Computer Science E-22
Data Structures

Harvard Extension School, Fall 2016
David G. Sullivan, Ph.D.

Introduction: Abstract Data Types and Java Review

Computer Science E-22
Harvard Extension School

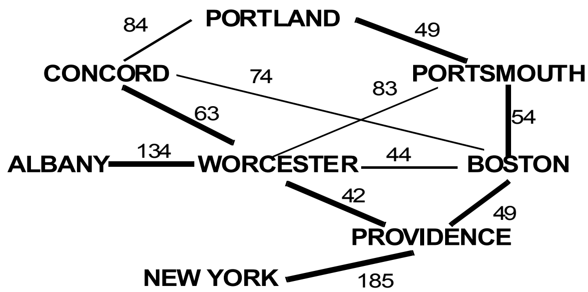
David G. Sullivan, Ph.D.

Welcome to Computer Science E-22!

- We will study fundamental *data structures.*
  - ways of imposing order on a collection of information
  - sequences: lists, stacks, and queues
  - trees
  - hash tables
  - graphs
- We will also:
  - study *algorithms* related to these data structures
  - learn how to *compare* data structures & algorithms
- Goals:
  - learn to think more intelligently about programming problems
  - acquire a set of useful tools and techniques

Sample Problem I: Finding Shortest Paths

- Given a set of routes between pairs of cities, determine the shortest path from city A to city B.



Sample Problem II: A Data "Dictionary"

- Given a large collection of data, how can we arrange it so that we can efficiently:
  - add a new item
  - search for an existing item
- Some data structures provide better performance than others for this application.
- More generally, we'll learn how to characterize the *efficiency* of different data structures and their associated algorithms.

Prerequisites

- A good working knowledge of Java
  - comfortable with object-oriented programming concepts
  - comfortable with arrays
  - some prior exposure to recursion would be helpful
  - if your skills are weak or rusty, you may want to consider first taking CSCI E-10b
- Reasonable comfort level with mathematical reasoning
  - mostly simple algebra, but need to understand the basics of logarithms (we'll review this)
  - will do some simple proofs

Requirements

- Lectures and weekly sections
  - sections: start next week; times and locations TBA
  - also available by streaming and recorded video
- Five problem sets
  - plan on 10-20 hours per week!
  - code in Java
  - must be your own work
  - grad-credit students will do extra problems
- Midterm exam
- Final exam
- Programming project
  - for grad credit only

Additional Administrivia

- Instructor: Dave Sullivan
- TAs: Alex Breen, Cody Doucette, Kylie Moses
- Office hours and contact info. will be available on the Web:
  http://sites.fas.harvard.edu/~cscie22
- For questions on content, homework, etc.:
  - use Piazza
  - send e-mail to cscie22@fas.harvard.edu

Review: What is an Object?

- An *object* groups together:
  - one or more data values (the object's *fields* – also known as *instance variables*)
  - a set of operations that the object can perform (the object's *methods*)
- In Java, we use a *class* to define a new type of object.
  - serves as a "blueprint" for objects of that type
  - simple example:

```
public class Rectangle {
    // fields
    private int width;
    private int height;

    // methods
    public int area() {
        return width * height;
    }
    …
```

Class vs. Object

- The `Rectangle` class is a blueprint:

```
public class Rectangle {
    // fields
    private int width;
    private int height;

    // methods
    ...
}
```

- `Rectangle` objects are built according to that blueprint:

(You can also think of the methods as being inside the object, but we won't show them in our diagrams.)

Creating and Using an Object

- We create an object by using the `new` operator and a special method known as a *constructor*:

```
Rectangle r1 = new Rectangle(10, 30);
```

- Once an object is created, we can call one of its methods by using *dot notation*:

```
int a1 = r1.area();
```

- The object on which the method is invoked is known as the *called object* or the *current object.*

Two Types of Methods

- Methods that belong to an object are referred to as *instance methods* or *non-static methods.*
  - they are invoked on an object
    ```
    int a1 = r1.area();
    ```
  - they have access to the fields of the called object
- *Static* methods do *not* belong to an object – they belong to the class as a whole.
  - they have the keyword static in their header:
    ```
    public static int max(int num1, int num2) {
        …
    ```
  - they do *not* have access to the fields of the class
  - outside the class, they are invoked using the class name:
    ```
    int result = Math.max(5, 10);
    ```

Abstract Data Types

- An *abstract data type* (ADT) is a model of a data structure that specifies:
  - the characteristics of the collection of data
  - the operations that can be performed on the collection
- It's *abstract* because it doesn't specify *how* the ADT will be implemented.
- A given ADT can have multiple implementations.

A Simple ADT: A Bag

- A bag is just a container for a group of data items.
  - analogy: a bag of candy
- The positions of the data items don't matter (unlike a list).
  - {3, 2, 10, 6} is equivalent to {2, 3, 6, 10}
- The items do *not* need to be unique (unlike a set).
  - {7, 2, 10, 7, 5} isn't a set, but it is a bag

A Simple ADT: A Bag (cont.)

- The operations supported by our Bag ADT:
  - `add(item)`: add `item` to the `Bag`
  - `remove(item)`: remove one occurrence of `item` (if any) from the `Bag`
  - `contains(item)`: check if `item` is in the `Bag`
  - `numItems()`: get the number of items in the `Bag`
  - `grab()`: get an item at random, without removing it
    - reflects the fact that the items don't have a position (and thus we can't say "get the 5^{th} item in the Bag")
  - `toArray()`: get an array containing the current contents of the bag
- Note that we *don't* specify *how* the bag will be implemented.

Specifying an ADT Using an Interface

- In Java, we can use an interface to specify an ADT:

```
public interface Bag {
    boolean add(Object item);
    boolean remove(Object item);
    boolean contains(Object item);
    int numItems();
    Object grab();
    Object[] toArray();
}
```

- An interface specifies a set of methods.
  - includes only the method headers
  - *cannot* include the actual method definitions

Implementing an ADT Using a Class

- To implement an ADT, we define a class:

```
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
    …
    public boolean add(Object item) {
        …
}
```

- When a class header includes an `implements` clause, the class must define all of the methods in the interface.

Encapsulation

- Our implementation provides proper *encapsulation*.
  - a key principle of object-oriented programming
  - also known as *information hiding*

- We prevent direct access to the internals of an object by making its fields *private.*

```
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
    …
```

- We provide limited *indirect* access through methods that are labeled *public.*

```
    public boolean add(Object item) {
```

All Interface Methods Are Public

- Methods specified in an interface *must* be `public`, so we don't need to use the keyword `public` in the interface definition.

- For example:

```
public interface Bag {
    boolean add(Object item);
    boolean remove(Object item);
    boolean contains(Object item);
    int numItems();
    Object grab();
    Object[] toArray();
}
```

- However, when we actually implement one of these methods in a class, we *do* need to explicitly use the keyword `public`:

```
public class ArrayBag implements Bag {
    …
    public boolean add(Object item) {
        …
```

Inheritance

- We can define a class that explicitly *extends* another class:

```
public class Animal {
    private String name;
    …
    public String getName() {
        return name;
    }
    …
}
public class Dog extends Animal {
    …
```

- We say that `Dog` is a *subclass* of `Animal`, and `Animal` is a *superclass* of `Dog`.

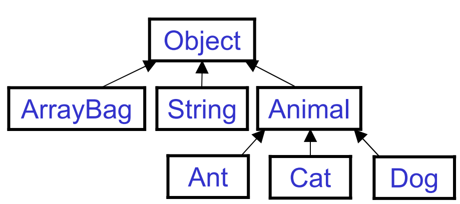
- A class *inherits* the instance variables and methods of the class that it extends.

The `Object` Class

- If a class does not explicitly extend another class, it implicitly extends Java's `Object` class.

- The `Object` class includes methods that all classes must possess. For example:
  - `toString()`: returns a string representation of the object
  - `equals()`: is this object equal to another object?

- The process of extending classes forms a hierarchy of classes, with the `Object` class at the top of the hierarchy:



Polymorphism

- An object can be used wherever an object of one of its superclasses is called for.
- For example:

```
Animal a = new Dog();

Animal[] zoo = new Animal[100];
zoo[0] = new Ant();
zoo[1] = new Cat();
…
```

- The name for this capability is *polymorphism.*
  - from the Greek for "many forms"
  - the same code can be used with objects of different types

Storing Items in an `ArrayBag`

- We store the items in an array of type `Object`.

```
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
    …
}
```

- This allows us to store *any* type of object in the `items` array, thanks to the power of polymorphism:

```
ArrayBag bag = new ArrayBag();
bag.add("hello");
bag.add(new Double(3.1416));
```

Another Example of Polymorphism

- An interface name can be used as the type of a variable.

```
Bag b;
```

- Variables that have an interface type can hold references to objects of any class that implements the interface.

```
Bag b = new ArrayBag();
```

- Using a variable that has the interface as its type allows us to write code that works with any implementation of an ADT.

```
public void processBag(Bag b) {
    for (int i = 0; i < b.numItems(); i++) {
      …
}
```

  - the param can be an instance of *any* `Bag` implementation
  - we must use method calls to access the object's internals, because we can't know for certain what the field names are

Memory Management: Looking Under the Hood

- In order to understand the implementation of the data structures we'll cover in this course, you'll need to have a good understanding of how memory is managed.
- There are three main types of memory allocation in Java.
- They correspond to three different regions of memory.

Memory Management, Type I: Static Storage

- Static storage is used in Java for *class variables*, which are declared using the keyword `static`:

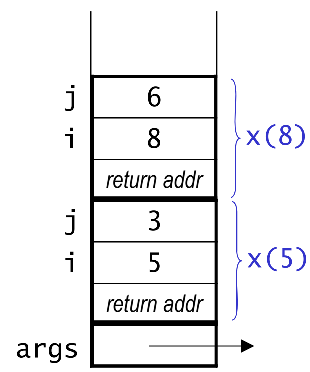
```
public static final PI = 3.1495;
public static int numCompares;
```

- There is only one copy of each class variable; it is shared by all instances (i.e., all objects) of the class.
- The Java runtime system allocates memory for class variables when the class is first encountered.
  - this memory stays fixed for the duration of the program

Memory Management, Type II: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.
- For each method call, a new *stack frame* is added to the top of the stack.

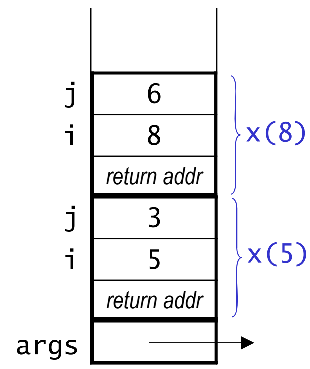
```
public class Foo {
  static void x(int i) {
      int j = i - 2;
      if (i >= 6) return;
      x(i + j);
  }
  public static void
    main(String[] args) {
      x(5);
  }
}
```



- When a method completes, its stack frame is removed. The values stored there are *not* preserved.

Stack Storage (cont.)

- Memory allocation on the stack is very efficient, because there are only two simple operations:
  - add a stack frame to the top of the stack
  - remove a stack frame from the top of the stack
- Limitations of stack storage:
  It can't be used if
  - the amount of memory needed isn't known in advance
  - we need the memory to persist after the method completes
- Because of these limitations, Java never stores arrays or objects on the stack.

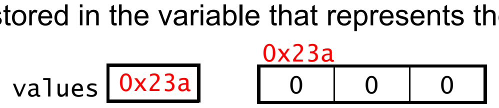


Memory Management, Type III: Heap Storage

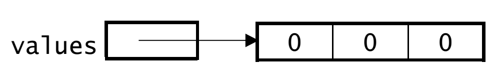
- Arrays and objects in Java are stored in a region of memory known as *the heap*.
- Memory on the heap is allocated using the `new` operator:

```
int[] values = new int[3];
ArrayBag b = new ArrayBag();
```

- `new` returns the memory address of the start of the array or object on the heap.
- This memory address – which is referred to as a *reference* in Java – is stored in the variable that represents the array/object:



- We will often use an arrow to represent a reference:



Heap Storage (cont.)

- In Java, an object or array persists until there are no remaining references to it.
- You can explicitly drop a reference by setting the variable equal to `null`. For example:

```
int[] values = {5, 23, 61, 10};
System.out.println(mean(values, 4));
values = null;
```

- Unused objects/arrays are *automatically* reclaimed by a process known as garbage collection.
  - makes their memory available for other objects or arrays

Constructors for the `ArrayBag` Class

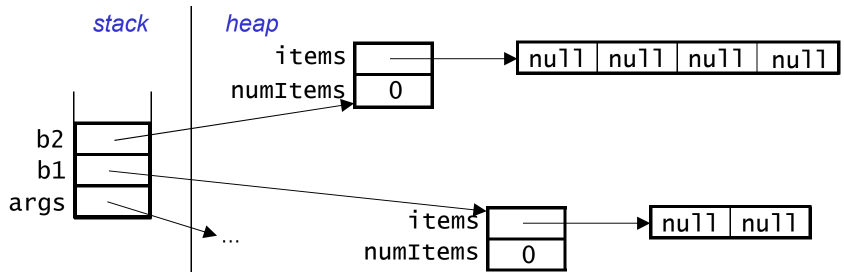
```
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
    public static final int DEFAULT_MAX_SIZE = 50;

    public ArrayBag() {
        items = new Object[DEFAULT_MAX_SIZE];
        numItems = 0;
    }
    public ArrayBag(int maxSize) {
        if (maxSize <= 0)
            throw new IllegalArgumentException(
              "maxSize must be > 0");
        items = new Object[maxSize];
        numItems = 0;
    }
    …
}
```

- If the user inputs an invalid value for `maxSize`, we throw an exception.

Example: Creating Two `ArrayBag` Objects

```
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);
    …
}
```

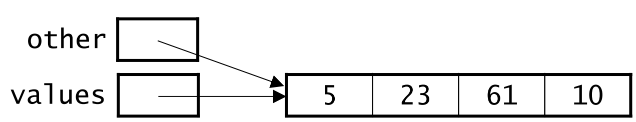
- After the objects have been created, here's what we have:



Copying References

- A variable that represents an array or object is known as a *reference variable*.
- Assigning the value of one reference variable to another reference variable copies the reference to the array or object. It does *not* copy the array or object itself.

```
int[] values = {5, 23, 61, 10};
int[] other = values;
```



- Given the lines above, what will the lines below output?

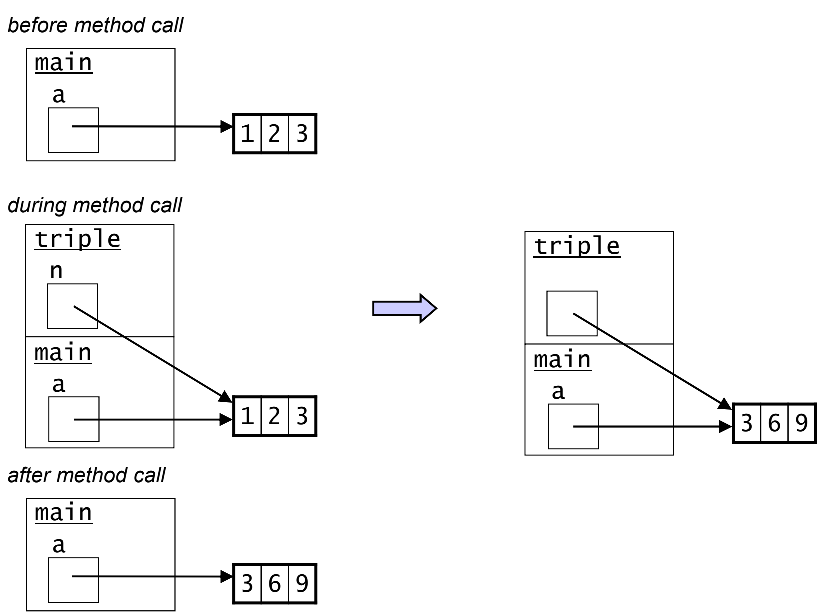
```
other[2] = 17;
System.out.println(values[2] + " " + other[2]);
```

Passing an Object/Array to a Method

- When a method is passed an object or array as a parameter, the method gets a copy of the *reference* to the object or array, *not* a copy of the object or array itself.
- Thus, any changes that the method makes to the object/array will still be there when the method returns.
- Consider the following:

```
public static void main(String[] args) {
    int[] a = {1, 2, 3};
    triple(a);
    System.out.println(Arrays.toString(a));
}

public static void triple(int[] n) {
    for (int i = 0; i < n.length; i++) {
        n[i] = n[i] * 3;
    }
}
```

Passing an Object/Array to a Method (cont.)



A Method for Adding an Item to a Bag

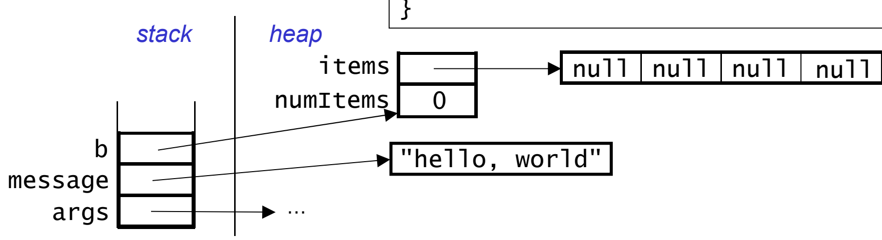
```
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
    …
    public boolean add(Object item) {
        if (item == null)
            throw new IllegalArgumentException();
        if (numItems == items.length)
            return false;  // no more room!
        else {
            items[numItems] = item;
            numItems++;
            return true;
        }
    }
    …
}
```

- add() is an instance method (a.k.a. a non-static method), so it has access to the fields of the current object.

Example: Adding an Item

```
public static void main(String[] args) {
    String message = "hello, world";
    ArrayBag b = new ArrayBag(4);
    b.add(message);
    …
}
```

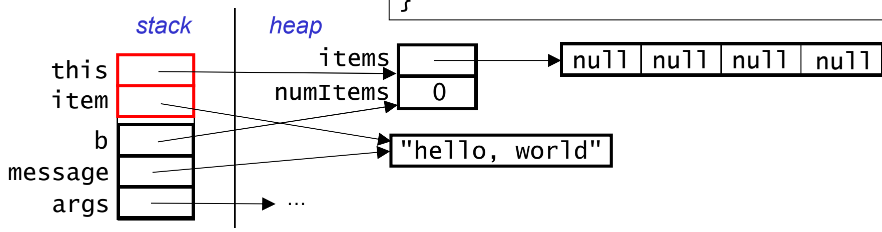
```
public boolean add(Object item) {
    …
    else {
        items[numItems] = item;
        numItems++;
        return true;
    }
}
```



Example: Adding an Item (cont.)

```
public static void main(String[] args) {
    String message = "hello, world";
    ArrayBag b = new ArrayBag(4);
    b.add(message);
    …
}
```

```
public boolean add(Object item) {
    …
    else {
        items[numItems] = item;
        numItems++;
        return true;
    }
}
```

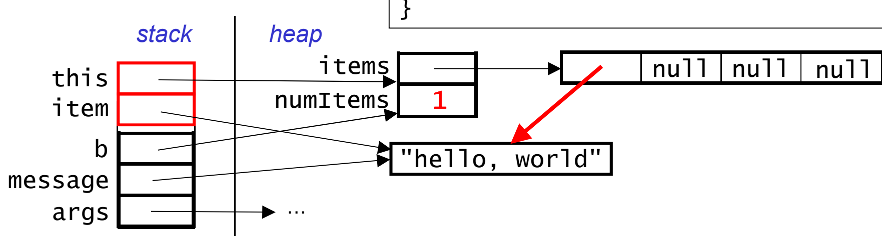


- add's stack frame includes:
  - `item`, which stores a copy of the reference passed as a param.
  - `this`, which stores a reference to the called/current object

Example: Adding an Item (cont.)

```
public static void main(String[] args) {
    String message = "hello, world";
    ArrayBag b = new ArrayBag(4);
    b.add(message);
    …
}
```

```
public boolean add(Object item) {
    …
    else {
        items[numItems] = item;
        numItems++;
        return true;
    }
}
```

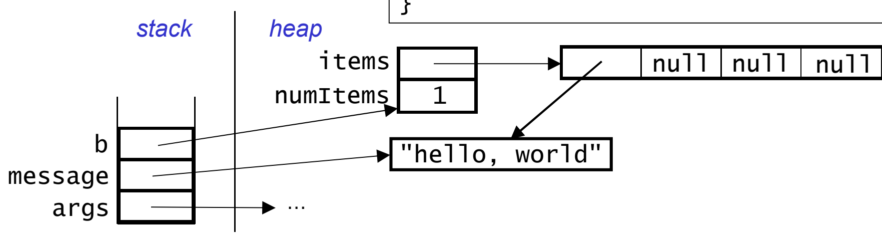


- The method modifies the `items` array and `numItems`.
  - note that the array holds a copy of the *reference* to the item, not a copy of the item itself.

Example: Adding an Item (cont.)

```
public static void main(String[] args) {
    String message = "hello, world";
    ArrayBag b = new ArrayBag(4);
    b.add(message);
    …
}
```

```
public boolean add(Object item) {
    …
    else {
        items[numItems] = item;
        numItems++;
        return true;
    }
}
```



- After the method call returns, `add`'s stack frame is removed from the stack.

Using the Implicit Parameter

```
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
    …
    public boolean add(Object item) {
        if (item == null)
            throw new IllegalArgumentException();
        if (this.numItems == this.items.length)
            return false;   // no more room!
        else {
            this.items[this.numItems] = item;
            this.numItems++;
            return true;
        }
    }
    …
}
```

- We can use `this` to emphasize the fact that we're accessing fields in the current object.

Determining if a Bag Contains an Item

- Let's write the `ArrayBag contains()` method together.
- Should return `true` if an object equal to `item` is found, and `false` otherwise.

```
__________________ contains(______________ item) {




}
```

An Incorrect `contains()` Method

```
public boolean contains(Object item) {
    for (int i = 0; i < numItems; i++) {
        if (items[i].equals(item))
            return true;
        else
            return false;
    }
    return false;
}
```

- Why won't this version of the method work in all cases?
- When would it work?

A Method That Takes a Bag as a Parameter

```
public boolean containsAll(Bag otherBag) {
    if (otherBag == null || otherBag.numItems() == 0)
        return false;

    Object[] otherItems = otherBag.toArray();
    for (int i = 0; i < otherItems.length; i++) {
        if (!contains(otherItems[i]))
            return false;
    }

    return true;
}
```

- We use `Bag` instead of `ArrayBag` as the type of the parameter.
  - allows this method to be part of the `Bag` interface
  - allows us to pass in *any* object that implements `Bag`
- Because the parameter may not be an `ArrayBag`, we can't assume it has `items` and `numItems` fields.
  - instead, we use `toArray()` and `numItems()`

A Need for Casting

- Let's say that we want to store a collection of `String` objects in an `ArrayBag`.
- `String` is a subclass of `Object`, so we can store `String` objects in the bag without doing anything special:

```
ArrayBag stringBag = new ArrayBag();
stringBag.add("hello");
stringBag.add("world");
```

- `Object` isn't a subclass of `String`, so this will <u>not</u> work:

```
String str = stringBag.grab();    // compiler error
```

- Instead, we need to use casting:

```
String str = (String)stringBag.grab();
```

Extra: Thinking About a Method's Efficiency

- For a bag with 1000 items, how many items will `contains()` look at:
  - in the best case?
  - in the worst case?
  - in the average case?

- Could we make it more efficient?

- If so, what changes would be needed to do so, and what would be the impact of those changes?

Extra: Understanding Memory Management

- Our Bag ADT has a method `toArray()`, which returns an array containing the current contents of the bag
  - allows users of the ADT to iterate over the items

- When implementing `toArray()` in our `ArrayBag` class, can we just return a reference to the `items` array? Why or why not?

Recursion and Recursive Backtracking

Computer Science E-22
Harvard Extension School

David G. Sullivan, Ph.D.

Iteration

- When we encounter a problem that requires repetition, we often use *iteration* – i.e., some type of loop.
- Sample problem: printing the series of integers from n1 to n2, where n1 <= n2.
  - example: `printSeries(5, 10)` should print the following:

    `5, 6, 7, 8, 9, 10`
- Here's an iterative solution to this problem:

```
public static void printSeries(int n1, int n2) {
    for (int i = n1; i < n2; i++) {
        System.out.print(i + ", ");
    }
    System.out.println(n2);
}
```

Recursion

- An alternative approach to problems that require repetition is to solve them using *recursion.*
- A recursive method is a method that calls itself.
- Applying this approach to the print-series problem gives:

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

Tracing a Recursive Method

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- What happens when we execute `printSeries(5, 7)`?

```
printSeries(5, 7):
    System.out.print(5 + ", ");
    printSeries(6, 7):
        System.out.print(6 + ", ");
        printSeries(7, 7):
            System.out.print(7);
            return
        return
    return
```

Recursive Problem-Solving

- When we use recursion, we solve a problem by reducing it to a simpler problem of the same kind.
- We keep doing this until we reach a problem that is simple enough to be solved directly.
- This simplest problem is known as the *base case*.

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {            // base case
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- The base case stops the recursion, because it doesn't make another call to the method.

Recursive Problem-Solving (cont.)

- If the base case hasn't been reached, we execute the *recursive case.*

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {            // base case
        System.out.println(n2);
    } else {                   // recursive case
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- The recursive case:
  - reduces the overall problem to one or more simpler problems of the same kind
  - makes recursive calls to solve the simpler problems

Structure of a Recursive Method

```
recursiveMethod(parameters) {
    if (stopping condition) {
        // handle the base case
    } else {
        // recursive case:
        // possibly do something here
        recursiveMethod(modified parameters);
        // possibly do something here
    }
}
```

- There can be multiple base cases and recursive cases.
- When we make the recursive call, we typically use parameters that bring us closer to a base case.

Tracing a Recursive Method: Second Example

```
public static void mystery(int i) {
    if (i <= 0) {      // base case
        return;
    }
    // recursive case
    System.out.println(i);
    mystery(i - 1);
    System.out.println(i);
}
```

- What happens when we execute `mystery(2)`?

