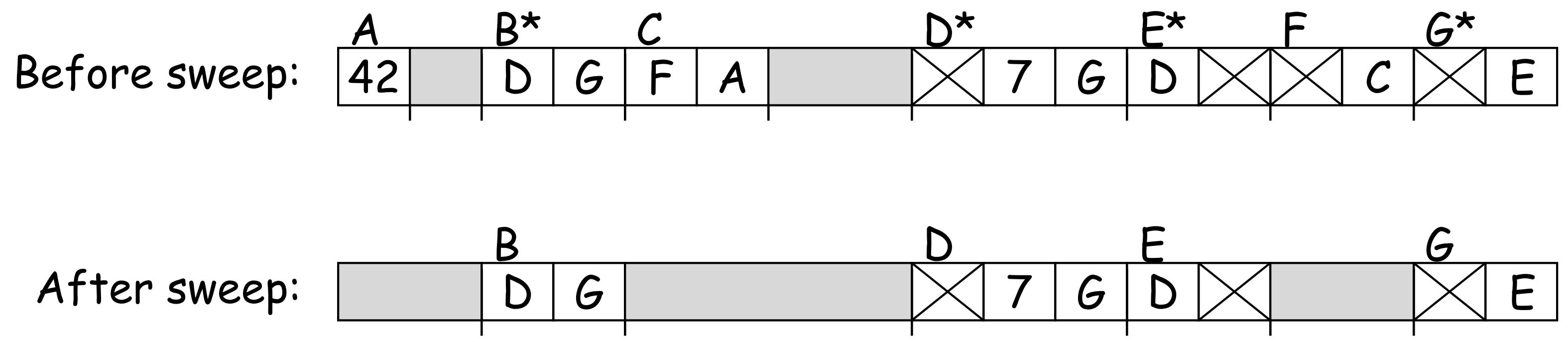


Garbage Collection: Mark and Sweep

Roots (locals + statics)