Printing a File to the Console

- Here's a method that prints a file using iteration:

```
public static void print(Scanner input) {
    while (input.hasNextLine()) {
        System.out.println(input.nextLine());
    }
}
```

- Here's a method that uses recursion to do the same thing:

```
public static void printRecursive(Scanner input) {
    // base case
    if (!input.hasNextLine()) {
        return;
    }

    // recursive case
    System.out.println(input.nextLine());
    printRecursive(input);  // print the rest
}
```

Printing a File in Reverse Order

- What if we want to print the lines of a file in reverse order?
- It's not easy to do this using iteration. Why not?
- It's easy to do it using recursion!
- How could we modify our previous method to make it print the lines in reverse order?

```
public static void printRecursive(Scanner input) {
    if (!input.hasNextLine()) {   // base case
        return;
    }

    String line = input.nextLine();
    System.out.println(line);
    printRecursive(input);  // print the rest
}
```

A Recursive Method That Returns a Value

- Simple example: summing the integers from 1 to n

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int total = n + sum(n – 1);
    return total;
}
```

- Example of this approach to computing the sum:

sum(6) = 6 + sum(5)
= 6 + 5 + sum(4)

Tracing a Recursive Method

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int total = n + sum(n – 1);
    return total;
}
```

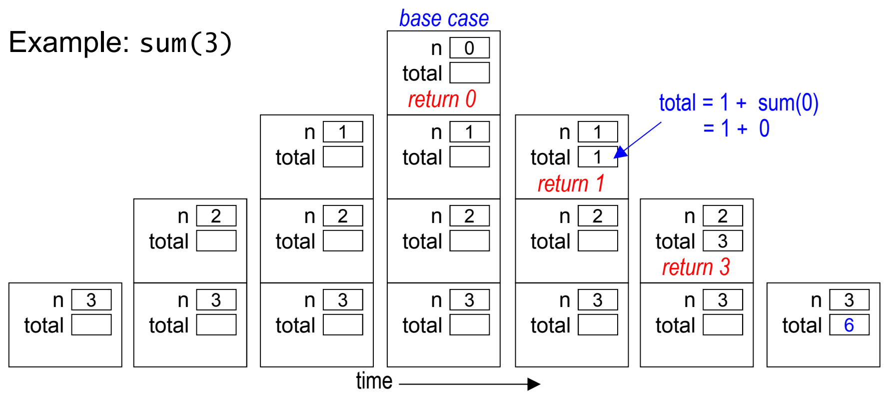
- What happens when we execute `int x = sum(3);` from inside the `main()` method?

```
main() calls sum(3)
    sum(3) calls sum(2)
        sum(2) calls sum(1)
            sum(1) calls sum(0)
                sum(0) returns 0
            sum(1) returns 1 + 0 or 1
        sum(2) returns 2 + 1 or 3
    sum(3) returns 3 + 3 or 6
main()
```

Tracing a Recursive Method on the Stack

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int total = n + sum(n – 1);
    return total;
}
```

Example: `sum(3)`



Infinite Recursion

- We have to ensure that a recursive method will eventually reach a base case, regardless of the initial input.
- Otherwise, we can get *infinite recursion*.
  - produces *stack overflow* – there's no room for more frames on the stack!
- Example: here's a version of our `sum()` method that uses a different test for the base case:

```
public static int sum(int n) {
    if (n == 0) {
        return 0;
    }
    int total = n + sum(n – 1);
    return total;
}
```

  - what values of n would cause infinite recursion?

Thinking Recursively

- When solving a problem using recursion, ask yourself these questions:
  1. How can I break this problem down into one or more smaller subproblems?
     - make recursive method calls to solve the subproblems
  2. What are the base cases?
     - i.e., which subproblems are small enough to solve directly?
  3. Do I need to combine the solutions to the subproblems? If so, how should I do so?

Raising a Number to a Power

- We want to write a recursive method to compute

$$x^n = \underbrace{x*x*x* \quad *x}_{n \text{ of them}}$$

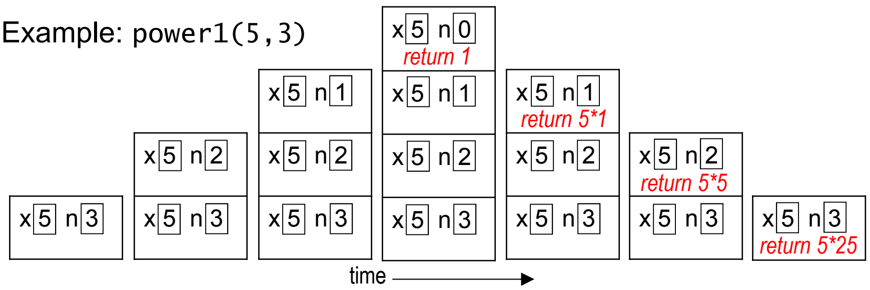
  where x and n are both integers and n >= 0.

- Examples:
  - 2^{10} = 2*2*2*2*2*2*2*2*2*2 = 1024
  - 10^5 = 10*10*10*10*10 = 100000
- Computing a power recursively: $2^{10} = 2*2^9$
  $= 2*(2 * 2^8)$
  $=$
- Recursive definition: $x^n = x * x^{n-1}$ when $n > 0$
  $x^0 = 1$

Power Method: First Try

```
public class Power {
    public static int power1(int x, int n) {
        if (n < 0)
            throw new IllegalArgumentException(
              “n must be >= 0”);
        if (n == 0)
            return 1;
        else
            return x * power1(x, n-1);
    }
}
```

Example: `power1(5,3)`



Power Method: Second Try

- There’s a better way to break these problems into subproblems. For example: $2^{10} = (2*2*2*2*2)*(2*2*2*2*2)$
  $= (2^5) * (2^5) = (2^5)^2$
- A more efficient recursive definition of x^n (when $n > 0$):
  $x^n = (x^{n/2})^2$ when n is even
  $x^n = x * (x^{n/2})^2$ when n is odd (using integer division for n/2)
- Let's write the corresponding method together:

```
public static int power2(int x, int n) {






}
```

Analyzing power2

- How many method calls would it take to compute 2^{1000} ?

```
power2(2, 1000)
  power2(2, 500)
    power2(2, 250)
      power2(2, 125)
        power2(2, 62)
          power2(2, 31)
            power2(2, 15)
              power2(2, 7)
                power2(2, 3)
                  power2(2, 1)
                    power2(2, 0)
```

- Much more efficient than power1() for large n.
- It can be shown that it takes approx. $\log_2 n$ method calls.

An Inefficient Version of power2

- What's wrong with the following version of `power2()`?

```
public static int power2Bad(int x, int n) {
    // code to handle n < 0 goes here...
    if (n == 0)
        return 1;
    if ((n % 2) == 0)
        return power2(x, n/2) * power2(x, n/2);
    else
        return x * power2(x, n/2) * power2(x, n/2);
}
```

Processing a String Recursively

- A string is a recursive data structure. It is either:
  - empty ("")
  - a single character, followed by a string
- Thus, we can easily use recursion to process a string.
  - process one or two of the characters
  - make a recursive call to process the rest of the string
- Example: print a string vertically, one character per line:

```
public static void printVertical(String str) {
    if (str == null || str.equals("")) {
        return;
    }

    System.out.println(str.charAt(0)); // first char
    printVertical(str.substring(1));   // rest of string
}
```

Counting Occurrences of a Character in a String

- Let's design a recursive method called `numOccur()`.
- `numOccur(ch, str)` should return the number of times that the character `ch` appears in the string `str`
- Thinking recursively:

Counting Occurrences of a Character in a String (cont.)

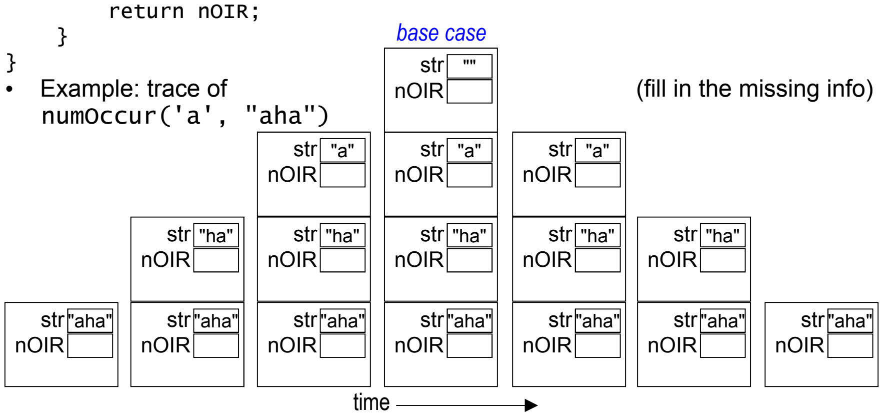
- Put the method definition here:

Tracing a Recursive Method on the Stack

```
public static int numOccur(char ch, String str) {
    if (str == null || str.equals("")) {
        return 0;
    }
    int nOIR = numOccur(ch, str.substring(1));
    if (str.charAt(0) == ch) {
        return 1 + nOIR;
    } else {
        return nOIR;
    }
}
```

- Example: trace of `numOccur('a', "aha")`

(fill in the missing info)



Common Mistake

- This version of the method does *not* work:

```
public static int numOccur(char ch, String str) {
    if (str == null || str.equals("")) {
        return 0;
    }

    int count = 0;
    if (str.charAt(0) == ch) {
        count++;
    }

    numOccur(ch, str.substring(1));
    return count;
}
```

Another Faulty Approach

- Some people make `count` "global" to fix the prior version:

```
public static int count = 0;

public static int numOccur(char ch, String str) {
    if (str == null || str.equals("")) {
        return 0;
    }
    if (str.charAt(0) == ch) {
        count++;
    }
    numOccur(ch, str.substring(1));
    return count;
}
```

- Not recommended, and not allowed on the problem sets!
- Problems with this approach?

Removing Vowels from a String

- Let's design a recursive method called `removeVowels()`.
- `removeVowels(str)` should return a string in which all of the vowels in the string `str` have been removed.
  - example:
    `removeVowels("recurse")`

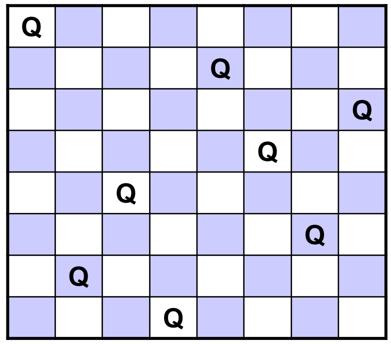
    should return
    `"rcrs"`
- Thinking recursively:

Removing Vowels from a String (cont.)

- Put the method definition here:

Recursive Backtracking: the n-Queens Problem

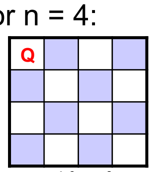
- Find all possible ways of placing n queens on an n x n chessboard so that no two queens occupy the same row, column, or diagonal.
- Sample solution for n = 8:



- This is a classic example of a problem that can be solved using a technique called *recursive backtracking*.

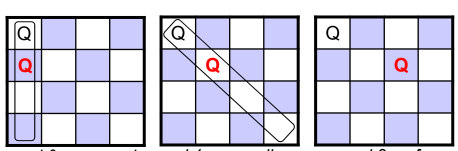
Recursive Strategy for n-Queens

- Consider one row at a time. Within the row, consider one column at a time, looking for a “safe” column to place a queen.
- If we find one, place the queen, and *make a recursive call* to place a queen on the next row.
- If we can’t find one, *backtrack* by returning from the recursive call, and try to find another safe column in the previous row.
- Example for n = 4:
  - row 0:



col 0: safe

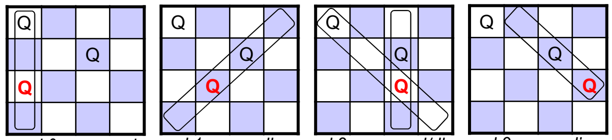
  - row 1:



col 0: same col *col 1: same diag* *col 2: safe*

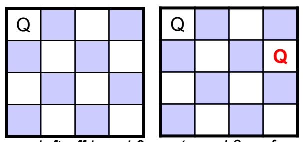
4-Queens Example (cont.)

- row 2:



col 0: same col | col 1: same diag | col 2: same col/diag | col 3: same diag

- We've run out of columns in row 2!
- *Backtrack* to row 1 by returning from the recursive call.
  - pick up where we left off
  - we had already tried columns 0-2, so now we try column 3:

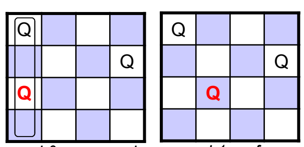


we left off in col 2 | try col 3: safe

- Continue the recursion as before.

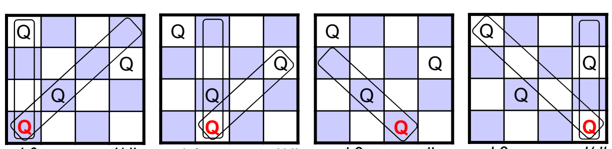
4-Queens Example (cont.)

- row 2:



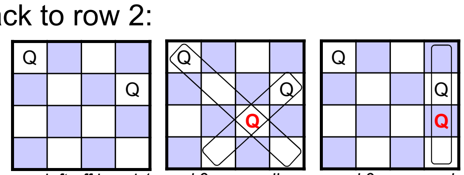
col 0: same col | col 1: safe

- row 3:



col 0: same col/diag | col 1: same col/diag | col 2: same diag | col 3: same col/diag

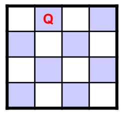
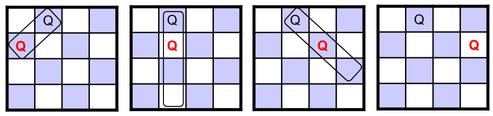
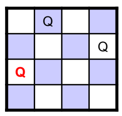
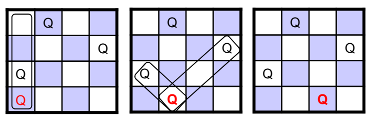
- Backtrack to row 2:



we left off in col 1 | col 2: same diag | col 3: same col

- Backtrack to row 1. No columns left, so backtrack to row 0!

4-Queens Example (cont.)

- row 0: 
- row 1: 
- row 2: 
- row 3: 

A solution!

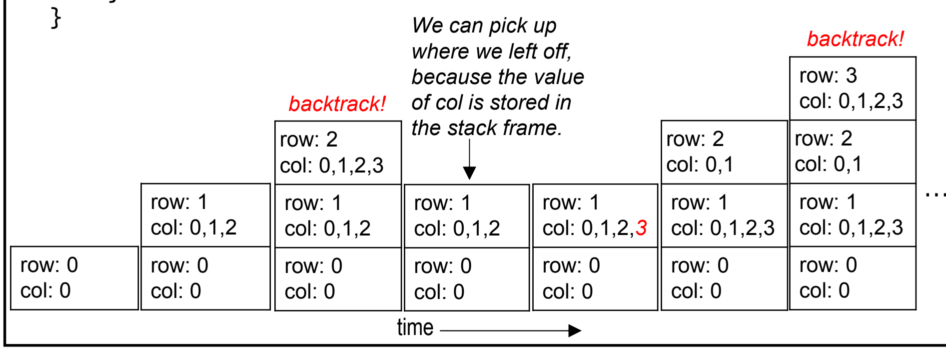
findSafeColumn() Method

```
public void findSafeColumn(int row) {
    if (row == boardSize) {  // base case: a solution!
        solutionsFound++;
        displayBoard();
        if (solutionsFound >= solutionTarget)
            System.exit(0);
        return;
    }

    for (int col = 0; col < boardSize; col++) {
        if (isSafe(row, col)) {
            placeQueen(row, col);

            // Move onto the next row.
            findSafeColumn(row + 1);

            // If we get here, we've backtracked.
            removeQueen(row, col);
        }
    }
}
```

Note: neither row++ *nor* ++row *will work here.*

Tracing findSafeColumn()

```
public void findSafeColumn(int row) {
    if (row == boardSize) {
        // code to process a solution goes here...
    }
    for (int col = 0; col < BOARD_SIZE; col++) {
        if (isSafe(row, col)) {
            placeQueen(row, col);
            findSafeColumn(row + 1);
            removeQueen(row, col);
        }
    }
}
```



Template for Recursive Backtracking

```
void findSolutions(n, other params) {
    if (found a solution) {
        solutionsFound++;
        displaySolution();
        if (solutionsFound >= solutionTarget)
            System.exit(0);
        return;
    }

    for (val = first to last) {
        if (isValid(val, n)) {
            applyValue(val, n);
            findSolutions(n + 1, other params);
            removeValue(val, n);
        }
    }
}
```

Template for Finding a Single Solution

```
boolean findSolutions(n, other params) {
    if (found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val, n)) {
            applyValue(val, n);
            if (findSolutions(n + 1, other params))
                return true;
            removeValue(val, n);
        }
    }

    return false;
}
```

Data Structures for n-Queens

- Three key operations:
  - `isSafe(row, col)`: check to see if a position is safe
  - `placeQueen(row, col)`
  - `removeQueen(row, col)`
- A two-dim. array of booleans would be sufficient:

```
public class Queens {
    private boolean[][] queenOnSquare;
```

- Advantage: easy to place or remove a queen:

```
    public void placeQueen(int row, int col) {
        queenOnSquare[row][col] = true;
    }
    public void removeQueen(int row, int col) {
        queenOnSquare[row][col] = false;
    }
…
```

- Problem: `isSafe()` takes a lot of steps. What matters more?

Additional Data Structures for n-Queens

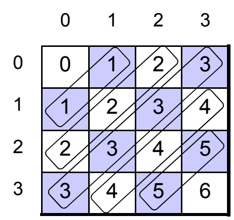
- To facilitate `isSafe()`, add three arrays of booleans:

```
private boolean[] colEmpty;
private boolean[] upDiagEmpty;
private boolean[] downDiagEmpty;
```

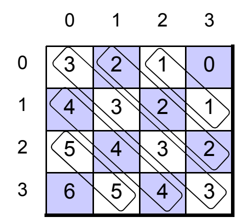
- An entry in one of these arrays is:
  - `true` if there are no queens in the column or diagonal
  - `false` otherwise

- Numbering diagonals to get the indices into the arrays:

upDiag = row + col



downDiag = (boardSize – 1) + row – col



Using the Additional Arrays

- Placing and removing a queen now involve updating four arrays instead of just one. For example:

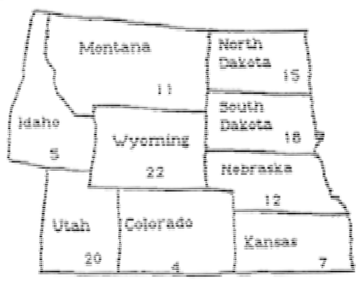
```
public void placeQueen(int row, int col) {
    queenOnSquare[row][col] = true;
    colEmpty[col] = false;
    upDiagEmpty[row + col] = false;
    downDiagEmpty[(boardSize - 1) + row - col] = false;
}
```

- However, checking if a square is safe is now more efficient:

```
public boolean isSafe(int row, int col) {
    return (colEmpty[col]
      && upDiagEmpty[row + col]
      && downDiagEmpty[(boardSize - 1) + row - col]);
}
```

Recursive Backtracking II: Map Coloring

- Using just four colors (e.g., red, orange, green, and blue), we want color a map so that no two bordering states or countries have the same color.
- Sample map (numbers show alphabetical order in full list of state names):



- This is another example of a problem that can be solved using recursive backtracking.

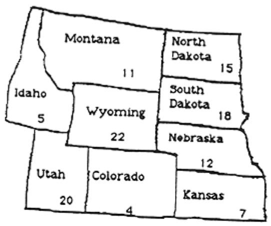
Applying the Template to Map Coloring

```
boolean findSolutions(n, other params) {
    if (found a solution) {
        displaySolution();
        return true;
    }
    for (val = first to last) {
        if (isValid(val, n)) {
            applyValue(val, n);
            if (findSolutions(n + 1, other params))
                return true;
            removeValue(val, n);
        }
    }
    return false;
}
```

template element	*meaning in map coloring*
n	
found a solution	
val	
isValid(val, n)	
applyValue(val, n)	
removeValue(val, n)	

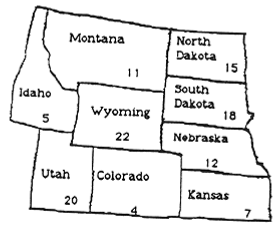
Map Coloring Example

consider the states in alphabetical order. colors = { red, yellow, green, blue }.



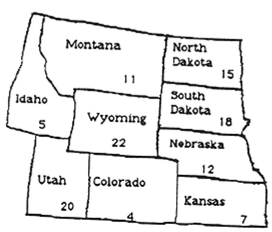
We color Colorado through Utah without a problem.

Colorado:
Idaho:
Kansas:
Montana:
Nebraska:
North Dakota:
South Dakota:
Utah:



No color works for Wyoming, so we backtrack

Map Coloring Example (cont.)



Now we can complete the coloring:

Recursive Backtracking in General

- Useful for *constraint satisfaction problems* that involve assigning values to variables according to a set of constraints.
  - n-Queens:
    - variables = Queen's position in each row
    - constraints = no two queens in same row, column, diagonal
  - map coloring
    - variables = each state's color
    - constraints = no two bordering states with the same color
  - many others: factory scheduling, room scheduling, etc.
- Backtracking reduces the # of possible value assignments that we consider, because it never considers invalid assignments....
- Using recursion allows us to easily handle an arbitrary number of variables.
  - stores the state of each variable in a separate stack frame

Recursion vs. Iteration

- Recursive methods can often be easily converted to a non-recursive method that uses iteration.
- This is especially true for methods in which:
  - there is only one recursive call
  - it comes at the end (tail) of the method

  These are known as *tail-recursive* methods.
- Example: an iterative sum() method.

```
public static int sum(n) {
    // handle negative values of n here
    int sum = 0;
    for (int i = 1; i <= n; i++)
        sum += i;
    return sum;
}
```

Recursion vs. Iteration (cont.)

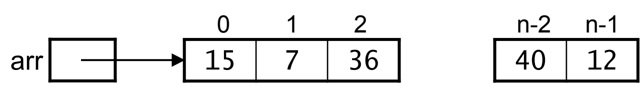
- Once you're comfortable with using recursion, you'll find that some algorithms are easier to implement using recursion.
- We'll also see that some data structures lend themselves to recursive algorithms.
- Recursion is a bit more costly because of the overhead involved in invoking a method.
- Rule of thumb:
  - if it's easier to formulate a solution recursively, use recursion, unless the cost of doing so is too high
  - otherwise, use iteration

Sorting and Algorithm Analysis

Computer Science E-22
Harvard Extension School

David G. Sullivan, Ph.D.

Sorting an Array of Integers



- Ground rules:
  - sort the values in increasing order
  - sort "in place," using only a small amount of additional storage
- Terminology:
  - position: one of the memory locations in the array
  - element: one of the data items stored in the array
  - element i: the element at position i
- Goal: minimize the number of **comparisons *C*** and the number of **moves *M*** needed to sort the array.
  - move = copying an element from one position to another
    example: `arr[3] = arr[5];`

Defining a Class for our Sort Methods

```
public class Sort {
    public static void bubbleSort(int[] arr) {
        ...
    }
    public static void insertionSort(int[] arr) {
        ...
    }
    ...
}
```

- Our `Sort` class is simply a collection of methods like Java's built-in `Math` class.
- Because we never create `Sort` objects, all of the methods in the class must be *static*.
  - outside the class, we invoke them using the class name: e.g., `Sort.bubbleSort(arr)`

Defining a Swap Method

- It would be helpful to have a method that swaps two elements of the array.
- Why won't the following work?

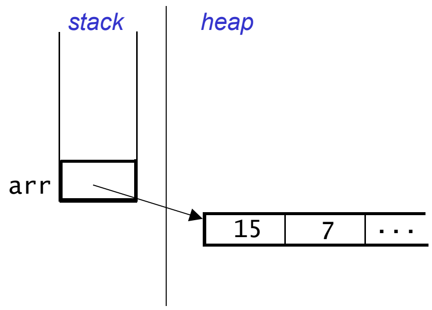
```
public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

An Incorrect Swap Method

```
public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

- Trace through the following lines to see the problem:

```
int[] arr = {15, 7, …};
swap(arr[0], arr[1]);
```



A Correct Swap Method

- This method works:

```
public static void swap(int[] arr, int a, int b) {
    int temp = arr[a];
    arr[a] = arr[b];
    arr[b] = temp;
}
```

- Trace through the following with a memory diagram to convince yourself that it works:

```
int[] arr = {15, 7, …};
swap(arr, 0, 1);
```

Selection Sort

- Basic idea:
  - consider the positions in the array from left to right
  - for each position, find the element that belongs there and put it in place by swapping it with the element that's currently there
- Example:

0	1	2	3	4
15	6	2	12	4

0	1	2	3	4
2	6	15	12	4

0	1	2	3	4
2	4	15	12	6

0	1	2	3	4
2	4	6	12	15

Why don't we need to consider position 4?

Selecting an Element

- When we consider position `i`, the elements in positions 0 through `i` – 1 are already in their final positions.

example for `i` = 3:

0	1	2	3	4	5	6
2	4	7	21	25	10	17

- To select an element for position `i`:
  - consider elements `i`, `i+1,i+2,…,arr.length` – 1, and keep track of `indexMin`, the index of the smallest element seen thus far

`indexMin: 3, 5`

0	1	2	3	4	5	6
2	4	7	21	25	10	17

  - when we finish this pass, `indexMin` is the index of the element that belongs in position `i`.
  - swap `arr[i]` and `arr[indexMin]`:

0	1	2	3	4	5	6
2	4	7	10	25	21	17

Implementation of Selection Sort

- Use a helper method to find the index of the smallest element:

```
private static int indexSmallest(int[] arr,
  int lower, int upper) {
    int indexMin = lower;

    for (int i = lower+1; i <= upper; i++)
        if (arr[i] < arr[indexMin])
            indexMin = i;

    return indexMin;
}
```

- The actual sort method is very simple:

```
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length-1; i++) {
        int j = indexSmallest(arr, i, arr.length-1);
        swap(arr, i, j);
    }
}
```

Time Analysis

- Some algorithms are much more efficient than others.
- The *time efficiency* or *time complexity* of an algorithm is some measure of the number of "operations" that it performs.
  - for sorting algorithms, we'll focus on two types of operations: comparisons and moves
- The number of operations that an algorithm performs typically depends on the size, n, of its input.
  - for sorting algorithms, n is the # of elements in the array
  - `C(n)` = number of comparisons
  - `M(n)` = number of moves
- To express the time complexity of an algorithm, we'll express the number of operations performed as a function of `n`.
  - examples: $C(n) = n^2 + 3n$
    $M(n) = 2n^2 - 1$

Counting Comparisons by Selection Sort

```
private static int indexSmallest(int[] arr, int lower, int upper){
    int indexMin = lower;

    for (int i = lower+1; i <= upper; i++)
        if (arr[i] < arr[indexMin])
            indexMin = i;

    return indexMin;
}
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length-1; i++) {
        int j = indexSmallest(arr, i, arr.length-1);
        swap(arr, i, j);
    }
}
```

- To sort n elements, selection sort performs $n - 1$ passes:

  on 1st pass, it performs $n - 1$ comparisons to find `indexSmallest`
  on 2nd pass, it performs $n - 2$ comparisons

  on the $(n-1)$st pass, it performs 1 comparison

- Adding up the comparisons for each pass, we get:

  $C(n) = 1 + 2 + \ldots + (n - 2) + (n - 1)$

Counting Comparisons by Selection Sort (cont.)

- The resulting formula for $C(n)$ is the sum of an arithmetic sequence:

$$C(n) = 1 + 2 + \ldots + (n - 2) + (n - 1) = \sum_{i=1}^{n-1} i$$

- Formula for the sum of this type of arithmetic sequence:

$$\sum_{i=1}^{m} i = \frac{m(m+1)}{2}$$

- Thus, we can simplify our expression for C(n) as follows:

$$\begin{aligned} C(n) &= \sum_{i=1}^{n-1} i \\ &= \frac{(n-1)((n-1)+1)}{2} \\ &= \frac{(n-1)n}{2} \end{aligned}$$

$$\boxed{C(n) = n^2/2 - n/2}$$

Focusing on the Largest Term

- When n is large, mathematical expressions of n are dominated by their "largest" term — i.e., the term that grows fastest as a function of n.
  - example:

n	$n^2/2$	$n/2$	$n^2/2 - n/2$
10	50	5	45
100	5000	50	4950
10000	50,000,000	5000	49,995,000

- In characterizing the time complexity of an algorithm, we'll focus on the largest term in its operation-count expression.
  - for selection sort, $C(n) = n^2/2 - n/2 \approx n^2/2$
- In addition, we'll typically ignore the coefficient of the largest term (e.g., $n^2/2 \rightarrow n^2$).

Big-*O* Notation

- We specify the largest term using big-*O* notation.
  - e.g., we say that $C(n) = n^2/2 - n/2$ is $O(n^2)$
- Common classes of algorithms:

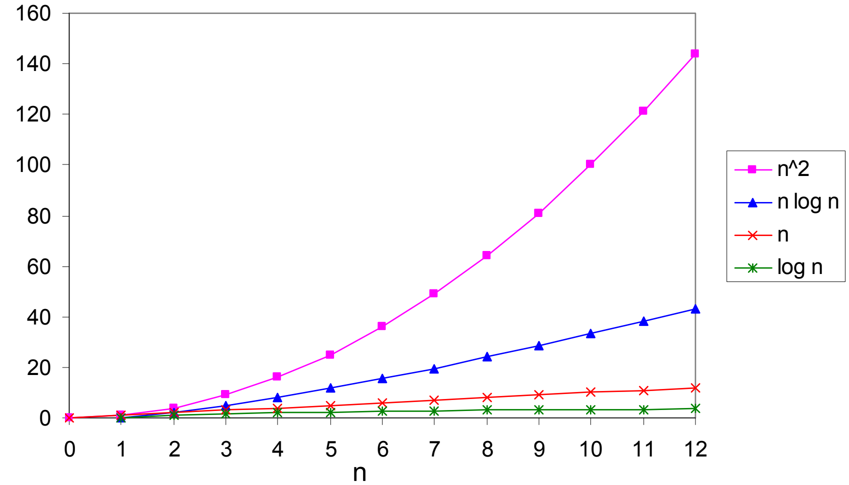
name	example expressions	big-O notation
constant time	1, 7, 10	$O(1)$
logarithmic time	$3\log_{10}n$, $\log_2 n + 5$	$O(\log n)$
linear time	$5n$, $10n - 2\log_2 n$	$O(n)$
nlogn time	$4n\log_2 n$, $n\log_2 n + n$	$O(n\log n)$
quadratic time	$2n^2 + 3n$, $n^2 - 1$	$O(n^2)$
exponential time	2^n, $5e^n + 2n^2$	$O(c^n)$

slower

- For large inputs, efficiency matters more than CPU speed.
  - e.g., an $O(\log n)$ algorithm on a slow machine will outperform an $O(n)$ algorithm on a fast machine

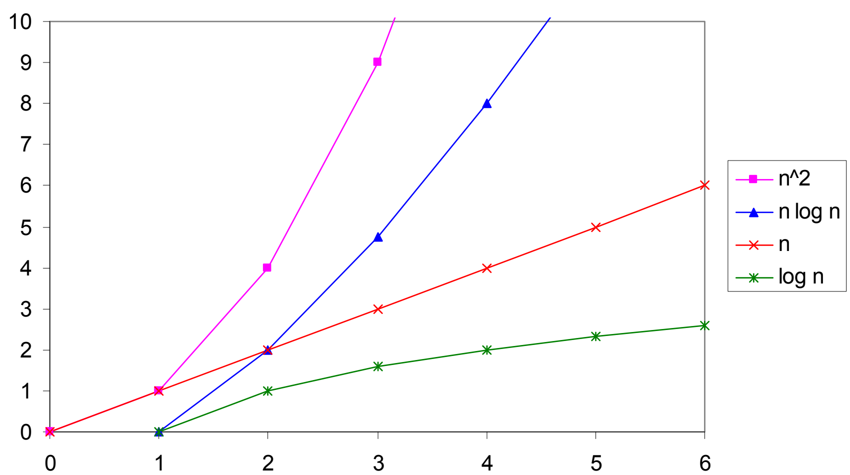
Ordering of Functions

- We can see below that: n^2 grows faster than $n\log_2 n$
  $n\log_2 n$ grows faster than n
  n grows faster than $\log_2 n$



Ordering of Functions (cont.)

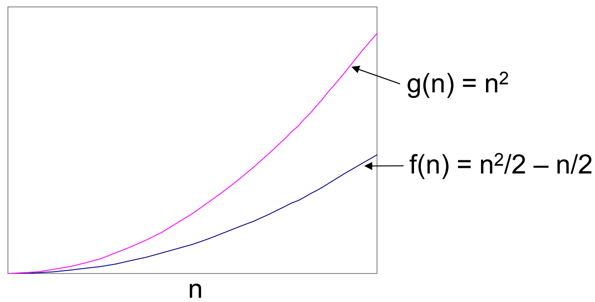
- Zooming in, we see that: n^2 >= n for all n >= 1
  $n\log_2 n$ >= n for all n >= 2
  n > $\log_2 n$ for all n >= 1



Mathematical Definition of Big-*O* Notation

- $f(n) = O(g(n))$ if there exist positive constants c and n_0 such that $f(n) <= c g(n)$ for all $n >= n_0$
- Example: $f(n) = n^2/2 – n/2$ is $O(n^2)$, because

$$n^2/2 – n/2 \underbrace{<=}_{c\,=\,1} n^2 \text{ for all } n \underbrace{>= 0}_{n_0\,=\,0}.$$



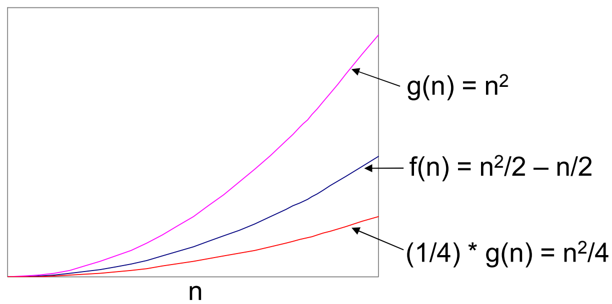
- Big-*O* notation specifies an *upper bound* on a function f(n) as n grows large.

Big-*O* Notation and Tight Bounds

- Big-O notation provides an upper bound, *not* a tight bound (upper and lower).
- Example:
  - $3n – 3$ is $O(n^2)$ because $3n – 3 <= n^2$ for all $n >= 1$
  - $3n – 3$ is also $O(2^n)$ because $3n – 3 <= 2^n$ for all $n >= 1$
- However, we generally try to use big-O notation to characterize a function as closely as possible – i.e., as if we were using it to specify a tight bound.
  - for our example, we would say that $3n – 3$ is $O(n)$

Big-Theta Notation

- In theoretical computer science, *big-theta* notation (Θ) is used to specify a tight bound.
- $f(n) = \Theta(g(n))$ if there exist constants c_1, c_2, and n_0 such that $c_1 g(n) \ <= \ f(n) \ <= \ c_2 g(n)$ for all $n > n_0$
- Example: $f(n) = n^2/2 - n/2$ is $\Theta(n^2)$, because
  $(1/4)*n^2 <= n^2/2 - n/2 <= n^2$ for all $n >= 2$
  $c_1 = 1/4$ $\quad c_2 = 1$ $\quad n_0 = 2$



Big-*O* Time Analysis of Selection Sort

- Comparisons: we showed that `C(n)` = $n^2/2 - n/2$
  - selection sort performs $O(n^2)$ comparisons
- Moves: after each of the `n-1` passes to find the smallest remaining element, the algorithm performs a swap to put the element in place.
  - `n-1` swaps, 3 moves per swap
  - `M(n) = 3(n-1) = 3n-3`
  - selection sort performs $O(n)$ moves.
- Running time (i.e., total operations): ?

Sorting by Insertion I: Insertion Sort

- Basic idea:
  - going from left to right, “insert” each element into its proper place with respect to the elements to its left, “sliding over” other elements to make room.
- Example:

0	1	2	3	4
15	*4*	2	12	6
4	15	*2*	12	6
2	4	15	*12*	6
2	4	12	15	*6*
2	4	6	12	15

Comparing Selection and Insertion Strategies

- In selection sort, we start with the *positions* in the array and *select* the correct elements to fill them.
- In insertion sort, we start with the *elements* and determine where to *insert* them in the array.
- Here’s an example that illustrates the difference:

0	1	2	3	4	5	6
18	12	15	9	25	2	17

- Sorting by selection:
  - consider position 0: find the element (2) that belongs there
  - consider position 1: find the element (9) that belongs there
  -
- Sorting by insertion:
  - consider the 12: determine where to insert it
  - consider the 15; determine where to insert it
  -

Inserting an Element

- When we consider element `i`, elements 0 through `i – 1` are already sorted with respect to each other.

example for `i = 3`:

0	1	2	3	4
6	14	19	9	...

- To insert element `i`:
  - make a copy of element `i`, storing it in the variable `toInsert`:

`toInsert` | 9 |

0	1	2	3
6	14	19	9

  - consider elements `i-1`, `i-2`,
    - if an element `>` `toInsert`, slide it over to the right
    - stop at the first element `<=` `toInsert`

`toInsert` | 9 |

0	1	2	3
6		14	19

  - copy `toInsert` into the resulting "hole":

0	1	2	3
6	9	14	19

Insertion Sort Example (done together)

description of steps

12	5	2	13	18	4

Implementation of Insertion Sort

```
public class Sort {
  ...
  public static void insertionSort(int[] arr) {
      for (int i = 1; i < arr.length; i++) {
          if (arr[i] < arr[i-1]) {
              int toInsert = arr[i];

              int j = i;
              do {
                  arr[j] = arr[j-1];
                  j = j - 1;
              } while (j > 0  &&  toInsert < arr[j-1]);

              arr[j] = toInsert;
          }
      }
  }
}
```

Time Analysis of Insertion Sort

- The number of operations depends on the contents of the array.
- *best case:*
- *worst case:*
- *average case:*

Sorting by Insertion II: Shell Sort

- Developed by Donald Shell in 1959
- Improves on insertion sort
- Takes advantage of the fact that insertion sort is fast when an array is almost sorted.
- Seeks to eliminate a disadvantage of insertion sort: if an element is far from its final location, many "small" moves are required to put it where it belongs.
- Example: if the largest element starts out at the beginning of the array, it moves one place to the right on *every* insertion!

0	1	2	3	4	5		1000
999	42	56	30	18	23	…	11

- Shell sort uses "larger" moves that allow elements to quickly get close to where they belong.

Sorting Subarrays

- Basic idea:
  - use insertion sort on subarrays that contain elements separated by some increment
    - increments allow the data items to make larger "jumps"
  - repeat using a decreasing sequence of increments
- Example for an initial increment of 3:

0	1	2	3	4	5	6	7
36	18	10	27	3	20	9	8

  - three subarrays:
    1) elements 0, 3, 6   2) elements 1, 4, 7   3) elements 2 and 5
- Sort the subarrays using insertion sort to get the following:

0	1	2	3	4	5	6	7
9	3	10	27	8	20	36	18

- Next, we complete the process using an increment of 1.

Shell Sort: A Single Pass

- We *don't* consider the subarrays one at a time.
- We consider elements `arr[incr]` through `arr[arr.length-1]`, inserting each element into its proper place with respect to the elements *from its subarray* that are to the left of the element.
- The same example (`incr` = 3):

0	1	2	3	4	5	6	7
36	18	10	*27*	3	20	9	8
27	*18*	10	36	*3*	20	9	8
27	3	*10*	36	18	*20*	9	8
27	3	10	*36*	18	20	*9*	8
9	*3*	10	27	*18*	20	36	*8*
9	3	10	27	8	20	36	18

Inserting an Element in a Subarray

- When we consider element `i`, the other elements in its subarray are already sorted with respect to each other.

example for `i = 6`:
`(incr = 3)`

0	1	2	3	4	5	6	7
27	3	10	*36*	18	20	*9*	8

the other element's in 9's subarray (the 27 and 36)
are already sorted with respect to each other

- To insert element `i`:
  - make a copy of element `i`, storing it in the variable `toInsert`:

`toInsert` *9*

0	1	2	3	4	5	6	7
27	3	10	*36*	18	20	*9*	8

  - consider elements `i-incr`, `i-(2*incr)`, `i-(3*incr)`,...
    - if an element `>` `toInsert`, slide it right *within the subarray*
    - stop at the first element `<=` `toInsert`

`toInsert` *9*

0	1	2	3	4	5	6	7
	3	10	*27*	18	20	*36*	8

  - copy `toInsert` into the "hole":

0	1	2	3	4	
9	3	10	*27*	18	...

The Sequence of Increments

- Different sequences of decreasing increments can be used.
- Our version uses values that are one less than a power of two.
  - $2^k - 1$ for some k
  - 63, 31, 15, 7, 3, 1
  - can get to the next lower increment using integer division:

    ```
    incr = incr/2;
    ```
- Should avoid numbers that are multiples of each other.
  - otherwise, elements that are sorted with respect to each other in one pass are grouped together again in subsequent passes
    - repeat comparisons unnecessarily
    - get fewer of the large jumps that speed up later passes
  - example of a bad sequence: 64, 32, 16, 8, 4, 2, 1
    - what happens if the largest values are all in odd positions?

Implementation of Shell Sort

```
public static void shellSort(int[] arr) {
    int incr = 1;
    while (2 * incr <= arr.length)
        incr = 2 * incr;
    incr = incr - 1;

    while (incr >= 1) {
        for (int i = incr; i < arr.length; i++) {
            if (arr[i] < arr[i-incr]) {
                int toInsert = arr[i];

                int j = i;
                do {
                     arr[j] = arr[j-incr];
                     j = j - incr;
                } while (j > incr-1 &&
                   toInsert < arr[j-incr]);

                arr[j] = toInsert;
            }
        }
        incr = incr/2;
    }
}
```

(If you replace `incr` with `1` in the for-loop, you get the code for insertion sort.)

Time Analysis of Shell Sort

- Difficult to analyze precisely
  - typically use experiments to measure its efficiency
- With a bad interval sequence, it's $O(n^2)$ in the worst case.
- With a good interval sequence, it's better than $O(n^2)$.
  - at least $O(n^{1.5})$ in the average and worst case
  - some experiments have shown average-case running times of $O(n^{1.25})$ or even $O(n^{7/6})$
- Significantly better than insertion or selection for large n:

n	n^2	$n^{1.5}$	$n^{1.25}$
10	100	31.6	17.8
100	10,000	1000	316
10,000	100,000,000	1,000,000	100,000
10^6	10^{12}	10^9	3.16×10^7

- We've wrapped insertion sort in another loop and increased its efficiency! The key is in the larger jumps that Shell sort allows.

Sorting by Exchange I: Bubble Sort

- Perform a sequence of passes through the array.
- On each pass: proceed from left to right, swapping adjacent elements if they are out of order.
- Larger elements "bubble up" to the end of the array.
- At the end of the kth pass, the k rightmost elements are in their final positions, so we don't need to consider them in subsequent passes.
- Example:

	0	1	2	3
	28	24	27	18
after the first pass:	24	27	18	*28*
after the second:	24	18	*27*	*28*
after the third:	18	*24*	*27*	*28*

Implementation of Bubble Sort

```
public class Sort {
    ...
    public static void bubbleSort(int[] arr) {
        for (int i = arr.length - 1; i > 0; i--) {
            for (int j = 0; j < i; j++) {
                if (arr[j] > arr[j+1])
                    swap(arr, j, j+1);
            }
        }
    }
}
```

- One for-loop nested in another:
  - the inner loop performs a single pass
  - the outer loop governs the number of passes, and the ending point of each pass

Time Analysis of Bubble Sort

- Comparisons: the kth pass performs ______ comparisons,

  so we get `C(n) =`

- Moves: depends on the contents of the array
  - in the worst case:
  - in the best case:
- Running time:

Sorting by Exchange II: Quicksort

- Like bubble sort, quicksort uses an approach based on exchanging out-of-order elements, but it's more efficient.
- A recursive, divide-and-conquer algorithm:
  - *divide:* rearrange the elements so that we end up with two subarrays that meet the following criterion:

    each element in the left array <= each element in the right array

    example:

12	8	14	4	6	13

6	8	4	14	12	13

  - *conquer:* apply quicksort recursively to the subarrays, stopping when a subarray has a single element
  - *combine:* nothing needs to be done, because of the criterion used in forming the subarrays

Partitioning an Array Using a Pivot

- The process that quicksort uses to rearrange the elements is known as *partitioning* the array.
- Partitioning is done using a value known as the *pivot.*
- We rearrange the elements to produce two subarrays:
  - left subarray: all values <= pivot
  - right subarray: all values >= pivot

  equivalent to the criterion on the previous page.

7	15	4	9	6	18	9	12

partition using a pivot of 9

7	9	4	6	9	18	15	12

all values <= 9 all values >= 9

- Our approach to partitioning is one of several variants.
- Partitioning is useful in its own right.
  ex: find all students with a GPA > 3.0.

Possible Pivot Values

- First element or last element
  - risky, can lead to terrible worst-case behavior
  - especially poor if the array is almost sorted

4	8	14	12	6	18

pivot = 18

4	8	14	12	6	18

- Middle element (what we will use)
- Randomly chosen element
- Median of three elements
  - left, center, and right elements
  - three randomly selected elements
  - taking the median of three decreases the probability of getting a poor pivot

Partitioning an Array: An Example

`arr`

first							last
7	15	4	*9*	6	18	9	12

`pivot = 9`

- Maintain indices `i` and `j`, starting them "outside" the array:

  `i = first - 1`
  `j = last + 1`

i									j
	7	15	4	9	6	18	9	12	

- *Find* "out of place" elements:
  - increment i until `arr[i] >= pivot`
  - decrement j until `arr[j] <= pivot`

	i					j	
7	15	4	9	6	18	9	12

- *Swap* `arr[i]` and `arr[j]`:

	i					j	
7	9	4	9	6	18	15	12

Partitioning Example (cont.)

from prev. page:

	i					j	
7	9	4	9	6	18	15	12

- Find:

			i	j			
7	9	4	9	6	18	15	12

- Swap:

			i	j			
7	9	4	6	9	18	15	12

- Find:

			j	i			
7	9	4	6	9	18	15	12

and now the indices have crossed, so we return `j`.

- Subarrays: left = `arr[first:j]`, right = `arr[j+1:last]`

first			j	i			last
7	9	4	6	9	18	15	12

Partitioning Example 2

- Start (pivot = 13):

i									j
	24	5	2	*13*	18	4	20	19	

- Find:

i					j		
24	5	2	13	18	4	20	19

- Swap:

i					j		
4	5	2	13	18	24	20	19

- Find:

			i j				
4	5	2	13	18	24	20	19

and now the indices are equal, so we return `j`.

- Subarrays:

			i j				
4	5	2	13	18	24	20	19

Partitioning Example 3 (done together)

- Start (pivot = 5):

i j

4	14	7	*5*	2	19	26	6

- Find:

4	14	7	5	2	19	26	6

partition() Helper Method

```
private static int partition(int[] arr, int first, int last)
{
    int pivot = arr[(first + last)/2];
    int i = first - 1;  // index going left to right
    int j = last + 1;   // index going right to left
    while (true) {
        do {
            i++;
        } while (arr[i] < pivot);
        do {
            j--;
        } while (arr[j] > pivot);
        if (i < j)
            swap(arr, i, j);
        else
            return j;   // arr[j] = end of left array
    }
}
```

Implementation of Quicksort

```
public static void quickSort(int[] arr) {
    qSort(arr, 0, arr.length - 1);
}

private static void qSort(int[] arr, int first, int last) {
    int split = partition(arr, first, last);

    if (first < split)
        qSort(arr, first, split);      // left subarray
    if (last > split + 1)
        qSort(arr, split + 1, last);   // right subarray
}
```

Counting Students: Divide and Conquer

- Everyone stand up.
- You will each carry out the following algorithm:

```
count = 1;
while (you are not the only person standing) {
    find another person who is standing
    if (your first name < other person's first name)
        sit down (break ties using last names)
    else
        count = count + the other person's count
}
if (you are the last person standing)
    report your final count
```

Counting Students: Divide and Conquer (cont.)

- At each stage of the "joint algorithm", the problem size is divided in half.

☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺
☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺
☺ ☺ ☺ ☺
☺ ☺
☺

- How many stages are there as a function of the number of students, n?

- This approach benefits from the fact that you perform the algorithm *in parallel* with each other.

A Quick Review of Logarithms

- `log`$_b$n = the exponent to which b must be raised to get n
  - $\log_b n = p$ if $b^p = n$
  - examples: $\log_2 8 = 3$ because $2^3 = 8$
    $\log_{10} 10000 = 4$ because $10^4 = 10000$
- Another way of looking at logs:
  - let's say that you repeatedly divide n by b (using integer division)
  - $\log_b n$ is an upper bound on the number of divisions needed to reach 1
  - example: $\log_2 18$ is approx. 4.17
    18/2 = 9 9/2 = 4 4/2 = 2 2/2 = 1

A Quick Review of Logs (cont.)

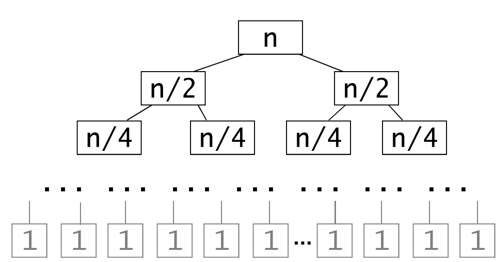
- If the number of operations performed by an algorithm is proportional to $\log_b n$ for any base b, we say it is a $O(\log n)$ algorithm – dropping the base.
- $\log_b n$ grows much more slowly than n

n	$\log_2 n$
2	1
1024 (1K)	10
1024*1024 (1M)	20

- Thus, for large values of n:
  - a $O(\log n)$ algorithm is much faster than a $O(n)$ algorithm
  - a $O(n \log n)$ algorithm is much faster than a $O(n^2)$ algorithm
- We can also show that an $O(n \log n)$ algorithm is faster than a $O(n^{1.5})$ algorithm like Shell sort.

Time Analysis of Quicksort

- Partitioning an array requires n comparisons, because each element is compared with the pivot.
- *best case:* partitioning always divides the array in half
  - repeated recursive calls give:



<u>*comparisons*</u>

n

$2*(n/2) = n$

$4*(n/4) = n$

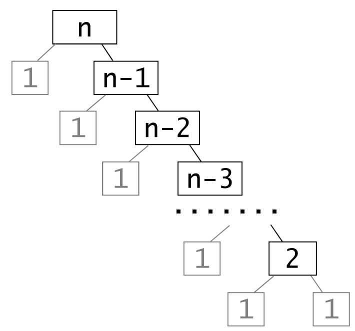
...

0

  - at each "row" except the bottom, we perform n comparisons
  - there are ________ rows that include comparisons
  - $C(n) = ?$
  - Similarly, $M(n)$ and running time are both ___________

Time Analysis of Quicksort (cont.)

- *worst case:* pivot is always the smallest or largest element
  - one subarray has `1` element, the other has `n - 1`
  - repeated recursive calls give:



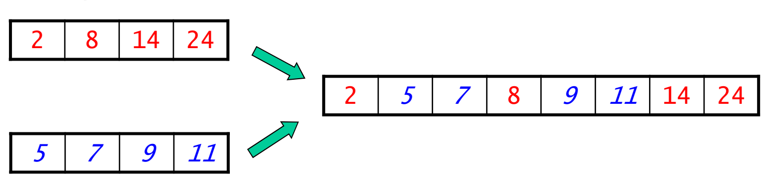
comparisons
n
n-1
n-2
n-3
...
2

  - $C(n) = \sum_{i=2}^{n} i = O(n^2)$. $M(n)$ and run time are also $O(n^2)$.

- *average case* is harder to analyze
  - $C(n) > n \log_2 n$, but it's still $O(n \log n)$

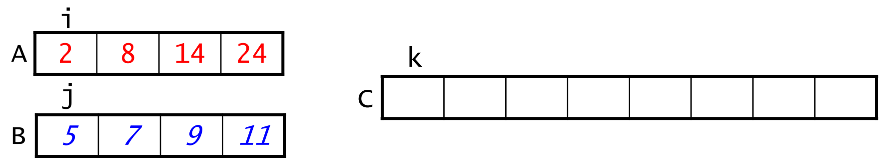
Mergesort

- All of the comparison-based sorting algorithms that we've seen thus far have sorted the array in place.
  - used only a small amount of additional memory
- Mergesort is a sorting algorithm that requires an additional temporary array of the same size as the original one.
  - it needs $O(n)$ additional space, where n is the array size
- It is based on the process of *merging* two sorted arrays into a single sorted array.
  - example:

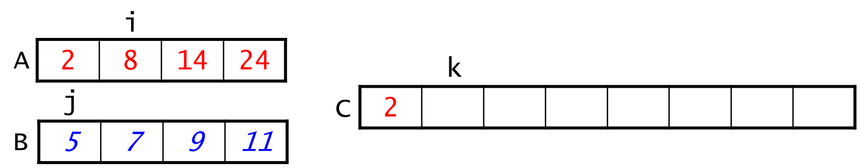


Merging Sorted Arrays

- To merge sorted arrays A and B into an array C, we maintain three indices, which start out on the first elements of the arrays:

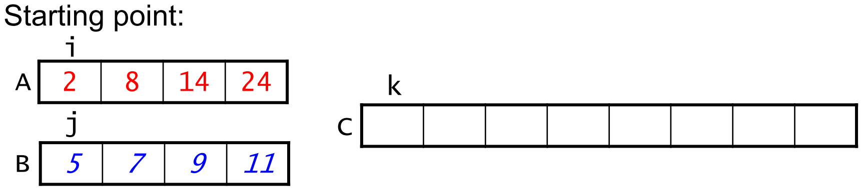


- We repeatedly do the following:
  - compare A[i] and B[j]
  - copy the smaller of the two to C[k]
  - increment the index of the array whose element was copied
  - increment k

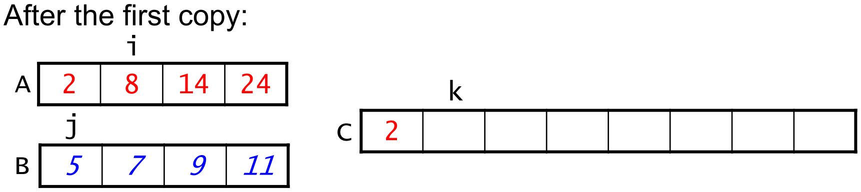


Merging Sorted Arrays (cont.)

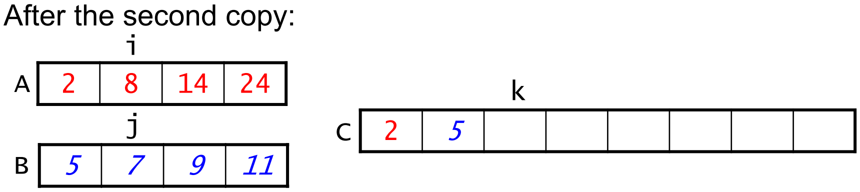
- Starting point:



- After the first copy:

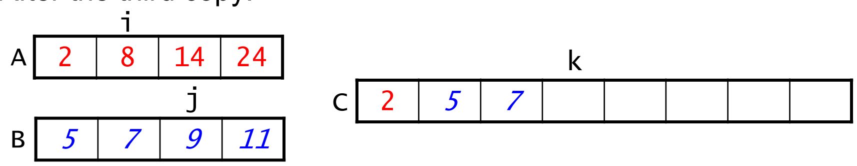


- After the second copy:

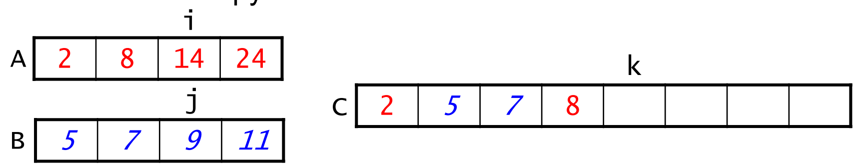


Merging Sorted Arrays (cont.)

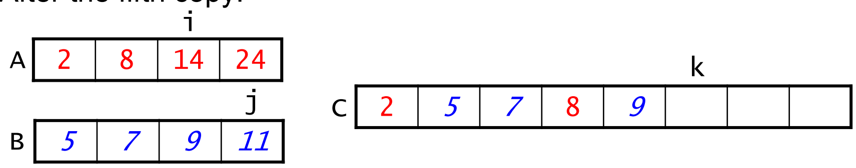
- After the third copy:



- After the fourth copy:

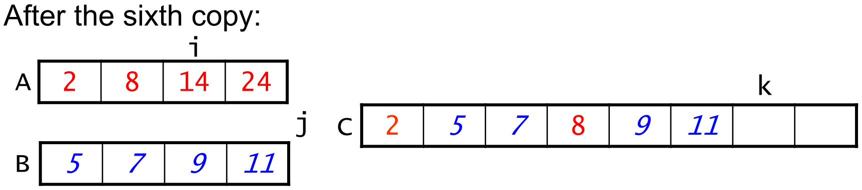


- After the fifth copy:

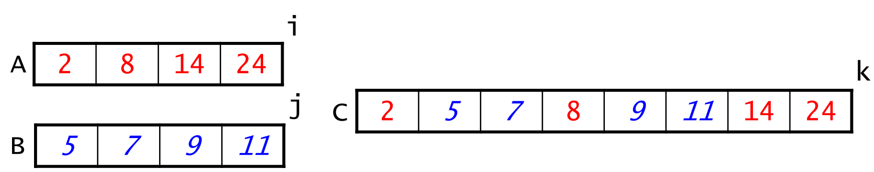


Merging Sorted Arrays (cont.)

- After the sixth copy:

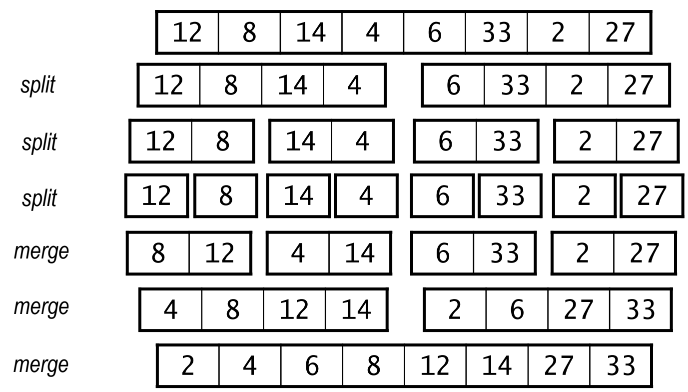


- There's nothing left in B, so we simply copy the remaining elements from A:



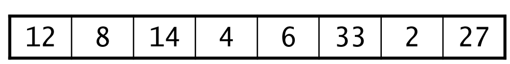
Divide and Conquer

- Like quicksort, mergesort is a divide-and-conquer algorithm.
  - *divide:* split the array in half, forming two subarrays
  - *conquer:* apply mergesort recursively to the subarrays, stopping when a subarray has a single element
  - *combine:* merge the sorted subarrays

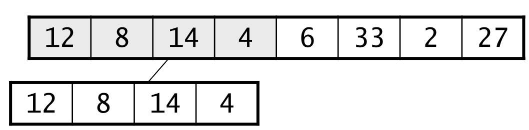


Tracing the Calls to Mergesort

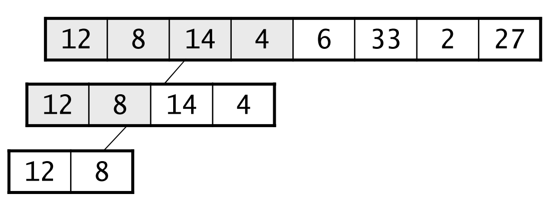
the initial call is made to sort the entire array:



split into two 4-element subarrays, and make a recursive call to sort the left subarray:

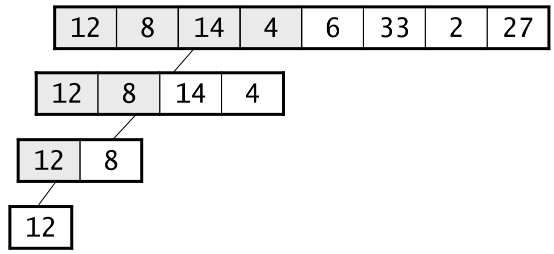


split into two 2-element subarrays, and make a recursive call to sort the left subarray:

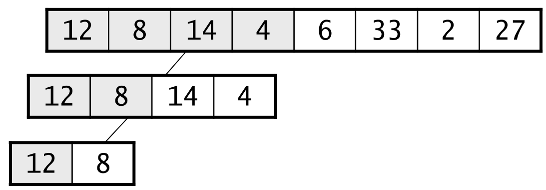


Tracing the Calls to Mergesort

split into two 1-element subarrays, and make a recursive call to sort the left subarray:

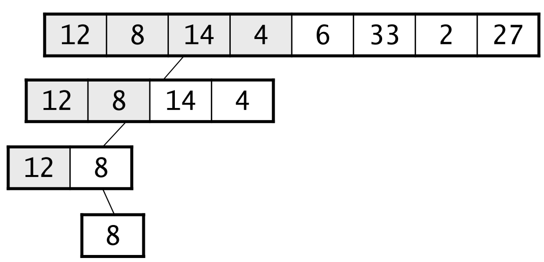


base case, so return to the call for the subarray {12, 8}:

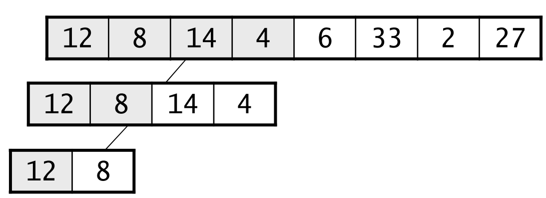


Tracing the Calls to Mergesort

make a recursive call to sort its right subarray:

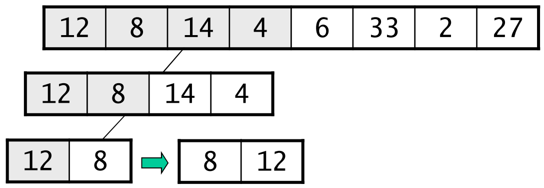


base case, so return to the call for the subarray {12, 8}:

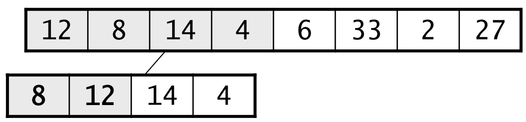


Tracing the Calls to Mergesort

merge the sorted halves of {12, 8}:

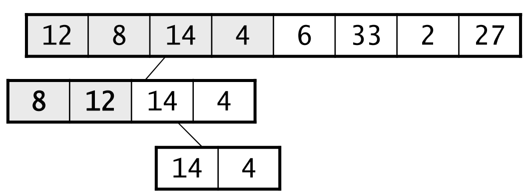


end of the method, so return to the call for the 4-element subarray, which now has a sorted left subarray:

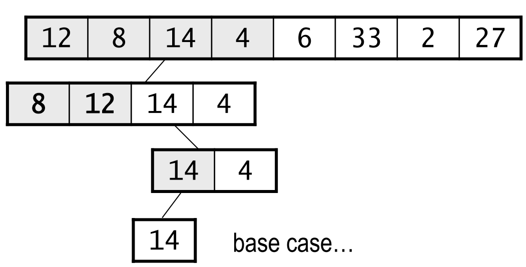


Tracing the Calls to Mergesort

make a recursive call to sort the right subarray of the 4-element subarray

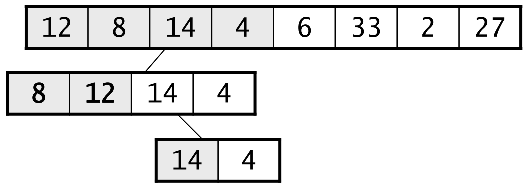


split it into two 1-element subarrays, and make a recursive call to sort the left subarray:

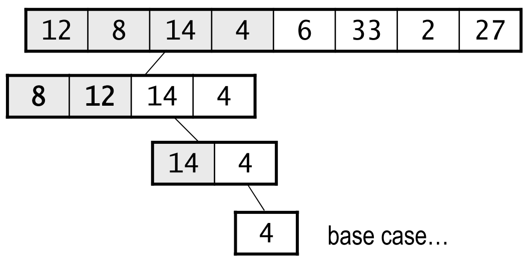


Tracing the Calls to Mergesort

return to the call for the subarray {14, 4}:



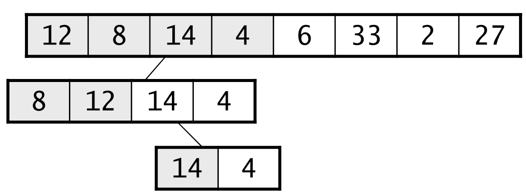
make a recursive call to sort its right subarray:



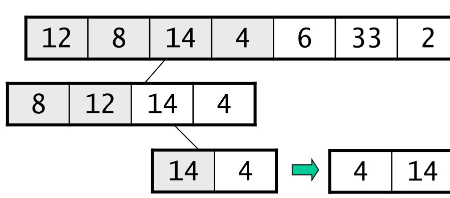
base case…

Tracing the Calls to Mergesort

return to the call for the subarray {14, 4}:



merge the sorted halves of {14, 4}:



Tracing the Calls to Mergesort

end of the method, so return to the call for the 4-element subarray, which now has two sorted 2-element subarrays:

12	8	14	4	6	33	2	27

8	12	4	14

merge the 2-element subarrays:

12	8	14	4	6	33	2	27

8	12	4	14

⇨

4	8	12	14

Tracing the Calls to Mergesort

end of the method, so return to the call for the original array, which now has a sorted left subarray:

4	**8**	**12**	**14**	6	33	2	27

perform a similar set of recursive calls to sort the right subarray. here's the result:

4	8	12	14	**2**	**6**	**27**	**33**

finally, merge the sorted 4-element subarrays to get a fully sorted 8-element array:

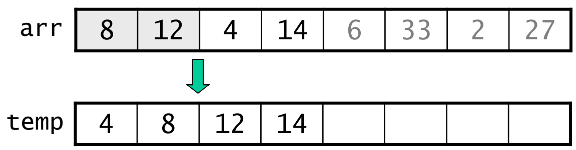
4	8	12	14	2	6	27	33

⇩

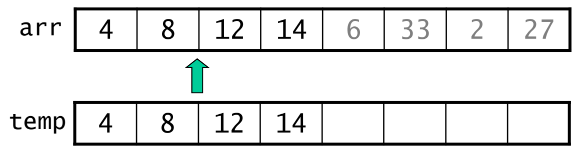
2	4	6	8	12	14	27	33

Implementing Mergesort

- One approach is to create new arrays for each new set of subarrays, and to merge them back into the array that was split.
- Instead, we'll create a temp. array of the same size as the original.
  - pass it to each call of the recursive mergesort method
  - use it when merging subarrays of the original array:



  - after each merge, copy the result back into the original array:



A Method for Merging Subarrays