1. Traverse and mark graph of objects.
2. Sweep through memory, freeing unmarked objects.



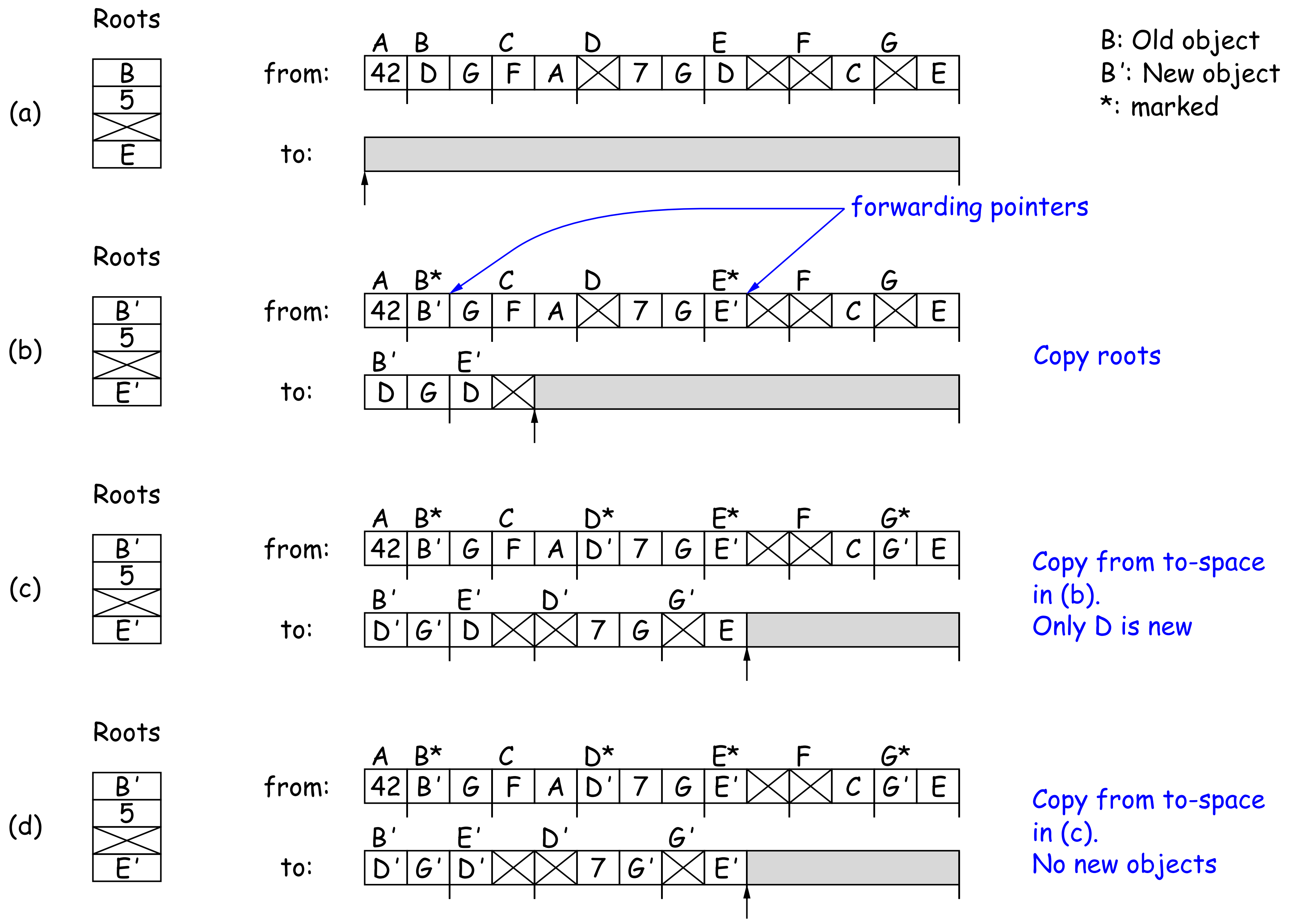
Cost of Mark-and-Sweep

- Mark-and-sweep algorithms don't move any exisiting objects—pointers stay the same.
- The total amount of work depends on the amount of memory swept—i.e., the total amount of active (non-garbage) storage + amount of garbage. Not necessarily a big hit: the garbage had to be active at one time, and hence there was always some "good" processing in the past for each byte of garbage scanned.

Copying Garbage Collection

- Another approach: *copying garbage collection* takes time proportional to amount of active storage:
  - Traverse the graph of active objects breadth first, *copying* them into a large contiguous area (called "to-space").
  - As you copy each object, mark it and put a *forwarding pointer* into it that points to where you copied it.
  - The next time you have to copy an already marked object, just use its forwarding pointer instead.
  - When done, the space you copied from ("from-space") becomes the next to-space; in effect, all its objects are freed in constant time.

Copying Garbage Collection Illustrated



Most Objects Die Young: Generational Collection

- Most older objects stay active, and need not be collected.
- Would be nice to avoid copying them over and over.
- *Generational garbage collection* schemes have two (or more) from spaces: one for newly created objects (*new space*) and one for "tenured" objects that have survived garbage collection (*old space*).
- A typical garbage collection collects only in new space, ignores pointers from new to old space, and moves objects to old space.
- As roots, uses usual roots plus pointers in old space that have changed (so that they might be pointing to new space).
- When old space full, collect all spaces.
- This approach leads to much smaller *pause times* in interactive systems.

There’s Much More

- These are just highlights.
- Lots of work on how to implement these ideas efficiently.
- *Distributed garbage collection:* What if objects scattered over many machines?
- *Real-time collection:* where predictable pause times are important, leads to *incremental* collection, doing a little at a time.

Lecture #38: Compression

Announcements

- HKN surveys Friday in class. Extra points awarded to those who participate!

Compression and Git

- Git creates a new object in the repository each time a changed file or directory is committed.
- Things can get crowded as a result.
- To save space, it *compresses* each object.
- Every now and then (such as when sending or receiving from another repository), it packs objects together into a single file: a "packfile."
- Besides just sticking the files together, uses a technique called *delta compression.*

Delta Compression

- Typically, there will be many versions of a file in a Git repository: the latest, and previous edits of it, each in different commits.
- Git doesn't keep track explicitly of which file came from where, since that's hard in general:
  - What if a file is split into two, or two are spliced together?
- But, can guess that files with same name and (roughly) same size in two commits are probably versions of the same file.
- When that happens, store one of them as a pointer to the other, plus a list of changes.

Delta Compression (II)

- So, store two versions

V1	V2
My eyes are fully open to my awful situation. I shall go at once to Roderick and make him an oration. I shall tell him I've recovered my forgotten moral senses,	My eyes are fully open to my awful situation. I shall go at once to Roderick and make him an oration. I shall tell him I've recovered my forgotten moral senses, and don't give twopence halfpenney for any consequences.

as

V1	V2
[Fetch 1st 6 lines from V2]	My eyes are fully open to my awful situation. I shall go at once to Roderick and make him an oration. I shall tell him I've recovered my forgotten moral senses, and don't give twopence halfpenney for any consequences.

Compression Techniques

Slides from Josh Hug

LZ77 and DEFLATE

- Git Actually uses a different scheme from LZW for compression: a combination of LZ77 and Huffman coding.
- LZ77 is kind of like delta compression, but within the same text.
- Convert a text such as

  ```
  One Mississippi, two Mississippi
  ```

  into something like

  ```
  One Mississippi, two <11,7>
  ```

  where the `<11,7>` is intended to mean "the next 11 characters come from the text that ends 7 characters before this point."
- We add new symbols to the alphabet to represent these (length, distance) inclusions.
- When done, Huffman encode the result.

Announcements

- Lab sections on December 5, 6, and 7 will be organized as follows: Students will work on an exam-like set of exercises covering linked lists, stacks, queues, binary trees, binary search trees. Solutions will be thoroughly reviewed. 1 bonus point (out of 200) for completing the exercises.
- Please use git-bug for problems with submission, your code, the skeleton, or any of our software.
- **Tutors and lab assistants needed.** Consider volunteering to be a tutor or lab assistant for CS 10, self-paced courses, CS 61A, or CS 61B next semester.
- **Programming Contest:** Visit my web page for information about the annual programming contest, which we hold each fall. There are large collections of programming problems you can try your hand on.