```
private static void merge(int[] arr, int[] temp,
  int leftStart, int leftEnd, int rightStart, int rightEnd) {
    int i = leftStart;    // index into left subarray
    int j = rightStart;   // index into right subarray
    int k = leftStart;    // index into temp

    while (i <= leftEnd && j <= rightEnd) {
        if (arr[i] < arr[j])
            temp[k++] = arr[i++];
        else
            temp[k++] = arr[j++];
    }

    while (i <= leftEnd)
        temp[k++] = arr[i++];

    while (j <= rightEnd)
        temp[k++] = arr[j++];

    for (i = leftStart; i <= rightEnd; i++)
        arr[i] = temp[i];
}
```

Methods for Mergesort

- We use a wrapper method to create the temp. array, and to make the initial call to a separate recursive method:

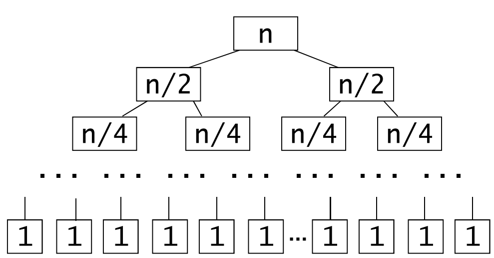
```
public static void mergeSort(int[] arr) {
    int[] temp = new int[arr.length];
    mSort(arr, temp, 0, arr.length - 1);
}
```

- Let's implement the recursive method together:

```
private static void mSort(int[] arr, int[] temp,
  int start, int end) {







}
```

Time Analysis of Mergesort

- Merging two halves of an array of size n requires 2n moves. Why?
- Mergesort repeatedly divides the array in half, so we have the following call tree (showing the sizes of the arrays):



moves
2n
2*2*(n/2) = 2n
4*2*(n/4) = 2n
...

  - at all but the last level of the call tree, there are 2n moves
  - how many levels are there?
  - `M(n)` = ?
  - `C(n)` = ?

Summary: Comparison-Based Sorting Algorithms

algorithm	best case	avg case	worst case	extra memory
selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Shell sort	$O(n \log n)$	$O(n^{1.5})$	$O(n^{1.5})$	$O(1)$
bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(1)$
mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$

- Insertion sort is best for nearly sorted arrays.
- Mergesort has the best worst-case complexity, but requires extra memory – and moves to and from the temp array.
- Quicksort is comparable to mergesort in the average case. With a reasonable pivot choice, its worst case is seldom seen.
- Use `SortCount.java` to experiment.

Comparison-Based vs. Distributive Sorting

- Until now, all of the sorting algorithms we have considered have been *comparison-based:*
  - treat the keys as wholes (comparing them)
  - don't "take them apart" in any way
  - all that matters is the relative order of the keys, not their actual values.
- No comparison-based sorting algorithm can do better than $O(n \log_2 n)$ on an array of length n.
  - $O(n \log_2 n)$ is a *lower bound* for such algorithms.
- *Distributive* sorting algorithms do more than compare keys; they perform calculations on the actual values of individual keys.
- Moving beyond comparisons allows us to overcome the lower bound.
  - tradeoff: use more memory.

Distributive Sorting Example: Radix Sort

- Relies on the representation of the data as a sequence of **m** quantities with **k** possible values.
- Examples:

	m	k
• integer in range 0 ... 999	3	10
• string of 15 upper-case letters	15	26
• 32-bit integer	32	2 (in binary)
	4	256 (as bytes)

- Strategy: Distribute according to the last element in the sequence, then concatenate the results:

    33 41 12 24 31 14 13 42 34

    get: 41 31 | 12 42 | 33 13 | 24 14 34

- Repeat, moving back one digit each time:

    get: | | |

Analysis of Radix Sort

- Recall that we treat the values as a sequence of `m` quantities with `k` possible values.
- Number of operations is $O($`n*m`$)$ for an array with `n` elements
    - better than $O($`n log n`$)$ when `m < log n`
- Memory usage increases as `k` increases.
    - `k` tends to increase as `m` decreases
    - tradeoff: increased speed requires increased memory usage

Big-O Notation Revisited

- We've seen that we can group functions into classes by focusing on the fastest-growing term in the expression for the number of operations that they perform.
  - e.g., an algorithm that performs $n^2/2 - n/2$ operations is a $O(n^2)$-time or quadratic-time algorithm
- Common classes of algorithms:

name	example expressions	big-O notation
constant time	$1, 7, 10$	$O(1)$
logarithmic time	$3\log_{10}n,\ \log_2 n + 5$	$O(\log n)$
linear time	$5n,\ 10n - 2\log_2 n$	$O(n)$
$n\log n$ time	$4n\log_2 n,\ n\log_2 n + n$	$O(n \log n)$
quadratic time	$2n^2 + 3n,\ n^2 - 1$	$O(n^2)$
cubic time	$n^2 + 3n^3,\ 5n^3 - 5$	$O(n^3)$
exponential time	$2^n,\ 5e^n + 2n^2$	$O(c^n)$
factorial time	$3n!,\ 5n + n!$	$O(n!)$

slower

How Does the Number of Operations Scale?

- Let's say that we have a problem size of 1000, and we measure the number of operations performed by a given algorithm.
- If we double the problem size to 2000, how would the number of operations performed by an algorithm increase if it is:
  - $O(n)$-time
  - $O(n^2)$-time
  - $O(n^3)$-time
  - $O(\log_2 n)$-time
  - $O(2^n)$-time

How Does the Actual Running Time Scale?

- How much time is required to solve a problem of size n?
  - assume that each operation requires 1 μsec (1×10^{-6} sec)

time function	problem size (n) 10	20	30	40	50	60
n	.00001 s	.00002 s	.00003 s	.00004 s	.00005 s	.00006 s
n^2	.0001 s	.0004 s	.0009 s	.0016 s	.0025 s	.0036 s
n^5	.1 s	3.2 s	24.3 s	1.7 min	5.2 min	13.0 min
2^n	.001 s	1.0 s	17.9 min	12.7 days	35.7 yrs	36,600 yrs

- sample computations:
  - when n = 10, an n^2 algorithm performs 10^2 operations.
    $10^2 * (1 \times 10^{-6}$ sec) = .0001 sec
  - when n = 30, a 2^n algorithm performs 2^{30} operations.
    $2^{30} * (1 \times 10^{-6}$ sec) = 1073 sec = 17.9 min

What's the Largest Problem That Can Be Solved?

- What's the largest problem size n that can be solved in a given time T? (again assume 1 μsec per operation)

time function	time available (T) 1 min	1 hour	1 week	1 year
n	60,000,000	3.6×10^9	6.0×10^{11}	3.1×10^{13}
n^2	7745	60,000	777,688	5,615,692
n^5	35	81	227	500
2^n	25	31	39	44

- sample computations:
  - 1 hour = 3600 sec
    that's enough time for $3600/(1 \times 10^{-6}) = 3.6 \times 10^9$ operations
    - n^2 algorithm:
      $n^2 = 3.6 \times 10^9 \rightarrow n = (3.6 \times 10^9)^{1/2} = 60{,}000$
    - 2^n algorithm:
      $2^n = 3.6 \times 10^9 \rightarrow n = \log_2(3.6 \times 10^9) \sim= 31$

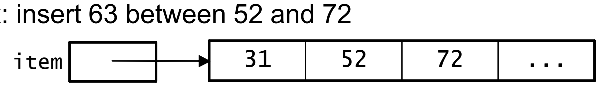
Linked Lists

Computer Science E-22
Harvard Extension School

David G. Sullivan, Ph.D.

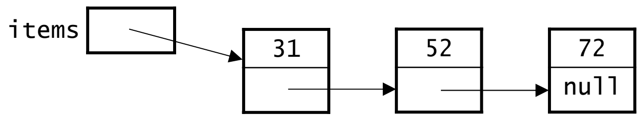
Representing a Sequence of Data

- Sequence – an ordered collection of items (position matters)
  - we will look at several types: lists, stacks, and queues
- Most common representation = an array
- Advantages of using an array:
  - easy and efficient access to *any* item in the sequence
    - `item[i]` gives you the item at position i
    - every item can be accessed in constant time
    - this feature of arrays is known as *random access*
  - very compact (but can waste space if positions are empty)
- Disadvantages of using an array:
  - have to specify an initial array size and resize it as needed
  - difficult to insert/delete items at arbitrary positions
    - ex: insert 63 between 52 and 72

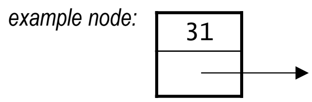


Alternative Representation: A Linked List

- Example:



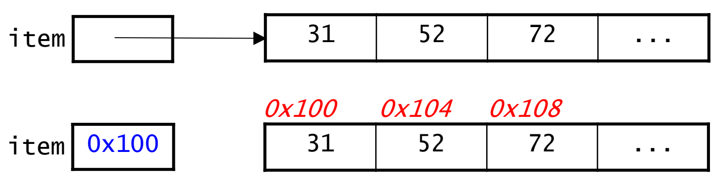
- A linked list stores a sequence of items in separate *nodes*.
- Each node contains:
  - a single item
  - a "link" (i.e., a reference) to the node containing the next item



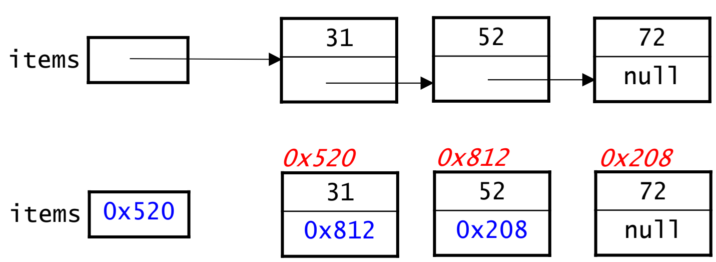
- The last node in the linked list has a link value of `null`.
- The linked list as a whole is represented by a variable that holds a reference to the first node (e.g., `items` in the example above).

Arrays vs. Linked Lists in Memory

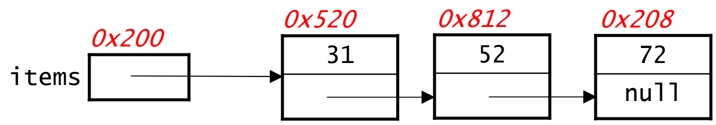
- In an array, the elements occupy consecutive memory locations:



- In a linked list, each node is a distinct object on the heap. The nodes do *not* have to be next to each other in memory. That's why we need the links to get from one node to the next.



Linked Lists in Memory

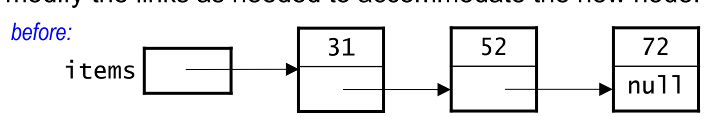


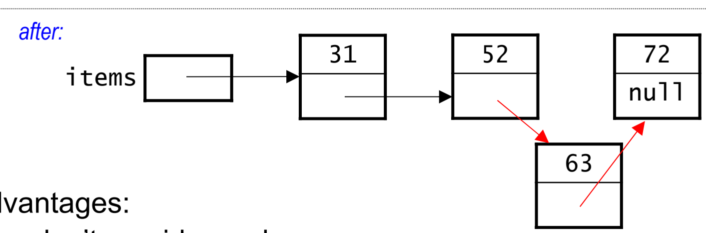
- Here's how the above linked list might actually look in memory:

0x200	`0x520`	← *the variable* `items`
0x204		
0x208	`72`	*the last node*
0x212	`null`	
0x216		
…	…	
0x520	`31`	*the first node*
0x524	`0x812`	
0x528		
…	…	
0x812	`52`	*the second node*
0x816	`0x208`	

Features of Linked Lists

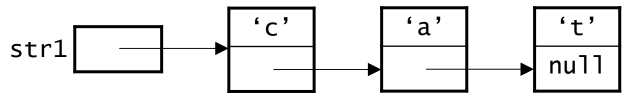
- They can grow without limit (provided there is enough memory).
- Easy to insert/delete an item – no need to "shift over" other items.
  - for example, to insert 63 between 52 and 72, we just modify the links as needed to accommodate the new node:





- Disadvantages:
  - they don't provide random access
    - need to "walk down" the list to access an item
  - the links take up additional memory

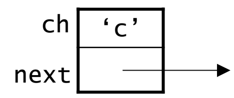
A String as a Linked List of Characters



- Each node in the linked list represents one character.
- Java class for this type of node:

```
public class StringNode {
    private char ch;
    private StringNode next;
    …
}
```

same type as the node itself!



- The string as a whole will be represented by a variable that holds a reference to the node containing the first character.

  example:

```
StringNode str1;   // shown in the diagram above
```

- Alternative approach: use another class for the string as a whole.

```
public class LLString {
    StringNode first;
    …
```

(we will *not* do this for strings)

A String as a Linked List (cont.)

- An empty string will be represented by a null value.

  example:

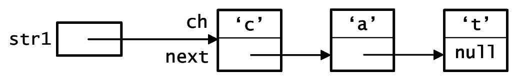
```
StringNode str2 = null;
```

- We will use *static* methods that take the string as a parameter.
  - e.g., we will write `length(str1)` instead of `str1.length()`
  - outside the class, need the class name: `StringNode.length(str1)`
- This approach is necessary so that the methods can handle empty strings.
  - if `str1 == null`, `length(str1)` will work, but `str1.length()` will throw a `NullPointerException`
- Constructor for our `StringNode` class:

```
public StringNode(char c, StringNode n) {
    ch = c;
    next = n;
}
```

A Linked List Is a Recursive Data Structure

- Recursive definition of a linked list: a linked list is either
  a) empty or
  b) a single node, followed by a linked list
- Viewing linked lists in this way allows us to write recursive methods that operate on linked lists.

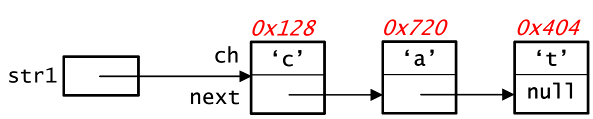


- Example: length of a string
  length of "`cat`" = 1 + the length of "`at`"
  length of "`at`" = 1 + the length of "`t`"
  length of "`t`" = 1 + the length of the empty string (which = 0)
- In Java:

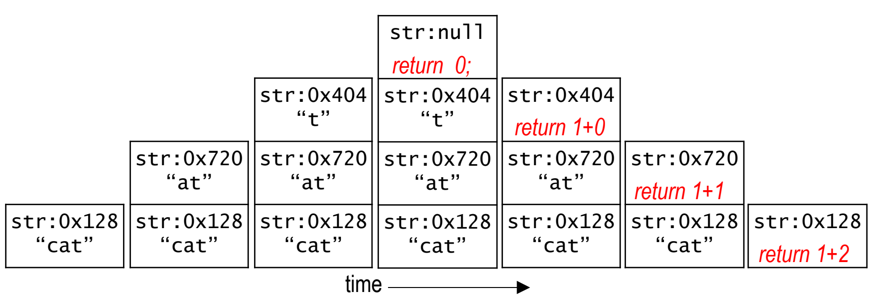
```
public static int length(StringNode str) {
    if (str == null)
        return 0;
    else
        return 1 + length(str.next);
}
```

Tracing `length()`

```
public static int length(StringNode str) {
    if (str == null)
        return 0;
    else
        return 1 + length(str.next);
}
```



- Example: `StringNode.length(str1)`

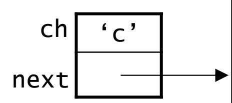


Getting the Node at Position i in a Linked List

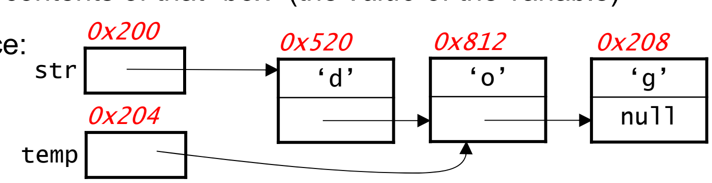
- `getNode(str, i)` – a private helper method that returns a reference to the ith node in the linked list (i == 0 for the first node)
- Recursive approach:
  node at position 2 in the linked list representing “linked”
  = node at position 1 in the linked list representing “inked”
  = node at position 0 in the linked list representing “nked”
  (return a reference to the node containing ‘n’)
- We’ll write the method together:

```
private static StringNode getNode(StringNode str, int i) {



}
```

Review of Variables



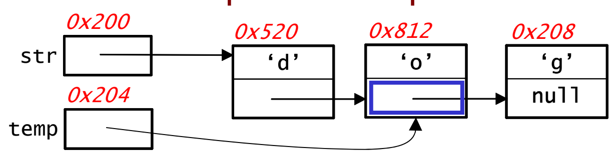
- A variable or variable expression represents both:
  - a “box” or location in memory (the *address* of the variable)
  - the contents of that “box” (the *value* of the variable)
- Practice:




```
StringNode str;   // points to the first node
StringNode temp;  // points to the second node
```

expression	*address*	*value*
`str`	0x200	0x520 (reference to the ‘d’ node)
`str.ch`		
`str.next`		

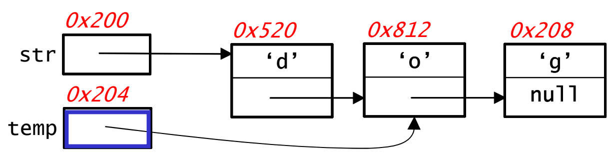
More Complicated Expressions



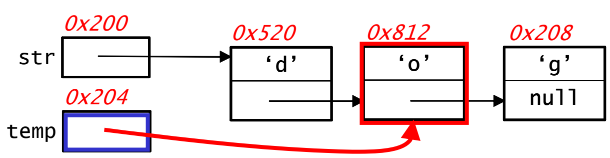
- Example: `temp.next.ch`
- Start with the start of the expression: `temp.next`
  It represents the `next` field of the node to which `temp` refers.
  - address =
  - value =
- Next, consider `temp.next.ch`
  It represents the `ch` field of the node to which `temp.next` refers.
  - address =
  - value =

Dereferencing a Reference

- Each dot causes us to *dereference* the reference represented by the expression preceding the dot.
- Consider again `temp.next.ch`
- Start with temp: `temp.next.ch`

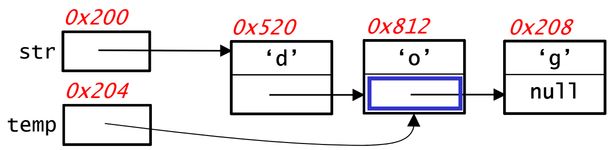


- Dereference: `temp.next.ch`

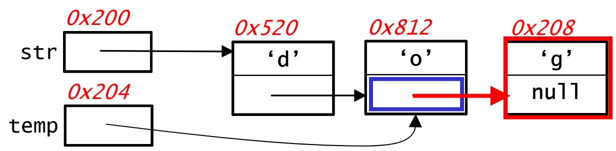


Dereferencing a Reference (cont.)

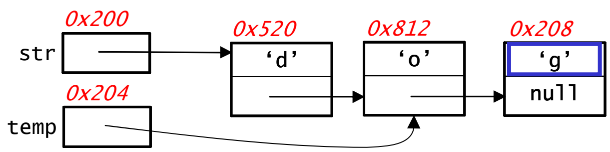
- Get the `next` field: `temp.next.ch`



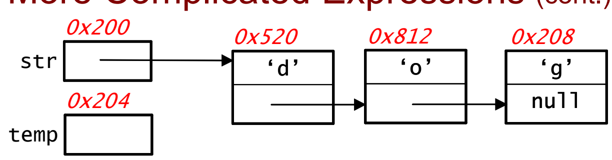
- Dereference: `temp.next.ch`



- Get the `ch` field: `temp.next.ch`



More Complicated Expressions (cont.)



- Here's another example: `str.next.next`
  - address = ?
  - value = ?

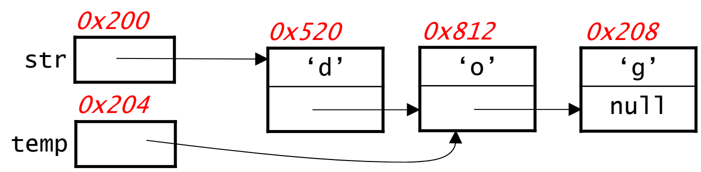
Assignments Involving References

- An assignment of the form

  `var1 = var2;`

  takes the *value* of `var2` and copies it into the location in memory given by the *address* of `var1`.

- Practice:

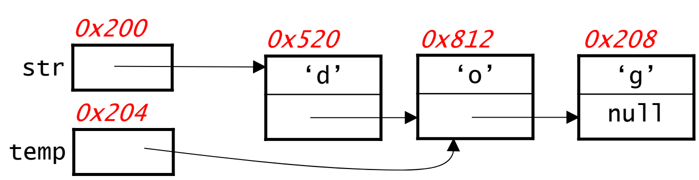


- What happens if we do the following?
  1) `str.next = temp.next;`

  2) `temp = temp.next;`

Assignments Involving References (cont.)

- Beginning with the original diagram, if `temp` didn't already refer to the `'o'` node, what assignment would we need to perform to make it refer to that node?



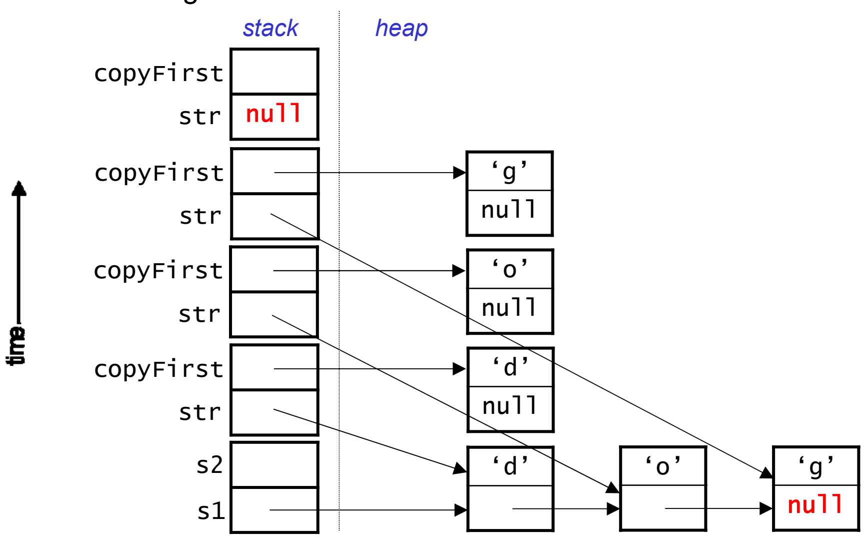
Creating a Copy of a Linked List

- `copy(str)` – create a copy of `str` and return a reference to it
- Recursive approach:
  - base case: if `str` is empty, return `null`
  - else: copy the first character
    make a recursive call to copy the rest

```
public static StringNode copy(StringNode str) {
    if (str == null)          // base case
        return null;

    // create the first node of the copy, copying the
    // first character into it
    StringNode copyFirst = new StringNode(str.ch, null);

    // make a recursive call to get a copy the rest and
    // store the result in the first node's next field
    copyFirst.next = copy(str.next);

    return copyFirst;
}
```

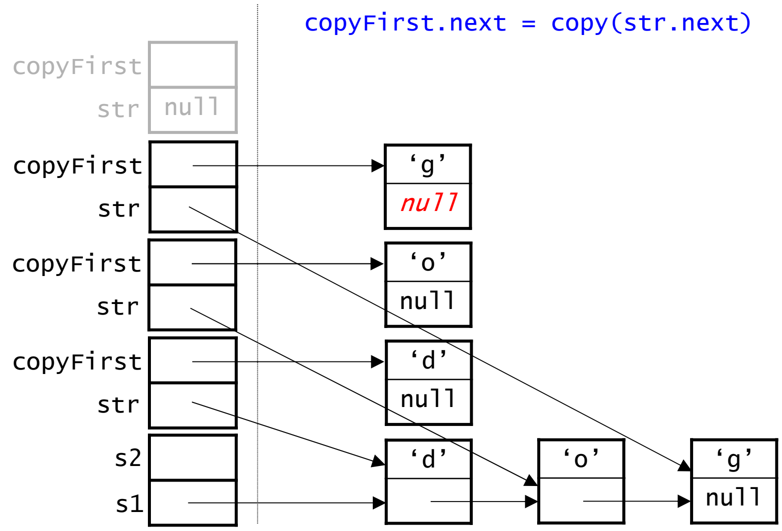
Tracing copy(): part I

- Example: `StringNode s2 = StringNode.copy(s1);`
- The stack grows as a series of recursive calls are made:



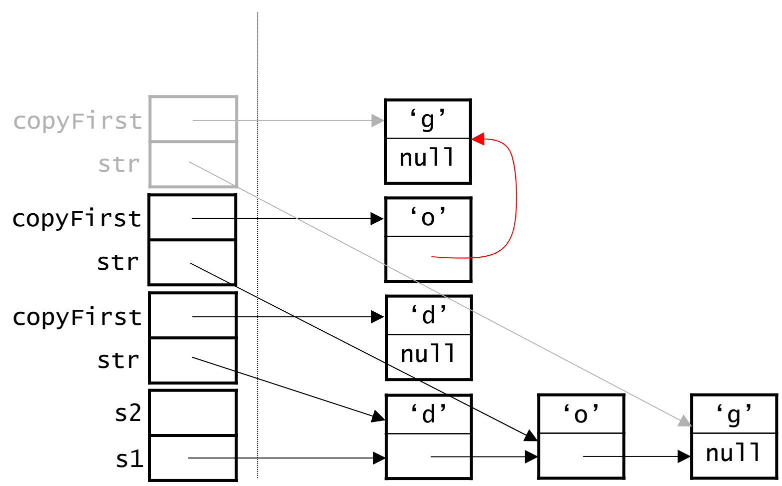
Tracing copy(): part II

- The base case is reached, so the final recursive call returns `null`.
- This return value is stored in the `next` field of the `'g'` node:



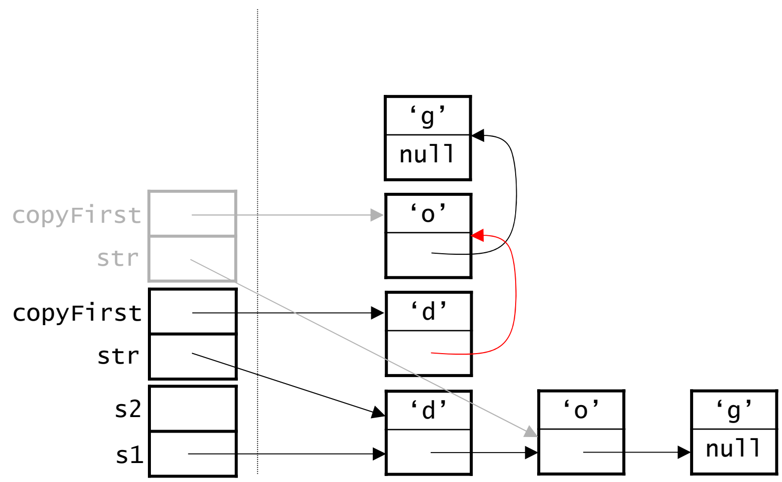
Tracing copy(): part III

- The recursive call that created the `'g'` node now completes, returning a reference to the `'g'` node.
- This return value is stored in the `next` field of the `'o'` node:



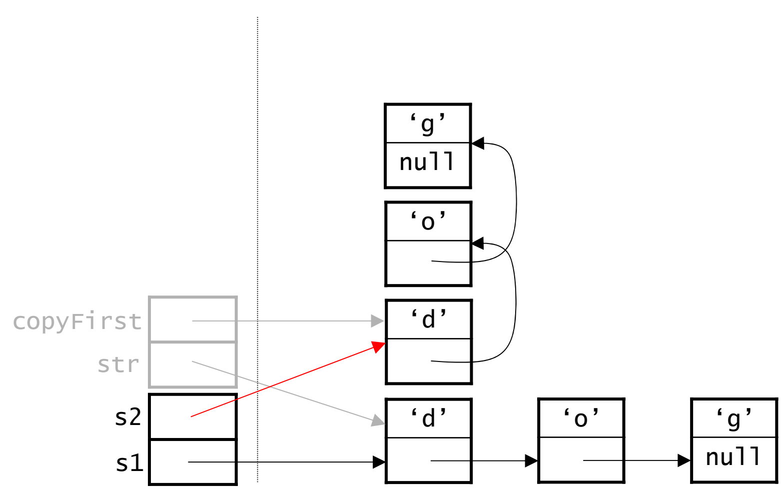
Tracing `copy()`: part IV

- The recursive call that created the `'o'` node now completes, returning a reference to the `'o'` node.
- This return value is stored in the `next` field of the `'d'` node:



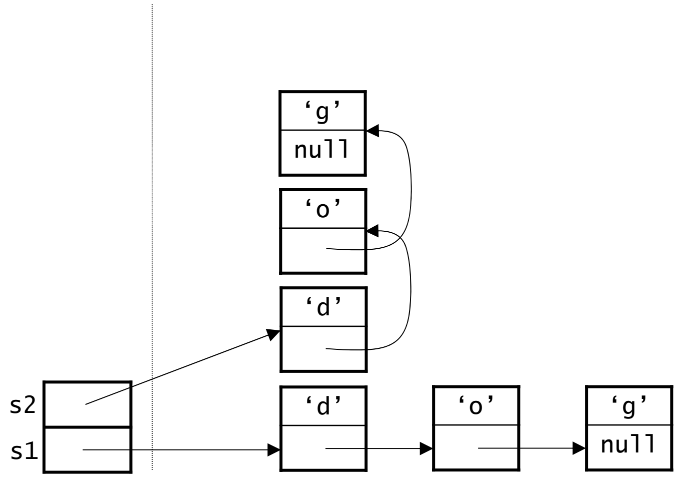
Tracing `copy()`: part V

- The original call (which created the `'d'` node) now completes, returning a reference to the `'d'` node.
- This return value is stored in `s2`:



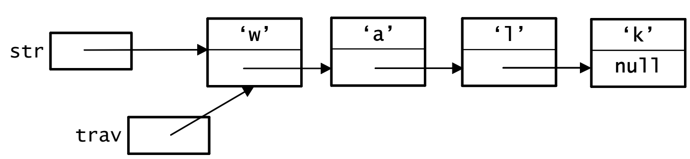
Tracing copy(): Final Result

- `StringNode s2 = StringNode.copy(s1);`
- `s2` now holds a reference to a linked list that is a copy of the linked list to which `s1` holds a reference.



Using Iteration to Traverse a Linked List

- Many tasks require us to traverse or “walk down” a linked list.
- We’ve already seen methods that use recursion to do this.
- It can also be done using iteration (for loops, while loops, etc.).
- We make use of a variable (call it `trav`) that keeps track of where we are in the linked list.

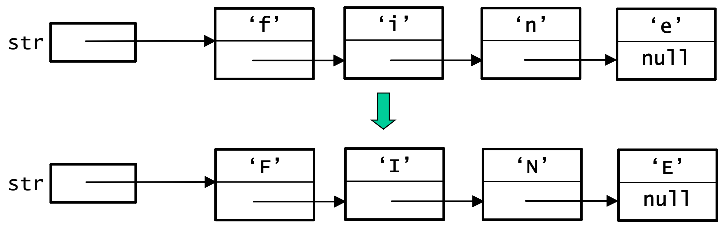


- Template for traversing an entire linked list:

```
StringNode trav = str;     // start with the first node
while (trav != null) {
    // usually do something here
    trav = trav.next;    // move trav down one node
}
```

Example of Iterative Traversal

- `toUpperCase(str)`: converting `str` to all upper-case letters

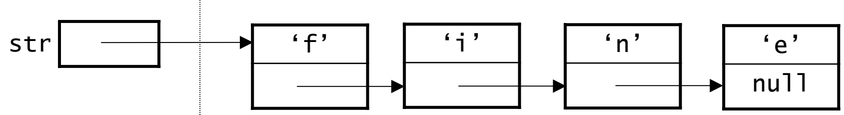


- Java method:

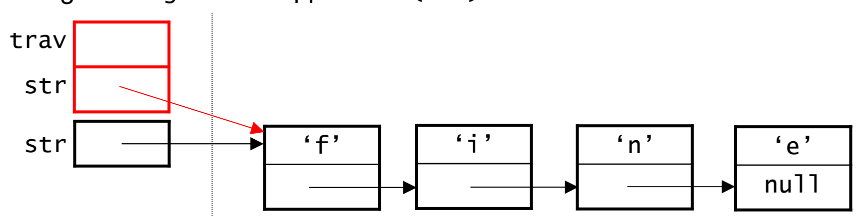
```
public static void toUpperCase(StringNode str) {
    StringNode trav = str;
    while (trav != null) {
        trav.ch = Character.toUpperCase(trav.ch);
        trav = trav.next;
    }
}
```

(makes use of the `toUpperCase()` method from Java's built-in `Character` class)

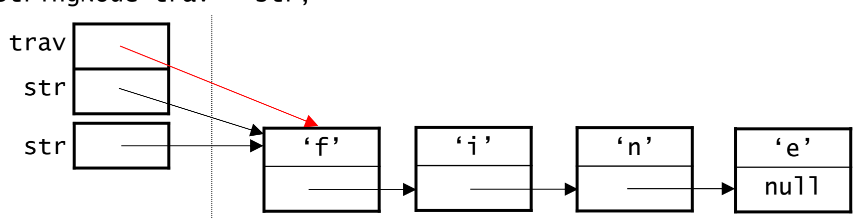
Tracing `toUpperCase()`: Part I



Calling `StringNode.toUpperCase(str)` adds a stack frame to the stack:

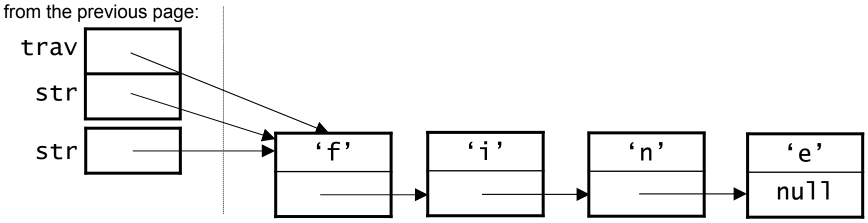


`StringNode trav = str;`



Tracing `toUpperCase()`: Part II

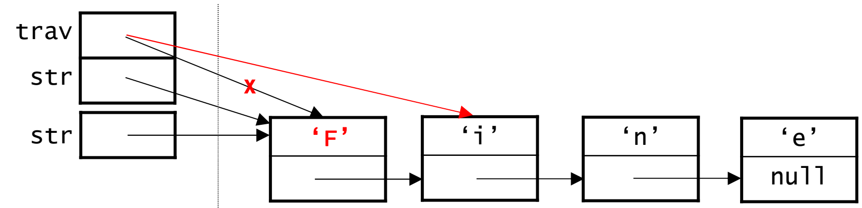
from the previous page:



we enter the `while` loop:

```
while (trav != null) {
    trav.ch = Character.toUpperCase(trav.ch);
    trav = trav.next;
}
```

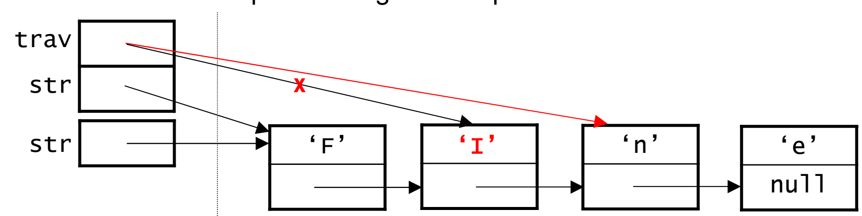
results of the first pass through the loop:



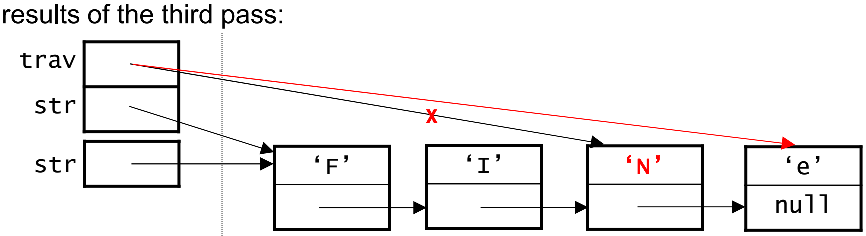
Tracing `toUpperCase()`: Part III

```
while (trav != null) {
    trav.ch = Character.toUpperCase(trav.ch);
    trav = trav.next;
}
```

results of the second pass through the loop:



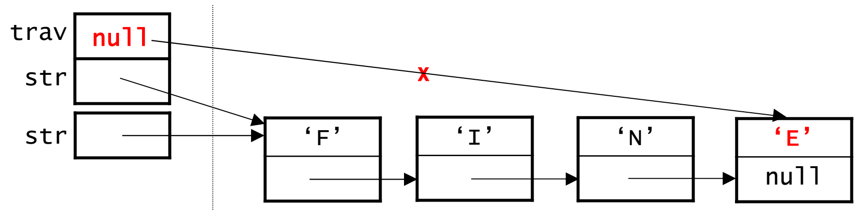
results of the third pass:



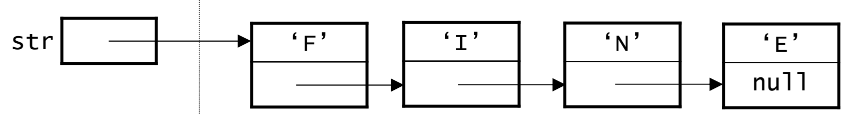
Tracing `toUpperCase()`: Part IV

```
while (trav != null) {
    trav.ch = Character.toUpperCase(trav.ch);
    trav = trav.next;
}
```

results of the fourth pass through the loop:



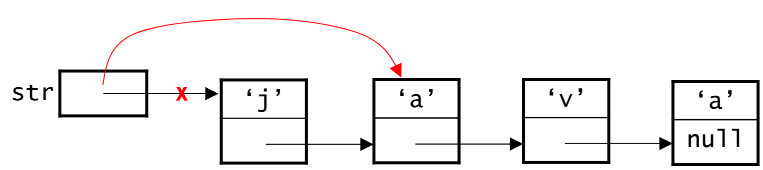
and now `trav == null`, so we break out of the loop and return:



Deleting the Item at Position i

- Special case: `i == 0` (deleting the first item)
- Update our reference to the first node by doing:

```
str = str.next;
```

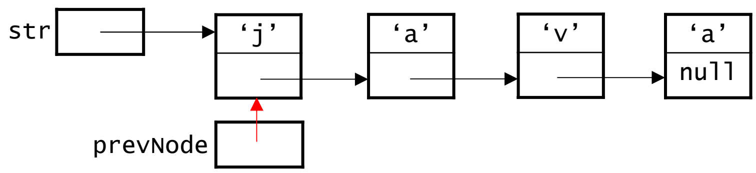


Deleting the Item at Position i (cont.)

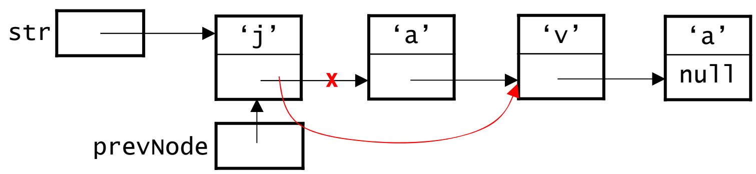
- General case: `i > 0`
- First obtain a reference to the *previous* node:

```
StringNode prevNode = getNode(i - 1);
```

(example for i == 1*)*

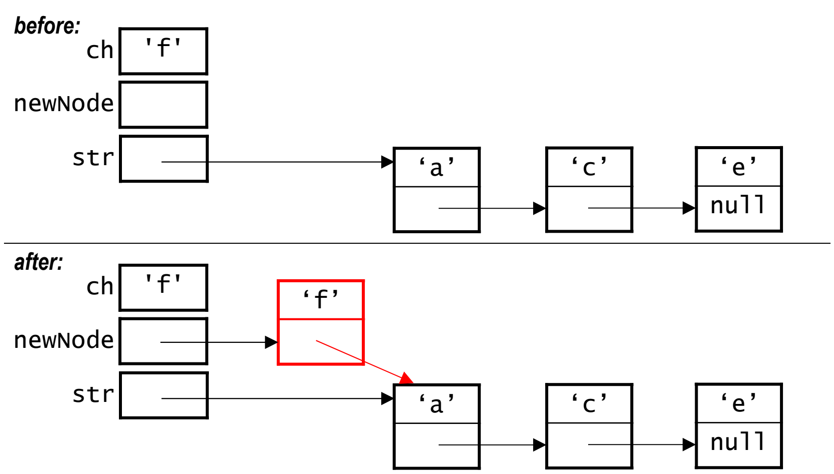


- What remains to be done? (to get the picture below)



Inserting an Item at Position i

- Special case: `i == 0` (insertion at the front of the list)
- What line of code will *create* the new node?

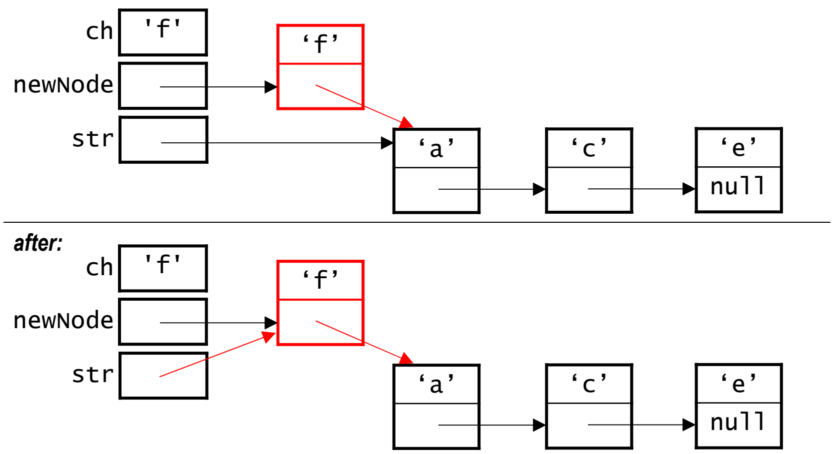



```
StringNode newNode = new StringNode(_______, _______);
```

Inserting an Item at Position i (cont.)

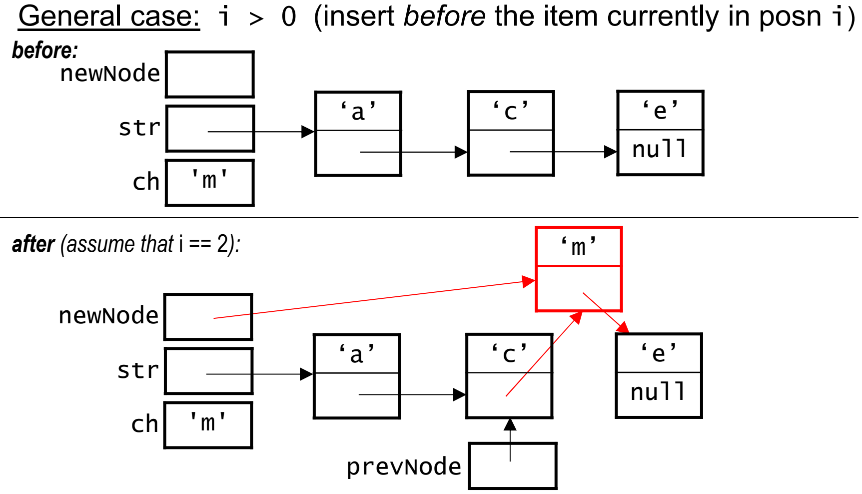
- Special case: `i == 0` (continued)
- What line of code will *insert* the new node?

before (result of previous slide):



Inserting an Item at Position i (cont.)

- General case: `i > 0` (insert *before* the item currently in posn `i`)




```
StringNode prevNode = getNode(i – 1);
StringNode newNode = new StringNode(ch, ________________);
___________________________________ // one more line
```

Returning a Reference to the First Node

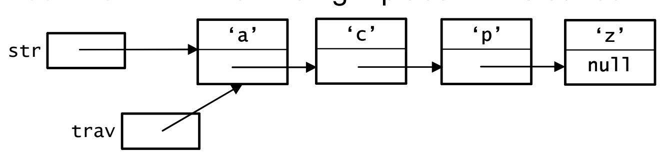
- Both `deleteChar()` and `insertChar()` return a reference to the first node in the linked list. For example:

```
private static StringNode deleteChar(StringNode str, int i) {
    …
    if (i == 0)                 // case 1
        str = str.next;
    else {                      // case 2
        StringNode prevNode = getNode(str, i-1);
        if (prevNode != null && prevNode.next != null)
            prevNode.next = prevNode.next.next;
        …
    }
    return str;
}
```

- They do so because the first node may change.
- Invoke as follows: `str = StringNode.deleteChar(str, i);`
  `str = StringNode.insertChar(str, i, ch);`
- If the first node changes, `str` will point to the new first node.

Using a "Trailing Reference" During Traversal

- When traversing a linked list, using a single `trav` reference isn't always good enough.
- Ex: insert `ch = 'n'` at the right place in this *sorted* linked list:



- Traverse the list to find the right position:

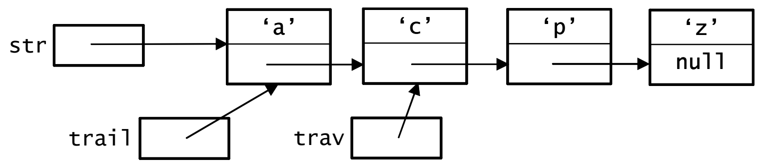
```
StringNode trav = str;
while (trav != null && trav.ch < ch)
    trav = trav.next;
```

- When we exit the loop, where will `trav` point? Can we insert `'n'`?
- The following changed version doesn't work either. Why not?