Lecture #40: Course Summary

- Programming language: Java
- Program Analysis
- Categories of data structure: Java library structure
- Sequences
- Trees
- Searching
- Sorting
- Pseudo-random numbers
- Graphs
- Pragmatic implementation topics

Programming-Language Topics

- Object-based programming: organizing around data types
- Object-oriented programming:
  - Dynamic vs. static type
  - Inheritance
  - Idea of interface vs. implementation
- Generic programming (the <· · ·> stuff).
- Memory model: containers, pointers, arrays
- Numeric types
- Java syntax and semantics
- Scope and extent
- Standard idioms, patterns:
  - Objects used as functions (e.g., Comparator)
  - Partial implementations (e.g., AbstractList)
  - Iterators
  - Views (e.g., sublists)

Analysis and Algorithmic Techniques

- Asymptotic analysis
- $O(\cdot)$, $o(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$ notations
- Worst case, average case.
- Amortized time
- Memoization and dynamic programming.

Major Categories of Data Structure

- Collection interface and its subtypes
- Map interface and its subtypes
- Generic skeleton implementations of collections, lists, maps (`AbstractList`, etc.)
- Complete concrete collection and map classes in Java library

Sequences

- Linking:
  - Single and double link manipulations
  - Sentinels
- Linking vs. arrays
- Stacks, queues, deques
- Circular buffering
- Trade-offs: costs of basic operations

Trees

- Uses of trees: search, representing hierarchical structures
- Basic operations: insertion, deletion
- Tree traversals
- Representing trees
- Game trees

Searching

- Search trees, range searching
- Multidimensional searches: quad trees.
- Hashing
- Priority queues and heaps
- Balanced trees
  - Rebalancing by rotation (red-black trees)
  - Balance by construction (B-trees)
  - Probabilistic balance (skip lists)
  - Tries
- Search times, trade-offs

Sorting

- Uses of sorting
- Insertion sort
- Selection sorting
- Merge sort
- Heap sort
- Quicksort and selection
- Distribution sort
- Radix sort
- Complexity of various algorithms, when to use them?

Random numbers

- Possible uses
- Idea of a pseudo-random sequence
- Linear congruential and additive generators
- Changing distributions:
  - Changing the range
  - Non-uniform distributions
- Shuffling, random selection

Graph structures

- Definition
- Uses: things represented by graphs
- Graph traversal: the generic traversal template
- Depth-first traversal, breadth-first traversal
- Topological sort
- Shortest paths
- Minimal spanning trees, union-find structures
- Memory management as a graph problem.

Debugging

- What debuggers can do
- How to use to pin down bugs
- Details of some debugger (Eclipse, gjdb, various Windows/Sun products).
- Unit testing: what it means, how to use it.
- JUnit mechanics.

Version Control

- What's it for?
- Basic concepts behind our particular system:
  - Working copy vs. repository copy
  - Committing changes
  - Updating and merging changes.
  - Tagging

A Case Study

- Presented Git version-control system as an example of a design using several ideas from this course.
- Graph (DAG) and tree structures represented with files as vertices and strings (file names), rather than machine addresses, as pointers.
- Use of hashing to create unique (or very, very likely to be unique) names: *probabilistic data structure*.
- Compression uses various kinds of map to facilitate conversion to and from compressed form, including arrays, tries, and hash tables
- Priority queue in Huffman coding.

Assorted Side Trips

- Compression.
- Parallel processing.
- Storage management and garbage collection.

What’s After the Lower Division?

- CS160: User Interface Design (Hartmann)
- CS161: Computer Security (Popa)
- CS162: Operating Systems and System Programming (Joseph, Ragan-Kelley)
- CS164: Programming Languages and Compilers (Hilfinger)
- CS170: Efficient Algorithms and Intractable Problems (Chiesa, Vazirani)
- CS174: Combinatorics and Discrete Probability (Friedman)
- CS184: Graphics (Ng)
- CS186: Databases
- CS188: Artificial Intelligence (Dragan, Levine)
- CS189: Machine Learning
- CS194: Assorted Special Topics: Computational Design and Fabrication, Designing, Visualizing and Understanding Deep Neural Networks.

What's After the Lower Division? (II)

- CS152: Computer Architecture (Asanovic)

ternet

ıeering

3iology

Design

- Numerous graduate courses: including advanced versions of 152, 160, 161 170, 184, 186, 189; plus Cryptography, VLSI design and many special topics.
- And, of course, EE courses!
- Various opportunities for participating in research and independent study (199)

What's After the Lower Division? (III)

- But EE and CS are just two of over 150 subjects!
- Internships offer more specific skills and exposure to real problems.
- Above all, I think that CS is a creative activity that (to the true artists) ought to fun!