```
StringNode trav = str;
while (trav != null && trav.next.ch < ch)
    trav = trav.next;
```

Using a “Trailing Reference” (cont.)

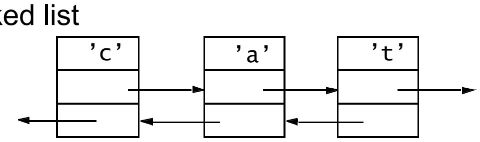
- To get around the problem seen on the previous page, we traverse the list using two different references:
  - `trav`, which we use as before
  - `trail`, which stays one node behind `trav`




```
StringNode trav = str;
StringNode trail = null;
while (trav != null && trav.ch < ch) {
    trail = trav;
    trav = trav.next;
}
// if trail == null, insert at the front of the list
// else insert after the node to which trail refers
```

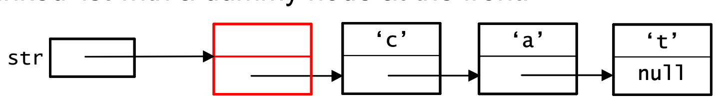
Other Variants of Linked Lists

- Doubly linked list



  - add a `prev` reference to each node -- refers to the previous node
  - allows us to “back up” from a given node

- Linked list with a dummy node at the front:



  - the dummy node doesn’t contain a data item
  - it eliminates the need for special cases to handle insertion and deletion at the front of the list
    - more on this in the next set of notes

Lists, Stacks, and Queues

Computer Science E-22
Harvard Extension School

David G. Sullivan, Ph.D.

Representing a Sequence: Arrays vs. Linked Lists

- Sequence – an ordered collection of items (position matters)
  - we will look at several types: lists, stacks, and queues
- Can represent any sequence using an array *or* a linked list

	array	***linked list***
representation in memory	elements occupy consecutive memory locations	nodes can be at arbitrary locations in memory; the links connect the nodes together
advantages		
disadvantages		

A List as an Abstract Data Type

- list = a sequence of items that supports at least the following functionality:
  - accessing an item at an arbitrary position in the sequence
  - adding an item at an arbitrary position
  - removing an item at an arbitrary position
  - determining the number of items in the list (the list's *length*)
- ADT: specifies *what* a list will do, without specifying the implementation

Review: Specifying an ADT Using an Interface

- Recall that in Java, we can use an interface to specify an ADT:

```
public interface List {
    Object getItem(int i);
    boolean addItem(Object item, int i);
    int length();
    …
}
```

- We make any implementation of the ADT a class that implements the interface:

```
public class MyList implements List {
    ...
```

- This approach allows us to write code that will work with different implementations of the ADT:

```
public static void processList(List l) {
    for (int i = 0; i < l.length(); i++) {
        Object item = l.getItem(i);
        …
```

Our List Interface

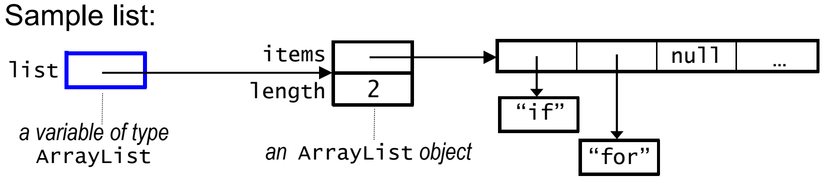
```
public interface List {
    Object getItem(int i);
    boolean addItem(Object item, int i);
    Object removeItem(int i);
    int length();
    boolean isFull();
}
```

- We include an `isFull()` method to test if the list already has the maximum number of items
- Recall that all methods in an interface are assumed to be public.
- The actual interface definition includes comments that describe what each method should do.

Implementing a List Using an Array

```
public class ArrayList implements List {
    private Object[] items;
    private int length;

    public ArrayList(int maxSize) {
        items = new Object[maxSize];
        length = 0;
    }

    public int length() {
        return length;
    }

    public boolean isFull() {
        return (length == items.length);
    }
    ...
}
```

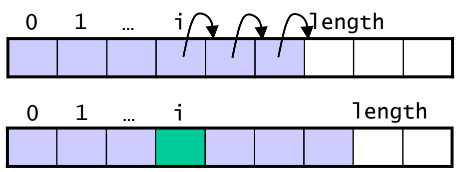
- Sample list:



Adding an Item to an `ArrayList`

- Adding at position i (shifting items i, i+1,   to the right by one):

```
public boolean addItem(Object item, int i) {
    if (i < 0 || i > length)
        throw new IndexOutOfBoundsException();
    if (isFull())
        return false;

    // make room for the new item
    for (int j = length - 1; j >= i; j--)
        items[j + 1] = items[j];

    items[i] = item;
    length++;
    return true;
}
```



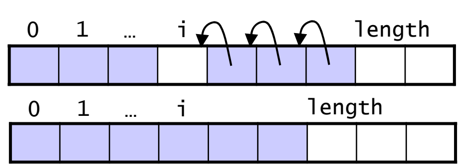
Other `ArrayList` Methods

- Getting item i:

```
public Object getItem(int i) {
    if (i < 0 || i >= length)
        throw new IndexOutOfBoundsException();
    return items[i];
}
```

- Removing item i (shifting items i+1, i+2,   to the left by one):

```
public Object removeItem(int i) {
    if (i < 0 || i >= length)
        throw new IndexOutOfBoundsException();




}
```



Converting an ArrayList to a String

- The toString() method is designed to allow objects to be displayed in a human-readable format.
- This method is called implicitly when you attempt to print an object or when you perform string concatenation:

```
ArrayList l = new ArrayList();
System.out.println(l);
String str = “My list: ” + l;
System.out.println(str);
```

- A default version of this method is inherited from the Object class.
  - returns a String consisting of the type of the object and a hash code for the object.
- It usually makes sense to override the default version.

toString() Method for the ArrayList Class

```
public String toString() {
    String str = "{";
    if (length > 0) {
        for (int i = 0; i < length - 1; i++)
            str = str + items[i] + ", ";
        str = str + items[length - 1];
    }
    str = str + "}"
    return str;
}
```

- Produces a string of the following form:

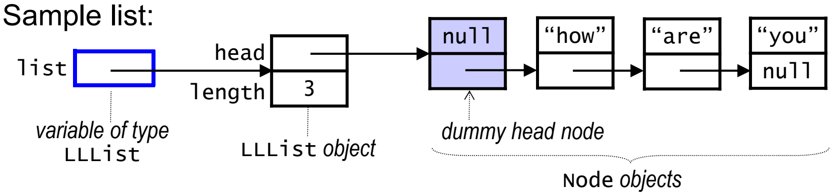
```
{items[0], items[1], … }
```

- Why is the last item added outside the loop?
- Why do we need the if statement?

Implementing a List Using a Linked List

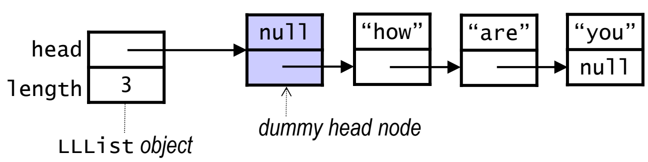
```
public class LLList implements List {
    private Node head;     // dummy head node
    private int length;
    …
}
```

- Sample list:

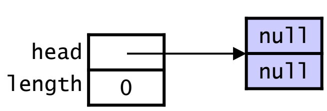


- Differences from the linked list we used for strings:
  - we "embed" the linked list inside another class
    - users of our `LLList` class will never actually touch the nodes
    - users of our `StringNode` class hold a reference to the first node
  - we use a dummy head node
  - we use instance methods instead of static methods
    - `myList.length()` instead of `length(myList)`

Using a Dummy Head Node

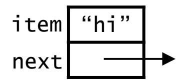


- The dummy head node is always at the front of the linked list.
  - like the other nodes in the linked list, it's of type `Node`
  - it does *not* store an item
  - it does *not* count towards the length of the list
- An empty `LLList` still has a dummy head node:



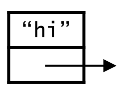
- Using a dummy head node allows us to avoid special cases when adding and removing nodes from the linked list.

An Inner Node Class

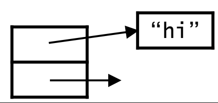



```
public class LLList implements List {
    private class Node {
        private Object item;
        private Node next;

        private Node(Object i, Node n) {
            item = i;
            next = n;
        }
    }
    ...
}
```

- We make `Node` an *inner class*, defining it within `LLList`.
  - allows the `LLList` methods to directly access `Node`'s private members, while restricting all other access
  - the compiler creates this class file: `LLList$Node.class`
- For simplicity, our diagrams show the items inside the nodes.



instead of



Other Details of Our `LLList` Class

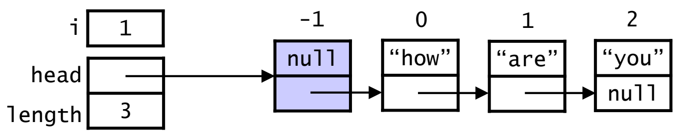
```
public class LLList implements List {
    private class Node {
        ...
    }

    private Node head;
    private int length;

    public LLList() {
        head = new Node(null, null);
        length = 0;
    }

    public boolean isFull() {
        return false;
    }
}
```

- Unlike `ArrayList`, there's no need to preallocate space for the items. The constructor simply creates the dummy head node.
- The linked list can grow indefinitely, so the list is never full!

Getting a Node

- Private helper method for getting node `i`
  - to get the dummy head node, use `i = -1`

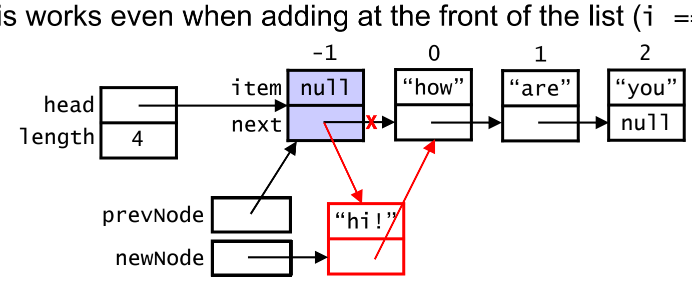
```
private Node getNode(int i) {
    // private method, so we assume i is valid!

    Node trav = __________;
    int travIndex = -1;
    while ( _________________ ) {
        travIndex++;
        ________________________;
    }
    return trav;
}
```



Adding an Item to an `LLList`

```
public boolean addItem(Object item, int i) {
    if (i < 0 || i > length)
        throw new IndexOutOfBoundsException();
    Node newNode = new Node(item, null);
    Node prevNode = getNode(i – 1);
    newNode.next = prevNode.next;
    prevNode.next = newNode;
    length++;
    return true;
}
```

- This works even when adding at the front of the list (`i == 0`):



`addItem()` Without a Dummy Head Node

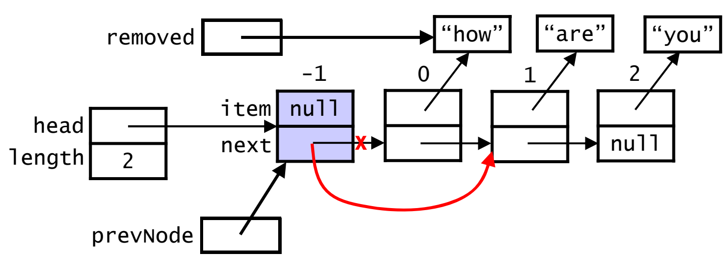
```
public boolean addItem(Object item, int i) {
    if (i < 0 || i > length)
        throw new IndexOutOfBoundsException();

    Node newNode = new Node(item, null);

    if (i == 0) {                    // case 1: add to front
        newNode.next = first;
        first = newNode;
    } else {                         // case 2: i > 0
        Node prevNode = getNode(i - 1);
        newNode.next = prevNode.next;
        prevNode.next = newNode;
    }

    length++;
    return true;
}
```

(instead of a reference named `head` *to the dummy head node, this implementation maintains a reference named* `first` *to the first node, which does hold an item).*

Removing an Item from an `LLList`

```
public Object removeItem(int i) {
    if (i < 0 || i >= length)
        throw new IndexOutOfBoundsException();

    Node prevNode = getNode(i - 1);
    Object removed = prevNode.next.item;
    ___________________________   // what line goes here?

    length--;
    return removed;
}
```

- This works even when removing the first item (`i == 0`):



toString() Method for the LLList Class

```
public String toString() {
    String str = "{";

    // what should go here?




    str = str + " }"

    return str;
}
```

Counting the Number of Occurrences of an Item

- One possible approach:

```
public class MyClass {
    public static int numOccur(List l, Object item) {
        int numOccur = 0;
        for (int i = 0; i < l.length(); i++) {
            Object itemAt = l.getItem(i);
            if (itemAt.equals(item))
                numOccur++;
        }
        return numOccur;
    } …
```

- Problem: for `LLList` objects, each call to `getItem()` starts at the head of the list and traverses to item `i`.
  - to access item 0, access 1 node
  - to access item 1, access 2 nodes
  - to access item i, access i+1 nodes
  - if length = n, total nodes accessed = 1 + 2 + … + n = $O(n^2)$

Solution 1: Make `numOccur()` an `LLList` Method

```
public class LLList {
    public int numOccur(Object item) {
        int numOccur = 0;
        Node trav = head.next;  // skip the dummy head node
        while (trav != null) {
            if (trav.item.equals(item))
                numOccur++;
            trav = trav.next;
        }
        return numOccur;
    } …
```

- Each node is only visited once, so the # of accesses = n = $O(n)$
- Problem: we can't anticipate all of the types of operations that users may wish to perform.
- We would like to give users the general ability to iterate over the list.

Solution 2: Give Access to the Internals of the List

- Make our private helper method `getNode()` a public method.
- Make `Node` a non-inner class and provide getter methods.
- This would allow us to do the following:

```
public class MyClass {
    public static int numOccur(LLList l, Object item) {
        int numOccur = 0;
        Node trav = l.getNode(0);
        while (trav != null) {
            Object itemAt = trav.getItem();
            if (itemAt.equals(item))
                numOccur++;
            trav = trav.getNext();
        }
        return numOccur;
    } …
}
```

- What's wrong with this approach?

Solution 3: Provide an Iterator

- An iterator is an object that provides the ability to iterate over a list *without* violating encapsulation.
- Our iterator class will implement the following interface:

```
public interface ListIterator {
    // Are there more items to visit?
    boolean hasNext();

    // Return next item and advance the iterator.
    Object next();
}
```

- The iterator starts out prepared to visit the first item in the list, and we use `next()` to access the items sequentially.
- Ex: position of the iterator is shown by the cursor symbol (|)

after the iterator `i` *is created:*	|	"do"		"we"		"go"	…
after calling `i.next()`*, which returns* "do":		"do"	|	"we"		"go"	…
after calling `i.next()`*, which returns* "we":		"do"		"we"	|	"go"	…

`numOccur()` Using an Iterator

```
public class MyClass {
    public static int numOccur(List l, Object item) {
        int numOccur = 0;
        ListIterator iter = l.iterator();
        while (iter.hasNext()) {
            Object itemAt = iter.next();
            if (itemAt.equals(item))
                numOccur++;
        }
        return numOccur;
    } …
}
```

- The `iterator()` method returns an iterator object that is ready to visit the first item in the list. (Note: we also need to add the header of this method to the `List` interface.)
- Note that `next()` does two things at once:
  - gets an item
  - advances the iterator.

Using an Inner Class for the Iterator

```
public class LLList {
    public ListIterator iterator() {
        return new LLListIterator();
    }

    private class LLListIterator implements ListIterator {
        private Node nextNode;
        private Node lastVisitedNode;

        public LLListIterator() {
            …
    }
}
```

- Using a inner class gives the iterator access to the list's internals.
- Because `LLListIterator` is a private inner class, methods outside `LLList` can't create `LLListIterator` objects or have variables that are declared to be of type `LLListIterator`.
- Other classes use the *interface name* as the declared type, e.g.:

```
ListIterator iter = l.iterator();
```

`LLListIterator` Implementation

```
private class LLListIterator implements ListIterator {
    private Node nextNode;
    private Node lastVisitedNode;

    public LLListIterator() {
        nextNode = head.next;    // skip over head node
        lastVisitedNode = null;
    }
    …
}
```

- Two instance variables:
  - `nextNode` keeps track of the next node to visit
  - `lastVisitedNode` keeps track of the most recently visited node
    - not needed by `hasNext()` and `next()`
    - what iterator operations might we want to add that *would* need this reference?

`LLListIterator` Implementation (cont.)

```
private class LLListIterator implements ListIterator {
    private Node nextNode;
    private Node lastVisitedNode;

    public LLListIterator() {
        nextNode = head.next;    // skip over dummy node
        lastVisitedNode = null;
    }

    public boolean hasNext() {
         return (nextNode != null);
    }

    public Object next() {
        if (nextNode == null)
            throw new NoSuchElementException();

        Object item = nextNode.item;
        lastVisited = nextNode;
        nextNode = nextNode.next;

        return item;
    }
}
```

More About Iterators

- In theory, we could write list-iterator methods that were methods of the list class itself.
- Instead, our list-iterator methods are encapsulated within an iterator object.
  - allows us to have multiple iterations active at the same time:

```
ListIterator i = l.iterator();
while (i.hasNext()) {
    ListIterator j = l.iterator();
    while (j.hasNext()) {
        …
```

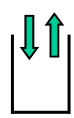
- Java's built-in *collection classes* all provide iterators.
  - `LinkedList`, `ArrayList`, etc.
  - the built-in `Iterator` interface specifies the iterator methods
    - they include `hasNext()` and `next()` methods like ours

Efficiency of the List Implementations

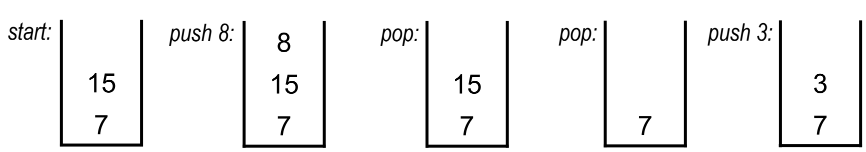
n = number of items in the list

	ArrayList	LLList
getItem()		
addItem()		
removeItem()		
space efficiency		

Stack ADT



- A stack is a sequence in which:
  - items can be added and removed only at one end (the *top*)
  - you can only access the item that is currently at the top
- Operations:
  - push: add an item to the top of the stack
  - pop: remove the item at the top of the stack
  - peek: get the item at the top of the stack, but don't remove it
  - isEmpty: test if the stack is empty
  - isFull: test if the stack is full
- Example: a stack of integers



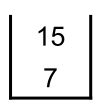
A Stack Interface: First Version

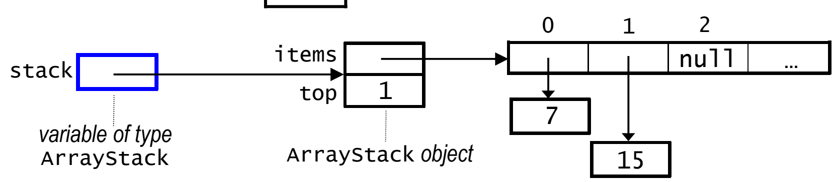
```
public interface Stack {
    boolean push(Object item);
    Object pop();
    Object peek();
    boolean isEmpty();
    boolean isFull();
}
```

- `push()` returns `false` if the stack is full, and `true` otherwise.
- `pop()` and `peek()` take no arguments, because we know that we always access the item at the top of the stack.
  - return `null` if the stack is empty.
- The interface provides no way to access/insert/delete an item at an arbitrary position.
  - encapsulation allows us to ensure that our stacks are manipulated only in ways that are consistent with what it means to be stack

Implementing a Stack Using an Array: First Version

```
public class ArrayStack implements Stack {
    private Object[] items;
    private int top;    // index of the top item

    public ArrayStack(int maxSize) {
        items = new Object[maxSize];
        top = -1;
    }
    ...
}
```

- Example: the stack  would be represented as follows:



- Items are added from left to right. The instance variable `top` stores the index of the item at the top of the stack.

Limiting a Stack to Objects of a Given Type

- We can do this by using a *generic* interface and class.
- Here is a generic version of our `Stack` interface:

```
public interface Stack<T> {
    boolean push(T item);
    T pop();
    T peek();
    boolean isEmpty();
    boolean isFull();
}
```

- It includes a *type variable* `T` in its header and body.
- This type variable is used as a placeholder for the actual type of the items on the stack.

A Generic `ArrayStack` Class

```
public class ArrayStack<T> implements Stack<T> {
    private T[] items;
    private int top;    // index of the top item
    …
    public boolean push(T object) {
        …
    }
    …
}
```

- Once again, a type variable `T` is used as a placeholder for the actual type of the items.
- When we create an `ArrayStack`, we specify the type of items that we intend to store in the stack:

```
ArrayStack<Integer> s1 = new ArrayStack<Integer>(10);
ArrayStack<String> s2 = new ArrayStack<String>(5);
ArrayStack<Object> s3 = new ArrayStack<Object>(20);
```

`ArrayStack` Constructor

- Java doesn't allow you to create an object or array using a type variable. Thus, we *cannot* do this:

```
public ArrayStack(int maxSize) {
    items = new T[maxSize];   // not allowed
    top = -1;
}
```

- To get around this limitation, we create an array of type `Object` and cast it to be an array of type `T`:

```
public ArrayStack(int maxSize) {
    items = (T[])new Object[maxSize];
    top = -1;
}
```

(This doesn't produce a `ClassCastException` at runtime, because in the compiled version of the class, `T` is replaced with `Object`.)

- The cast generates a compile-time warning, but we'll ignore it.
- Java's built-in `ArrayList` class takes this same approach.

More on Generics

- When a collection class uses the type `Object` for its items, we often need to use casting:

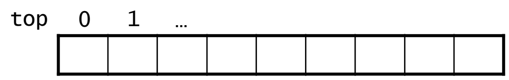
```
LLList list = new LLList();
list.addItem("hello");
list.addItem("world");
String item = (String)list.getItem(0);
```

- Using generics allows us to avoid this:

```
ArrayStack<String> s = new ArrayStack<String>;
s.push("hello");
s.push("world");
String item = s.pop();    // no casting needed
```

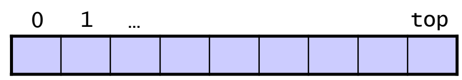
Testing if an `ArrayStack` is Empty or Full

- Empty stack:




```
public boolean isEmpty() {
    return (top == -1);
}
```

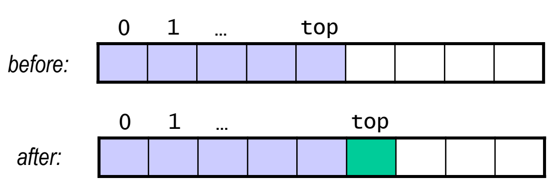
- Full stack:




```
public boolean isFull() {
    return (top == items.length - 1);
}
```

Pushing an Item onto an `ArrayStack`

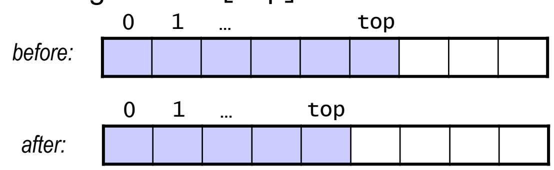
- We increment `top` before adding the item:




```
public boolean push(T item) {
    if (isFull())
        return false;
    top++;
    items[top] = item;
    return true;
}
```

ArrayStack pop() and peek()

- pop: need to get `items[top]` *before* we decrement `top`.




```
public T pop() {
    if (isEmpty())
        return null;
    T removed = items[top];
    items[top] = null;
    top--;
    return removed;
}
```

- peek just returns `items[top]` without decrementing `top`.

toString() Method for the ArrayStack Class

- Assume that we want the method to show us everything in the stack – returning a string of the form

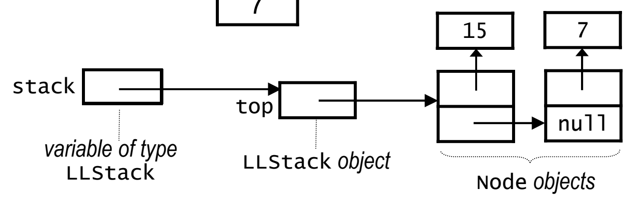
  `"{top, one-below-top, two-below-top, … bottom}"`

```
public String toString() {
    String str = "{";

    // what should go here?






    str = str + "}"

    return str;
}
```

Implementing a Generic Stack Using a Linked List

```
public class LLStack<T> implements Stack<T> {
    private Node top;    // top of the stack
    …
}
```

- Example: the stack 15 7 would be represented as follows:



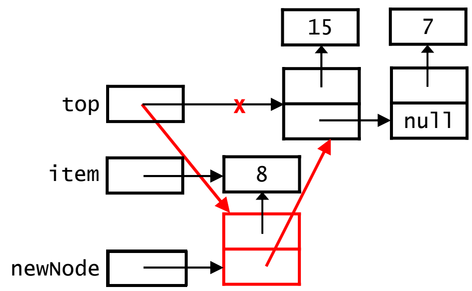
- Things worth noting:
  - our `LLStack` class needs only a single instance variable— a reference to the first node, which holds the top item
  - top item = leftmost item (vs. rightmost item in `ArrayStack`)
  - we don't need a dummy node, because we always insert at the front, and thus the insertion code is already simple

Other Details of Our `LLStack` Class

```
public class LLStack<T> implements Stack<T> {
    private class Node {
        private T item;
        private Node next;
        ...
    }
    private Node top;

    public LLStack() {
        top = null;
    }
    public boolean isEmpty() {
        return (top == null);
    }
    public boolean isFull() {
        return false;
    }
}
```

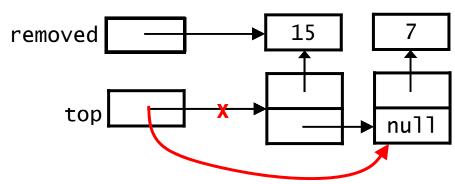
- The inner `Node` class uses the type parameter `T` for the item.
- We don't need to preallocate any memory for the items.
- The stack is never full!

LLStack.push




```
public boolean push(T item) {


}
```

LLStack pop() and peek()




```
public T pop() {
    if (isEmpty())
        return null;

    T removed = __________________;


}

public T peek() {
    if (isEmpty())
        return null;

    return top.item;
}
```

toString() Method for the LLStack Class

- Again, assume that we want a string of the form

  "{top, one-below-top, two-below-top, … bottom}"

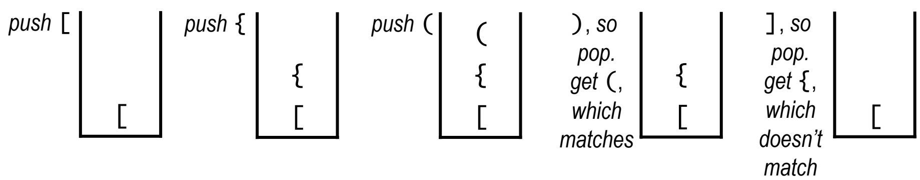
```
public String toString() {
    String str = "{";

    // what should go here?




    str = str + "}"

    return str;
}
```

Efficiency of the Stack Implementations

	ArrayStack	LLStack
push()	$O(1)$	$O(1)$
pop()	$O(1)$	$O(1)$
peek()	$O(1)$	$O(1)$
space efficiency	O(m) where m is the *anticipated* maximum number of items	O(n) where n is the number of items currently on the stack

Applications of Stacks

- The runtime stack in memory
- Converting a recursive algorithm to an iterative one by using a stack to emulate the runtime stack
- Making sure that delimiters (parens, brackets, etc.) are balanced:
  - push open (i.e., left) delimiters onto a stack
  - when you encounter a close (i.e., right) delimiter, pop an item off the stack and see if it matches
  - example: `5 * [3 + {(5 + 16 - 2)]`



- Evaluating arithmetic expressions (see textbooks)

An Example of Switching Between Implementations

- In the example code for this unit, there is a test program for each type of sequence:

  `ListTester.java`, `StackTester.java`, `QueueTester.java`

- Each test program uses a variable that has the appropriate *interface* as its type. For example:

```
Stack<String> myStack;
```

- The program asks you which implementation you want to test, and it calls the corresponding constructor:

```
if (type == 1)
    myStack = new ArrayStack<String>(10);
else if (type == 2)
    myStack = new LLStack<String>();
```

- This is an example of what principle of object-oriented programming?

Declared Type vs. Actual Type

- An object has two types that may or may not be the same.
  - declared type: type specified when declaring the variable
  - actual type: type specified when creating the object
- Consider again our `StackTester` program:

```
int type;
Stack<String> myStack;
Scanner in = new Scanner(System.in);
...
type = in.nextInt();
if (type == 1)
    myStack = new ArrayStack<String>(10);
else if (type == 2)
    myStack = new LLStack<String>();
```

- What is the declared type of `myStack`?
- What is its actual type?

Dynamic Binding

- Example of how `StackTester` tests the methods:

```
String item = myStack.pop();
```

- There are two different versions of the `pop` method, but we don't need two different sets of code to test them.
  - the line shown above will test whichever version of the method the user has specified!
- At runtime, the Java interpreter selects the version of the method that is appropriate to the *actual* type of `myStack`.
  - This is known as *dynamic binding.*
  - Why can't this selection be done by the compiler?

Determining if a Method Call is Valid

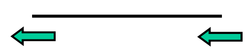
- The compiler uses the *declared* type of an object to determine if a method call is valid.
- Example:
  - assume that we add our `iterator()` method to `LLList` but do not add a header for it to the `List` interface
  - under that scenario, the following will *not* work:

```
List myList = new LLList();
ListIterator iter = myList.iterator();
```

- Because the declared type of `myList` is `List`, the compiler looks for that method in `List`.
  - if it's not there, the compiler will not compile the code.
- We can use a type cast to reassure the compiler:

```
ListIterator iter = ((LLList)myList).iterator();
```

Queue ADT



- A queue is a sequence in which:
  - items are added at the rear and removed from the front
    - first in, first out (FIFO) (vs. a stack, which is last in, first out)
  - you can only access the item that is currently at the front
- Operations:
  - insert: add an item at the rear of the queue
  - remove: remove the item at the front of the queue
  - peek: get the item at the front of the queue, but don't remove it
  - isEmpty: test if the queue is empty
  - isFull: test if the queue is full
- Example: a queue of integers

start: 12 8

insert 5: 12 8 5

remove: 8 5

Our Generic Queue Interface

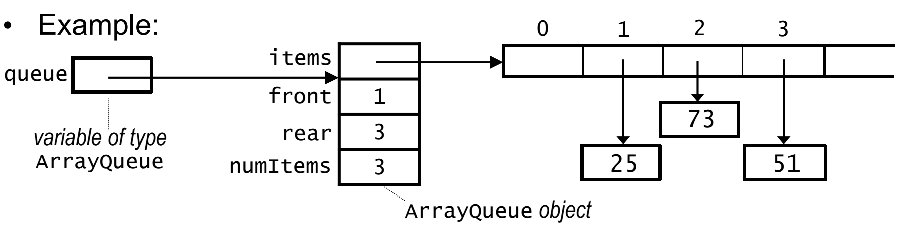
```
public interface Queue<T> {
    boolean insert(T item);
    T remove();
    T peek();
    boolean isEmpty();
    boolean isFull();
}
```

- `insert()` returns `false` if the queue is full, and `true` otherwise.
- `remove()` and `peek()` take no arguments, because we know that we always access the item at the front of the queue.
  - return `null` if the queue is empty.
- Here again, we will use encapsulation to ensure that the data structure is manipulated only in valid ways.

Implementing a Queue Using an Array

```
public class ArrayQueue<T> implements Queue<T> {
    private T[] items;
    private int front;
    private int rear;
    private int numItems;
    ...
}
```

- Example:



- We maintain two indices:
  - `front`: the index of the item at the front of the queue
  - `rear`: the index of the item at the rear of the queue

Avoiding the Need to Shift Items

- Problem: what do we do when we reach the end of the array?

  example: a queue of integers:

front					rear		
54	4	21	17	89	65		

  the same queue after removing two items and inserting one:

		front				rear	
		21	17	89	65	43	

  to insert two or more additional items, would need to shift items left

- Solution: maintain a *circular queue*. When we reach the end of the array, we wrap around to the beginning.

  the same queue after inserting two additional items:

rear		front					
5		21	17	89	65	43	81

A Circular Queue

- To get the front and rear indices to wrap around, we use the modulus operator (%).
- `x % y` = the remainder produced when you divide `x` by `y`
  - examples:
    - `10 % 7 = 3`
    - `36 % 5 = 1`
- Whenever we increment `front` or `rear`, we do so modulo the length of the array.

```
front = (front + 1) % items.length;
rear = (rear + 1) % items.length;
```

- Example:

		front					rear
		21	17	89	65	43	81

`items.length = 8, rear = 7`

before inserting the next item: `rear = (7 + 1) % 8 = 0`

which wraps `rear` around to the start of the array

Testing if an `ArrayQueue` is Empty

- Initial configuration:

  `rear = -1`
  `front = 0`

rear front

- We increment `rear` on every insertion, and we increment `front` on every removal.

after one insertion:

rear front						
15						

after two insertions:

front	rear					
15	32					

after one removal:

	front rear					
	32					

after two removals:

	rear	front				

- The queue is empty when `rear` is one position "behind" `front`:

```
((rear + 1) % items.length) == front
```

Testing if an `ArrayQueue` is Full

- Problem: if we use all of the positions in the array, our test for an empty queue will also hold when the queue is full!

  example: what if we added one more item to this queue?

rear		front					
5		21	17	89	65	43	81

- This is why we maintain `numItems`!

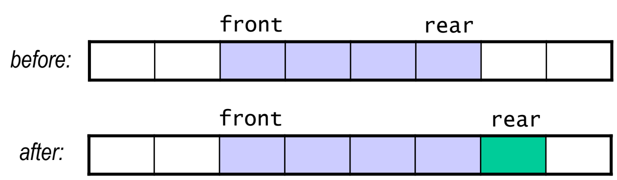
```
public boolean isEmpty() {
    return (numItems == 0);
}

public boolean isFull() {
    return (numItems == items.length);
}
```

Constructor

```
public ArrayQueue(int maxSize) {
    items = (T[])new Object[maxSize];
    front = 0;
    rear = -1;
    numItems = 0;
}
```

Inserting an Item in an `ArrayQueue`

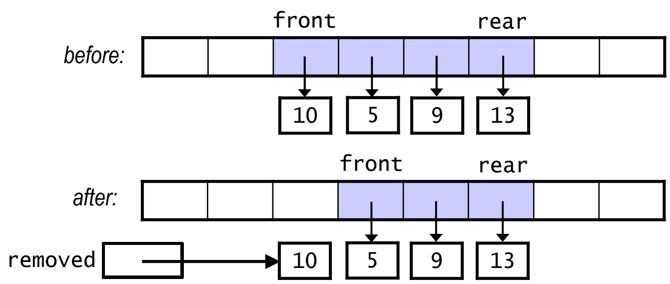
- We increment `rear` before adding the item:




```
public boolean insert(T item) {
    if (isFull())
        return false;
    rear = (rear + 1) % items.length;
    items[rear] = item;
    numItems++;
    return true;
}
```

`ArrayQueue remove()`

- remove: need to get `items[front]` *before* we increment `front`.

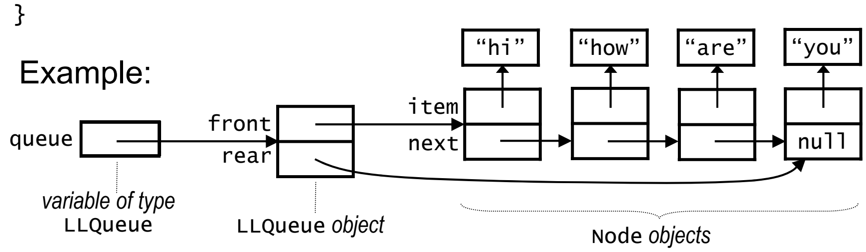



```
public T remove() {
    if (isEmpty())
        return null;
    T removed = items[front];
    items[front] = null;
    front = (front + 1) % items.length;
    numItems--;
    return removed;
}
```

Implementing a Queue Using a Linked List

```
public class LLQueue<T> implements Queue<T> {
    private Node front;    // front of the queue
    private Node rear;     // rear of the queue
    …
}
```

- Example:



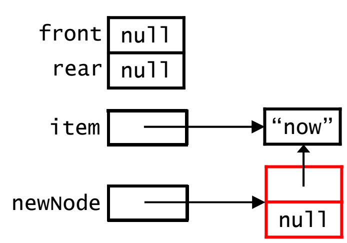
- Because a linked list can be easily modified on both ends, we don't need to take special measures to avoid shifting items, as we did in our array-based implementation.

Other Details of Our LLQueue Class

```
public class LLQueue<T> implements Queue<T> {
    private class Node {
        private T item;
        private Node next;
        ...
    }

    private Node front;
    private Node rear;

    public LLQueue() {
        front = rear = null;
    }
    public boolean isEmpty() {
        return (front == null);
    }
    public boolean isFull() {
        return false;
    }
    …
}
```

- Much simpler than the array-based queue!

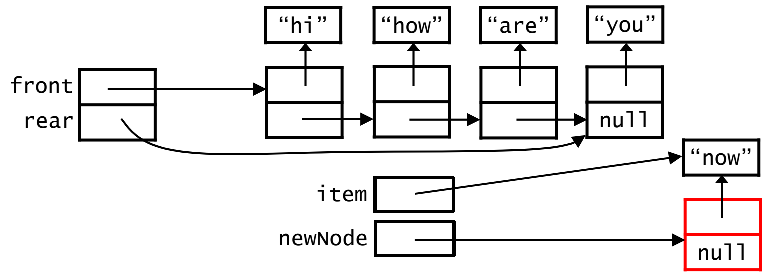
Inserting an Item in an Empty LLQueue



The `next` field in the `newNode` will be `null` in either case. Why?

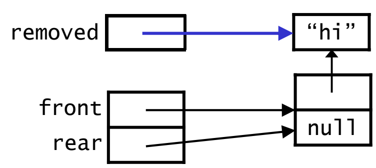
```
public boolean insert(T item) {
    Node newNode = new Node(item, null);
    if (isEmpty())

    else {


    }
    return true;
}
```

Inserting an Item in a Non-Empty LLQueue



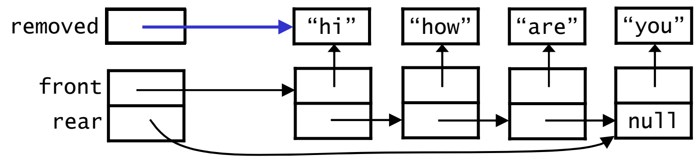

```
public boolean insert(T item) {
    Node newNode = new Node(item, null);
    if (isEmpty())

    else {


    }
    return true;
}
```

Removing from an LLQueue with One Item




```
public T remove() {
    if (isEmpty())
        return null;
    T removed = ______________;
    if (front == rear)       // removing the only item

    else

    return removed;
}
```

Removing from an LLQueue with Two or More Items




```
public T remove() {
    if (isEmpty())
        return null;

    T removed = ______________;
    if (front == rear)        // removing the only item

    else


    return removed;
}
```

Efficiency of the Queue Implementations

	ArrayQueue	LLQueue
insert()	*O*(1)	*O*(1)
remove()	*O*(1)	*O*(1)
peek()	*O*(1)	*O*(1)
space efficiency	*O*(m) where m is the *anticipated* maximum number of items	*O*(n) where n is the number of items currently in the queue

Applications of Queues

- first-in first-out (FIFO) inventory control
- OS scheduling: processes, print jobs, packets, etc.
- simulations of banks, supermarkets, airports, etc.
- breadth-first traversal of a graph or level-order traversal of a binary tree (more on these later)

Lists, Stacks, and Queues in Java's Class Library

- Lists:
  - interface: `java.util.List<T>`
    - slightly different methods, some extra ones
  - array-based implementations: `java.util.ArrayList<T>`
    `java.util.Vector<T>`
    - the array is expanded as needed
    - `Vector` has extra non-`List` methods
  - linked-list implementation: `java.util.LinkedList<T>`
    - `addLast()` provides $O(1)$ insertion at the end of the list
- Stacks: `java.util.Stack<T>`
    - extends `Vector` with methods that treat a vector like a stack
    - problem: other `Vector` methods can access items below the top
- Queues:
  - interface: `java.util.Queue<T>`
  - implementation: `java.util.LinkedList<T>`.

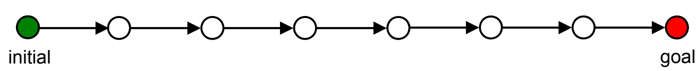
State-Space Search

Computer Science E-22
Harvard Extension School

David G. Sullivan, Ph.D.

Solving Problems by Searching

- A wide range of problems can be formulated as *searches*.
  - more precisely, as the process of searching for a *sequence of actions* that take you from an *initial state* to a *goal state*



- Examples:
  - n-queens
    - initial state: an empty n x n chessboard
    - actions (also called *operators*): place or remove a queen
    - goal state: n queens placed, with no two queens on the same row, column, or diagonal
  - map labeling, robot navigation, route finding, *many others*
- State space = all states reachable from the initial state by taking some sequence of actions.

The Eight Puzzle

- A 3 x 3 grid with 8 sliding tiles and one “blank”
- Goal state:

	1	2
3	4	5
6	7	8

- Initial state: some other configuration of the tiles
    - example:

3	1	2
4		5
6	7	8

- Slide tiles to reach the goal:

3	1	2
4		5
6	7	8

→

3	1	2
	4	5
6	7	8

→

	1	2
3	4	5
6	7	8

Formulating a Search Problem

- Need to specify:
    1. the *initial state*
    2. the *operators:* actions that take you from one state to another
    3. a *goal test:* determines if a state is a goal state
        - if only one goal state, see if the current state matches it
        - the test may also be more complex:
            - n-queens: do we have n queens on the board without any two queens on the same row, column, or diagonal?
    4. the *costs* associated with applying a given operator
        - allow us to differentiate between solutions
        - example: allow us to prefer 8-puzzle solutions that involve fewer steps
        - can be 0 if all solutions are equally preferable

Eight-Puzzle Formulation

- *initial state:* some configuration of the tiles
- *operators*: it's easier if we focus on the blank
  - get only four operators
    - move the blank up
    - move the blank down
    - move the blank left
    - move the blank right

3	1	2
4		5
6	7	8

move the blank up

3		2
4	1	5
6	7	8

- *goal test*: simple equality test, because there's only one goal

	1	2
3	4	5
6	7	8

- *costs:*
  - cost of each action = 1
  - cost of a sequence of actions = the number of actions

Performing State-Space Search

- Basic idea:

  If the initial state is a goal state, return it.

  If not, apply the operators to generate all states that are one step from the initial state (its *successors*).

3	1	2
4		5
6	7	8

initial state

3		2
4	1	5
6	7	8

3	1	2
4	7	5
6		8

3	1	2
	4	5
6	7	8

3	1	2
4	5	
6	7	8

its successors

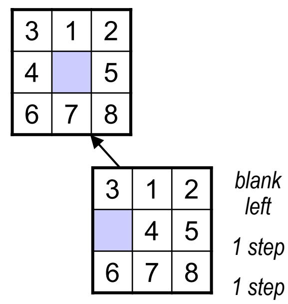
  Consider the successors (and their successors) until you find a goal state.

- Different search strategies consider the states in different orders.
  - they may use different data structures to store the states that have yet to be considered

Search Nodes

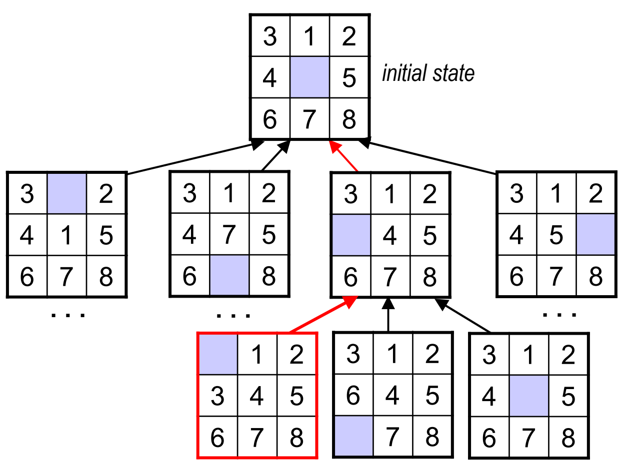
- When we generate a state, we create an object called a *search node* that contains the following:
  - a representation of the state
  - a reference to the node containing the *predecessor*
  - the operator (i.e., the action) that led from the predecessor to this state
  - the number of steps from the initial state to this state
  - the cost of getting from the initial state to this state
  - an estimate of the cost remaining to reach the goal

```
public class SearchNode {
    private Object state;
    private SearchNode predecessor;
    private String operator;
    private int numSteps;
    private double costFromStart;
    private double costToGoal;
    …
}
```



State-Space Search Tree

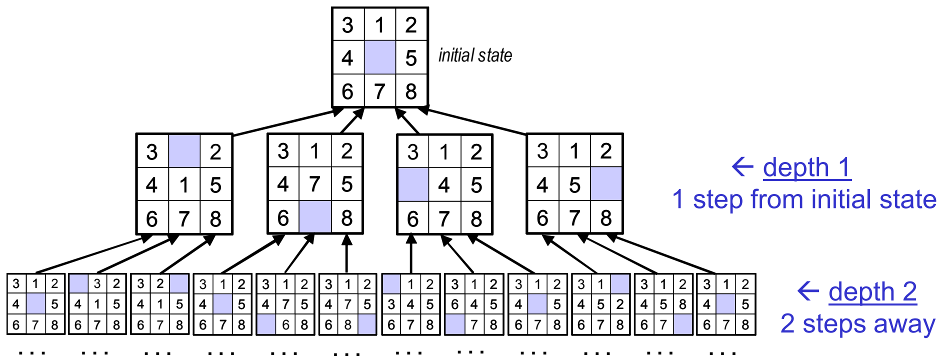
- The predecessor references connect the search nodes, creating a data structure known as a *tree*.



- When we reach a goal, we trace up the tree to get the solution – i.e., the sequence of actions from the initial state to the goal.

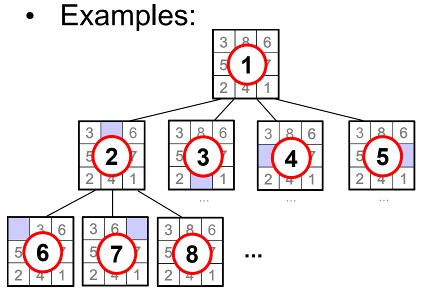
State-Space Search Tree (cont.)

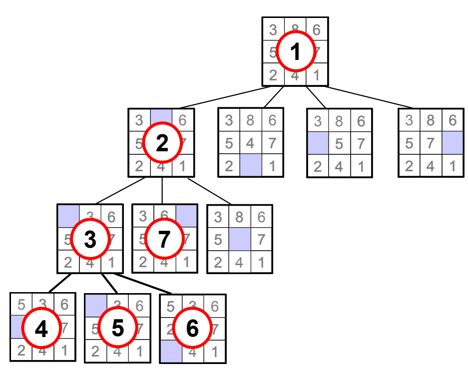
- The top node is called the *root*. It holds the initial state.
- The predecessor references are the *edges* of the tree.
- *depth* of a node *N* = # of edges on the path from *N* to the root
- All nodes at a depth i contain states that are i steps from the initial state:



State-Space Search Tree (cont.)

- Different search strategies correspond to different ways of considering the nodes in the search tree.
- Examples:

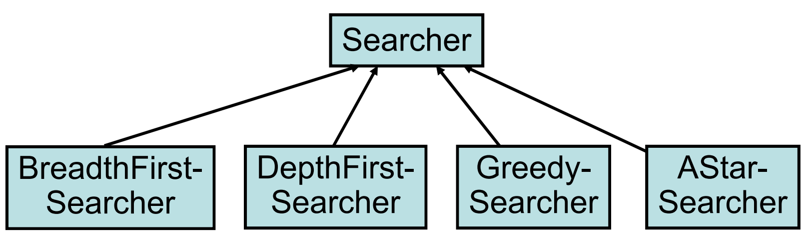




Representing a Search Strategy

- We'll use a *searcher* object.
- The searcher maintains a data structure containing the search nodes that we have yet to consider.
- Different search strategies have different searcher objects, which consider the search nodes in different orders.
- A searcher object may also have a *depth limit*, indicating that it will not consider search nodes beyond some depth.
- Every searcher must be able to do the following:
  - add a single node (or a list of nodes) to the collection of yet-to-be-considered nodes
  - indicate whether there are more nodes to be considered
  - return the next node to be considered
  - determine if a given node is at or beyond its depth limit

A Hierarchy of Searcher Classes



- `Searcher` is an *abstract* superclass.
  - defines instance variables and methods used by all search algorithms
  - includes one or more *abstract methods* – i.e., the method header is specified, but not the method definition
    - these methods are defined in the subclasses
  - it *cannot* be instantiated
- Implement each search algorithm as a subclass of `Searcher`.

An Abstract Class for Searchers

```
public abstract class Searcher {
    private int depthLimit;

    public abstract void addNode(SearchNode node);
    public abstract void addNodes(List nodes);
    public abstract boolean hasMoreNodes();
    public abstract SearchNode nextNode();
    …
    public void setDepthLimit(int limit) {
        depthLimit = limit;
    }
    public boolean depthLimitReached(SearchNode node) {
        …
    }
    …
}
```

- Classes for specific search strategies will extend this class and implement the abstract methods.
- We use an abstract class instead of an interface, because an abstract class allows us to include instance variables and method definitions that are inherited by classes that extend it.

Using Polymorphism

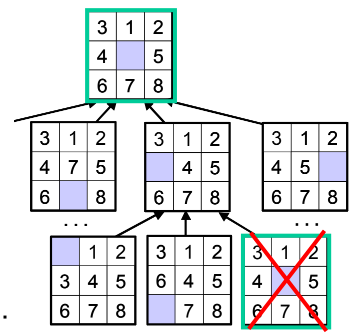
```
SearchNode findSolution(Searcher searcher, …) {
    numNodesVisited = 0;
    maxDepthReached = 0;

    searcher.addNode(makeFirstNode());
    ...
```

- The method used to find a solution takes a parameter of type `Searcher`.
- Because of polymorphism, we can pass in an object of *any* subclass of `Searcher`.
- Method calls made using the variable `searcher` will invoke the version of the method that is defined in the subclass to which the object belongs.
  - what is this called?

Pseudocode for Finding a Solution

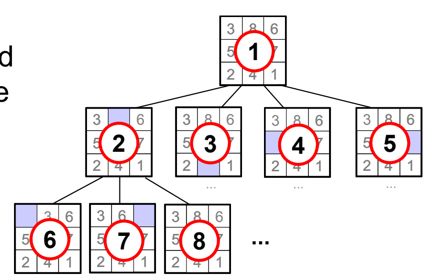
```
searcher.addNode(initial node);

while (searcher.hasMoreNodes()) {
    N = searcher.nextNode();
    if (N is the goal)
        return N;
    if (!searcher.depthLimitReached(N))
        searcher.addNodes(list of N's successors);
}
```

- Note that we don't generate a node's successors if the node is at or beyond the searcher's depth limit.
- Also, when generating successors, we usually don't include states that we've already seen in the current path from the initial state (ex. at right).



Breadth-First Search (BFS)

- When choosing a node from the collection of yet-to-be-considered nodes, always choose one of the shallowest ones.
  - consider all nodes at depth 0
  - consider all nodes at depth 1
  - ...

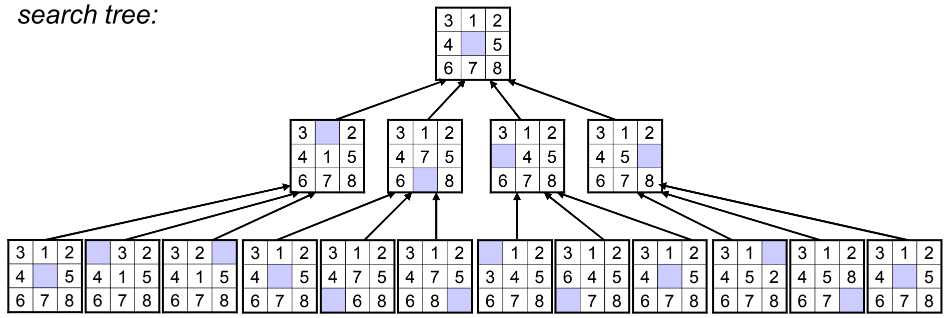


- The searcher for this strategy uses a *queue*.

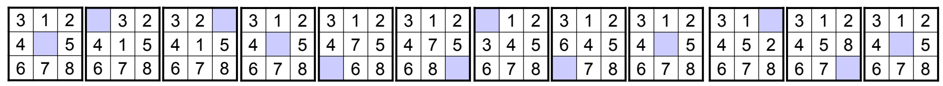
```
public class BreadthFirstSearcher extends Searcher {
    private Queue<SearchNode> nodeQueue;

    public void addNode(SearchNode node) {
        nodeQueue.insert(node);
    }
    public SearchNode nextNode() {
        return nodeQueue.remove();
    }
    …
}
```

Tracing Breadth-First Search

search tree:



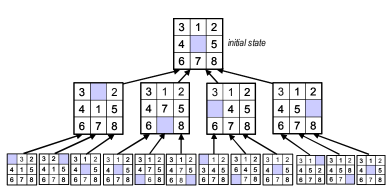
queue:



After considering all nodes at a depth of 1, BFS would move next to nodes with a depth of 2. *All previously considered nodes remain in the tree, because they have yet-to-be-considered successors.*

Features of Breadth-First Search

- It is *complete:* if there is a solution, BFS will find it.
- For problems like the eight puzzle in which each operator has the same cost, BFS is *optimal*: it will find a minimal-cost solution.
  - it may *not* be optimal if different operators have different costs
- Time and space complexity:
  - assume each node has b successors in the worst case
  - finding a solution that is at a depth d in the search tree has a time *and* space complexity = ?



← 1 node at depth 0

← $O(b)$ nodes at depth 1

← $O(b^2)$ nodes at depth 2

  - nodes considered (and *stored*) = $1 + b + b^2 + \quad + b^d$ = ?

Features of Breadth-First Search (cont.)

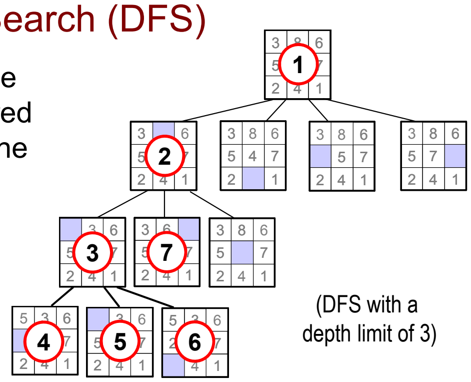
- Exponential space complexity turns out to be a bigger problem than exponential time complexity.
- Time and memory usage when b = 10:

solution depth	nodes considered	time	memory
0	1	1 millisecond	100 bytes
4	11,111	11 seconds	1 megabyte
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	***1 terabyte***
12	10^{12}	35 years	111 terabytes

- Try running our 8-puzzle solver on the initial state shown at right!

	8	7
6	5	4
3	2	1

Depth-First Search (DFS)



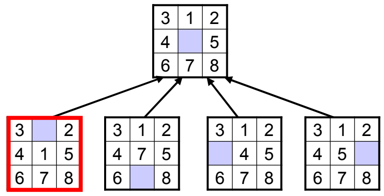
(DFS with a depth limit of 3)

- When choosing a node from the collection of yet-to-be-considered nodes, always choose one of the deepest ones.
  - keep going down a given path in the tree until you're stuck, and then backtrack
- What data structure should this searcher use?

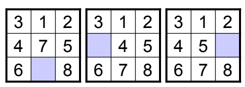
```
public class DepthFirstSearcher extends Searcher {

    public void addNode(SearchNode node) {

    }
    public SearchNode nextNode() {

    }
    …
}
```

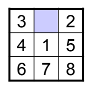
Tracing Depth-First Search (depth limit = 2)

search tree:



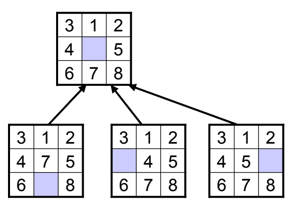
stack:



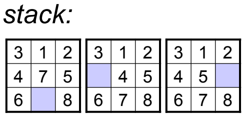
Once all of the successors of  have been considered, there are no remaining references to it. Thus, the memory for this node will also be reclaimed.

Tracing Depth-First Search (depth limit = 2)

search tree:



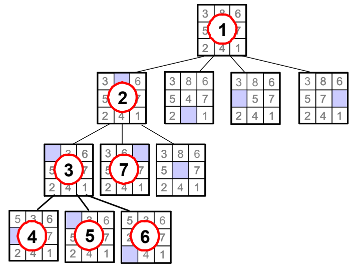
stack:



DFS would next consider paths that pass through its second successor. *At any point in time, only the nodes from a single path (along with their "siblings") are stored in memory.*

Features of Depth-First Search

- Much better space complexity:
  - let m be the maximum depth of a node in the search tree
  - DFS only stores a single path in the tree at a given time – along with the "siblings" of each node on the path
  - space complexity = $O(b*m)$

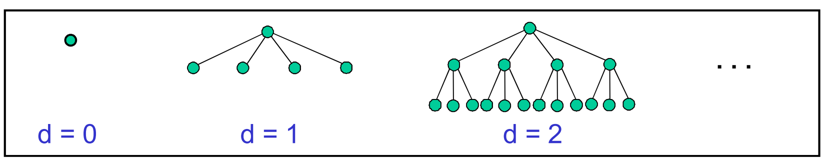


- Time complexity: if there are many solutions, DFS can often find one quickly. However, worst-case time complexity = $O(b^m)$.
- Problem – it can get stuck going down the wrong path.
  ➔ thus, it is neither complete nor optimal.
- Adding a depth limit helps to deal with long or even infinite paths, but how do you know what depth limit to use?

Iterative Deepening Search (IDS)

- Eliminates the need to choose a depth limit
- Basic idea:

```
d = 0;
while (true) {
    perform DFS with a depth limit of d;
    d++;
}
```



- Combines the best of DFS and BFS:
  - at any point in time, we're performing DFS, so the space complexity is linear
  - we end up considering all nodes at each depth limit, so IDS is complete like BFS (and optimal when BFS is)

Can't We Do Better?

- Yes!
- BFS, DFS, and IDS are all examples of *uninformed* search algorithms – they always consider the states in a certain order, without regard to how close a given state is to the goal.
- There exist other *informed* search algorithms that consider (estimates of) how close a state is from the goal when deciding which state to consider next.
- We'll come back to this topic once we've considered more data structures!
- For more on using state-space search to solve problems:

  Artificial Intelligence: A Modern Approach.
  Stuart Russell and Peter Norvig (Prentice-Hall).

- The code for the Eight-Puzzle solver is in code for this unit.
- To run the Eight-Puzzle solver:

```
javac EightPuzzle.java
java EightPuzzle
```

When it asks for the initial board, enter a string specifying the positions of the tiles, with 0 representing the blank.

example: for

	8	7
6	5	4
3	2	1

you would enter 087654321

- To get a valid initial state, take a known configuration of the tiles and swap *two pairs* of tiles. Example:

(you can also "move the blank" as you ordinarily would)

3	1	2
4		5
6	7	8

⇨

3	**7**	2
4		5
6	**1**	8

⇨

3	7	**4**
2		5
6	1	8

Binary Trees and Huffman Encoding
Binary Search Trees

Computer Science E-22
Harvard Extension School

David G. Sullivan, Ph.D.

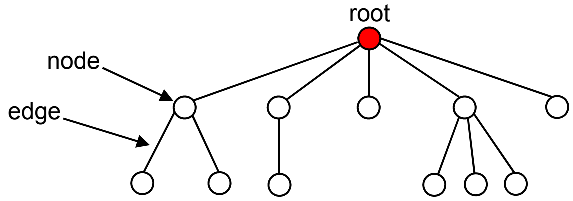
Motivation: Maintaining a Sorted Collection of Data

- A *data dictionary* is a sorted collection of data with the following key operations:
  - *search* for an item (and possibly delete it)
  - *insert* a new item
- If we use a list to implement a data dictionary, efficiency = $O(n)$.

data structure	***searching for an item***	***inserting an item***
a list implemented using an array		
a list implemented using a linked list		

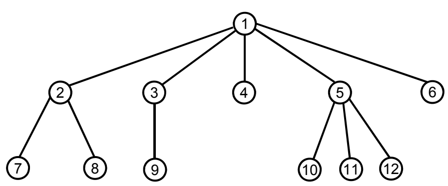
- In the next few lectures, we'll look at data structures (trees and hash tables) that can be used for a more efficient data dictionary.
- We'll also look at other applications of trees.

What Is a Tree?



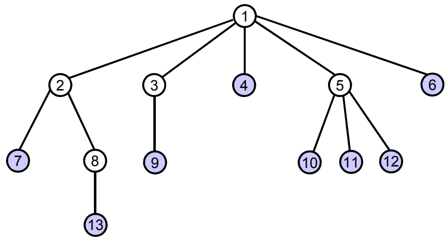
- A tree consists of:
  - a set of *nodes*
  - a set of *edges*, each of which connects a pair of nodes
- Each node may have one or more *data items.*
  - each data item consists of one or more fields
  - *key field* = the field used when searching for a data item
  - multiple data items with the same key are referred to as *duplicates*
- The node at the “top” of the tree is called the *root* of the tree.

Relationships Between Nodes



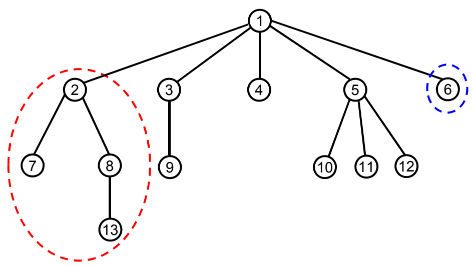
- If a node N is connected to other nodes that are directly below it in the tree, N is referred to as their *parent* and they are referred to as its *children*.
  - example: node 5 is the parent of nodes 10, 11, and 12
- Each node is the child of *at most one* parent.
- Other family-related terms are also used:
  - nodes with the same parent are *siblings*
  - a node’s *ancestors* are its parent, its parent’s parent, etc.
    - example: node 9’s ancestors are 3 and 1
  - a node’s *descendants* are its children, their children, etc.
    - example: node 1’s descendants are *all* of the other nodes

Types of Nodes



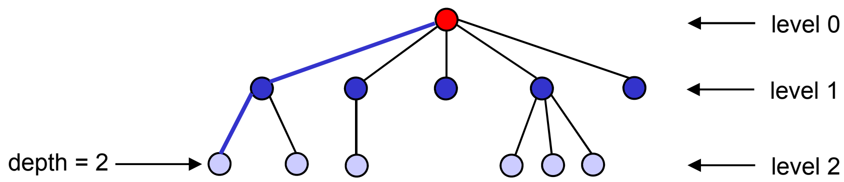
- A *leaf node* is a node without children.
- An *interior node* is a node with one or more children.

A Tree is a Recursive Data Structure



- Each node in the tree is the root of a smaller tree!
  - refer to such trees as *subtrees* to distinguish them from the tree as a whole
  - example: node 2 is the root of the subtree circled above
  - example: node 6 is the root of a subtree with only one node
- We'll see that tree algorithms often lend themselves to recursive implementations.

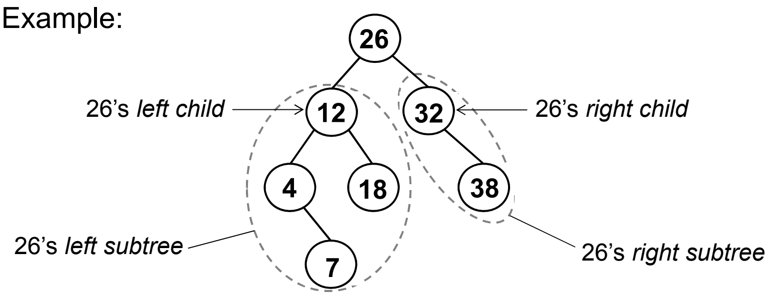
Path, Depth, Level, and Height



- There is exactly one *path* (one sequence of edges) connecting each node to the root.
- *depth* of a node = # of edges on the path from it to the root
- Nodes with the same depth form a *level* of the tree.
- The *height* of a tree is the maximum depth of its nodes.
    - example: the tree above has a height of 2

Binary Trees

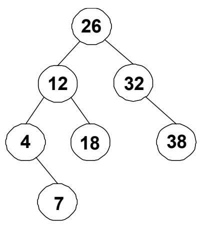
- In a *binary tree*, nodes have *at most two* children.
- Recursive definition: a binary tree is either:
    1) empty, or
    2) a node (the root of the tree) that has
        - one or more data fields
        - a *left child*, which is itself the root of a binary tree
        - a *right child*, which is itself the root of a binary tree
- Example:

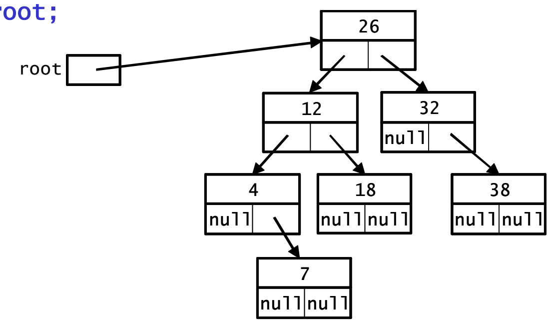


- How are the edges of the tree represented?

Representing a Binary Tree Using Linked Nodes

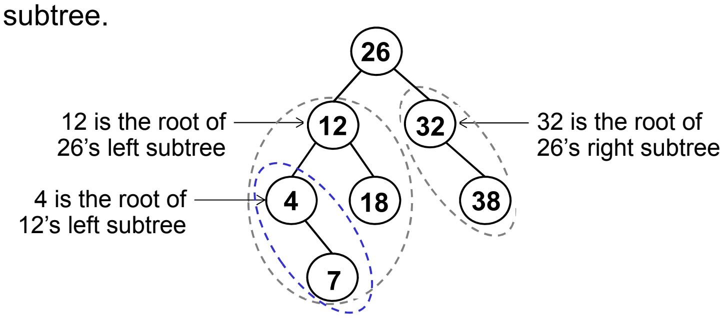
```
public class LinkedTree {
    private class Node {
        private int key;
        private LLList data;   // list of data for that key
        private Node left;     // reference to left child
        private Node right;    // reference to right child
        …
    }

    private Node root;
    …
}
```





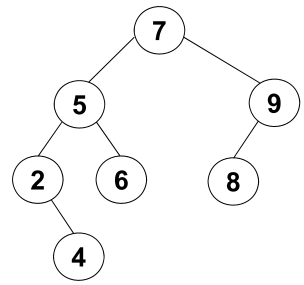
Traversing a Binary Tree

- Traversing a tree involves *visiting* all of the nodes in the tree.
  - visiting a node = processing its data in some way
    - example: print the key
- We will look at four types of traversals. Each of them visits the nodes in a different order.
- To understand traversals, it helps to remember the recursive definition of a binary tree, in which every node is the root of a subtree.



Preorder Traversal

- preorder traversal of the tree whose root is N:
  1) visit the root, N
  2) recursively perform a preorder traversal of N's left subtree
  3) recursively perform a preorder traversal of N's right subtree



- Preorder traversal of the tree above:

  7 5 2 4 6 9 8
- Which state-space search strategy visits nodes in this order?

Implementing Preorder Traversal

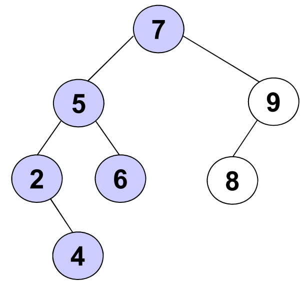
```
public class LinkedTree {
    …
    private Node root;
    public void preorderPrint() {
        if (root != null)
            preorderPrintTree(root);
    }
    private static void preorderPrintTree(Node root) {
        System.out.print(root.key + “  ”);
        if (root.left != null)
            preorderPrintTree(root.left);
        if (root.right != null)
            preorderPrintTree(root.right);
    }
}
```

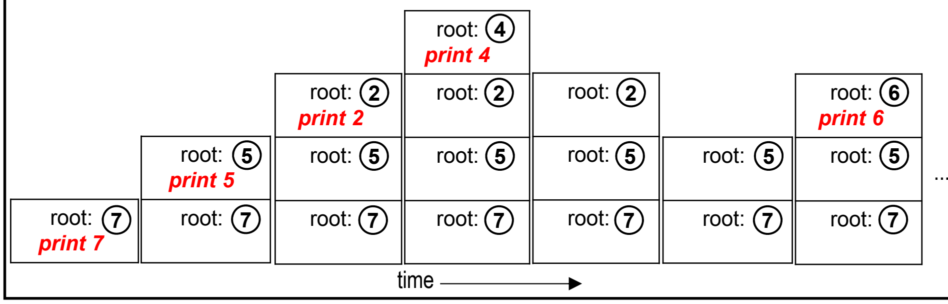
Not always the same as the root of the entire tree.

- `preorderPrintTree()` is a static, recursive method that takes as a parameter the root of the tree/subtree that you want to print.
- `preorderPrint()` is a non-static method that makes the initial call. It passes in the root of the entire tree as the parameter.

Tracing Preorder Traversal

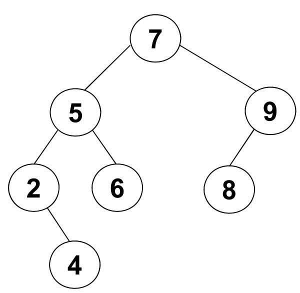
```
void preorderPrintTree(Node root) {
   System.out.print(root.key + "  ");
   if (root.left != null)
       preorderPrintTree(root.left);
   if (root.right != null)
       preorderPrintTree(root.right);
}
```





Postorder Traversal

- postorder traversal of the tree whose root is N:
  1) recursively perform a postorder traversal of N's left subtree
  2) recursively perform a postorder traversal of N's right subtree
  3) visit the root, N



- Postorder traversal of the tree above:

  4 2 6 5 8 9 7

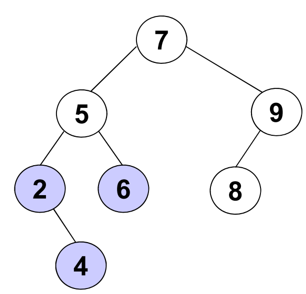
Implementing Postorder Traversal

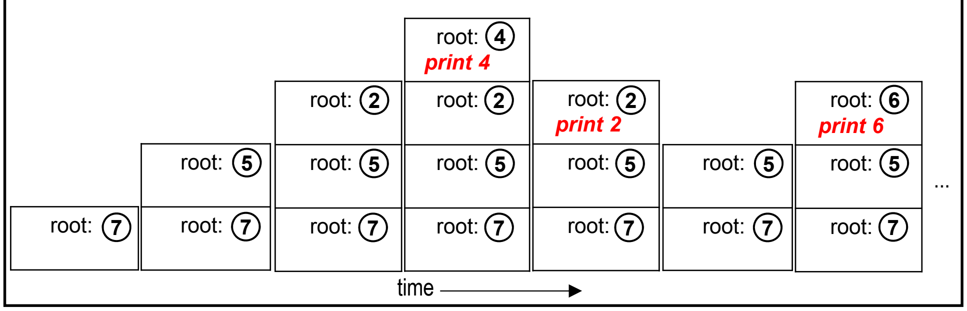
```
public class LinkedTree {
    …
    private Node root;

    public void postorderPrint() {
        if (root != null)
            postorderPrintTree(root);
    }

    private static void postorderPrintTree(Node root) {
        if (root.left != null)
            postorderPrintTree(root.left);
        if (root.right != null)
            postorderPrintTree(root.right);
        System.out.print(root.key + "  ");
    }
}
```

- Note that the root is printed *after* the two recursive calls.

Tracing Postorder Traversal

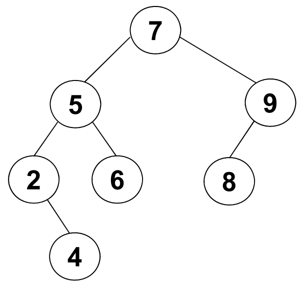
```
void postorderPrintTree(Node root) {
   if (root.left != null)
       postorderPrintTree(root.left);
   if (root.right != null)
       postorderPrintTree(root.right);
   System.out.print(root.key + "  ");
}
```





Inorder Traversal

- inorder traversal of the tree whose root is N:
  1) recursively perform an inorder traversal of N's left subtree
  2) visit the root, N
  3) recursively perform an inorder traversal of N's right subtree



- Inorder traversal of the tree above:

  2 4 5 6 7 8 9

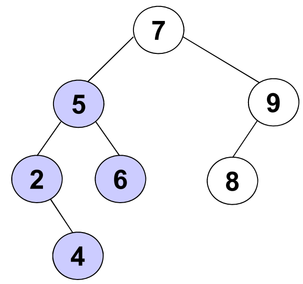
Implementing Inorder Traversal

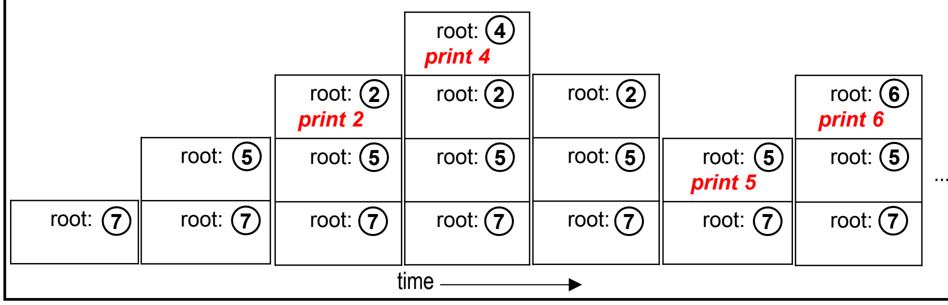
```
public class LinkedTree {
    …
    private Node root;

    public void inorderPrint() {
        if (root != null)
            inorderPrintTree(root);
    }

    private static void inorderPrintTree(Node root) {
        if (root.left != null)
            inorderPrintTree(root.left);
        System.out.print(root.key + "  ");
        if (root.right != null)
            inorderPrintTree(root.right);
    }
}
```

- Note that the root is printed *between* the two recursive calls.

Tracing Inorder Traversal

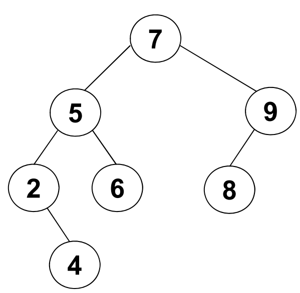
```
void inorderPrintTree(Node root) {
   if (root.left != null)
       inorderPrintTree(root.left);
   System.out.print(root.key + "  ");
   if (root.right != null)
       inorderPrintTree(root.right);
}
```





Level-Order Traversal

- Visit the nodes one level at a time, from top to bottom and left to right.



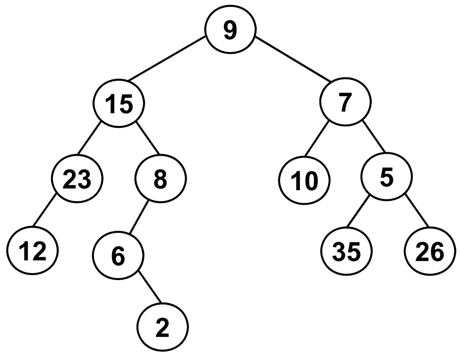
- Level-order traversal of the tree above: **7 5 9 2 6 8 4**
- Which state-space search strategy visits nodes in this order?

- How could we implement this type of traversal?

Tree-Traversal Summary

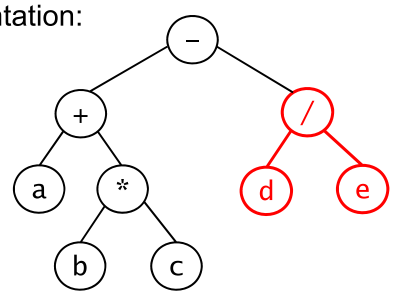
preorder: root, left subtree, right subtree
postorder: left subtree, right subtree, root
inorder: left subtree, root, right subtree
level-order: top to bottom, left to right

- Perform each type of traversal on the tree below:



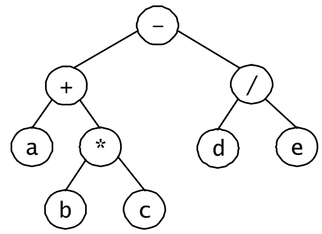
Using a Binary Tree for an Algebraic Expression

- We'll restrict ourselves to fully parenthesized expressions and to the following binary operators: +, –, *, /
- Example expression: `((a + (b * c)) – (d / e))`
- Tree representation:



- Leaf nodes are variables or constants; interior nodes are operators.
- Because the operators are binary, either a node has two children or it has none.

Traversing an Algebraic-Expression Tree



- Inorder gives conventional algebraic notation.
  - print '(' before the recursive call on the left subtree
  - print ')' after the recursive call on the right subtree
  - for tree at right: `((a + (b * c)) – (d / e))`
- Preorder gives functional notation.
  - print '('s and ')'s as for inorder, and commas after the recursive call on the left subtree
  - for tree above: `subtr(add(a, mult(b, c)), divide(d, e))`
- Postorder gives the order in which the computation must be carried out on a stack/RPN calculator.
  - for tree above: `push a, push b, push c, multiply, add,…`
- see `ExprTree.java`

Fixed-Length Character Encodings

- A character encoding maps each character to a number.
- Computers usually use fixed-length character encodings.
  - ASCII (American Standard Code for Information Interchange) uses 8 bits per character.

char	dec	binary
a	97	01100001
b	98	01100010
c	99	01100011

example: "`bat`" is stored in a text file as the following sequence of bits:

`01100010 01100001 01110100`

  - Unicode uses 16 bits per character to accommodate foreign-language characters. (ASCII codes are a subset.)
- Fixed-length encodings are simple, because
  - all character encodings have the same length
  - a given character always has the same encoding

Variable-Length Character Encodings

- Problem: fixed-length encodings waste space.
- Solution: use a variable-length encoding.
  - use encodings of different lengths for different characters
  - assign shorter encodings to frequently occurring characters
- Example:

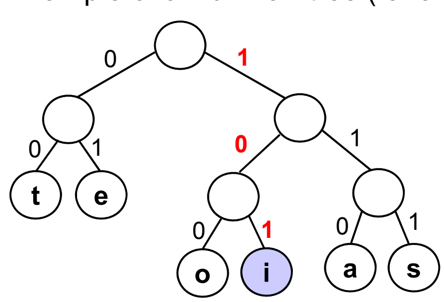
e	01
o	100
s	111
t	00

"test" would be encoded as
`00 01 111 00` → `000111100`

- Challenge: when decoding/decompressing an encoded document, how do we determine the boundaries between characters?
  - example: for the above encoding, how do we know whether the next character is 2 bits or 3 bits?
- One requirement: no character's encoding can be the prefix of another character's encoding (e.g., couldn't have 00 and 001).

Huffman Encoding

- Huffman encoding is a type of variable-length encoding that is based on the actual character frequencies in a given document.
- Huffman encoding uses a binary tree:
  - to determine the encoding of each character
  - to decode an encoded file – i.e., to decompress a compressed file, putting it back into ASCII
- Example of a Huffman tree (for a text with only six chars):



Leaf nodes are characters.

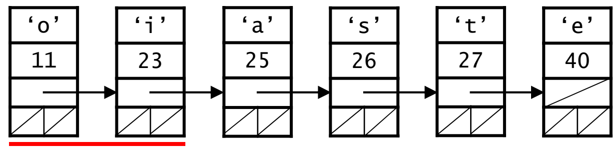
Left branches are labeled with a 0, and right branches are labeled with a 1.

If you follow a path from root to leaf, you get the encoding of the character in the leaf
  example: 101 = 'i'

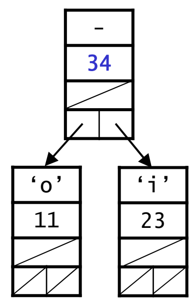
Building a Huffman Tree

1) Begin by reading through the text to determine the frequencies.

2) Create a list of nodes that contain (character, frequency) pairs for each character that appears in the text.

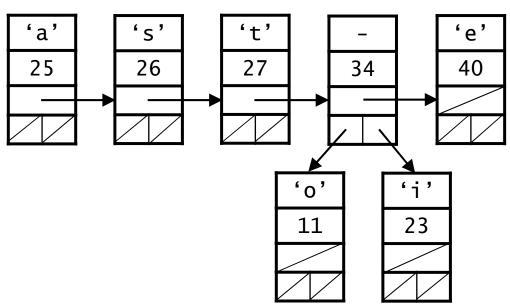


3) Remove and “merge” the nodes with the two lowest frequencies, forming a new node that is their parent.
- left child = lowest frequency node
- right child = the other node
- frequency of parent = sum of the frequencies of its children
  - in this case, 11 + 23 = 34



Building a Huffman Tree (cont.)

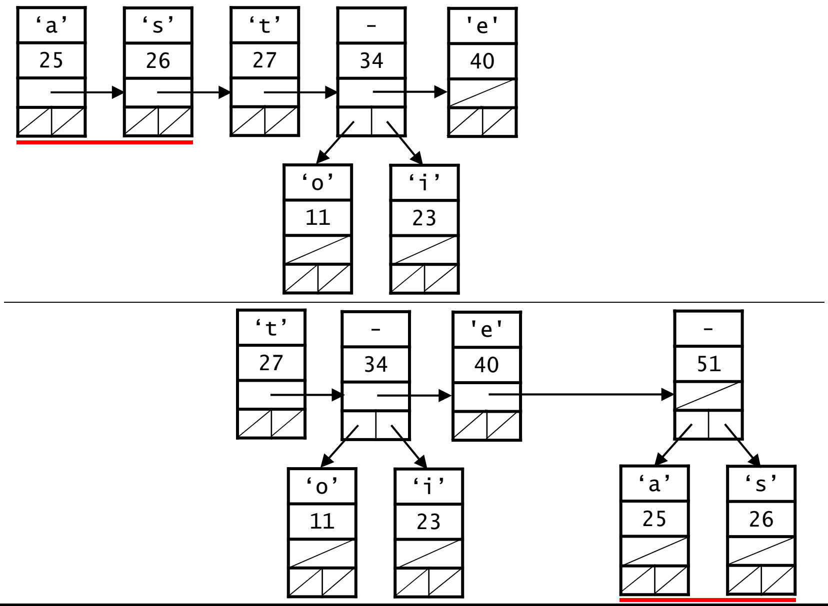
4) Add the parent to the list of nodes (maintaining sorted order):



5) Repeat steps 3 and 4 until there is only a single node in the list, which will be the root of the Huffman tree.

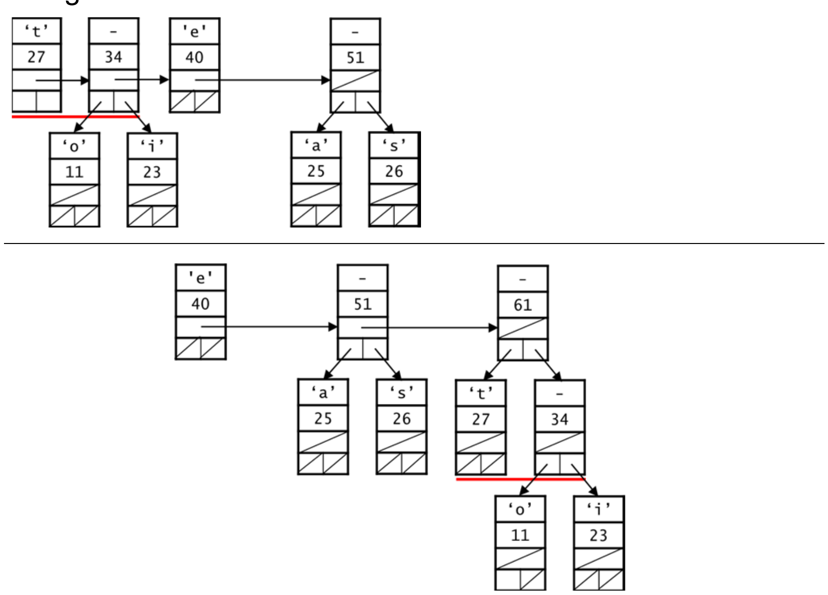
Completing the Huffman Tree Example I

- Merge the two remaining nodes with the lowest frequencies:



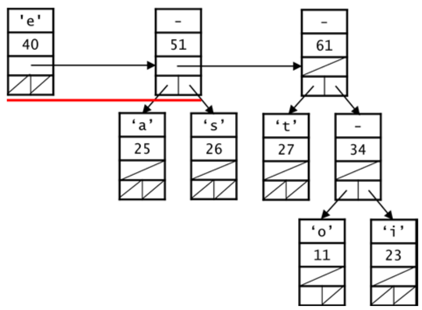
Completing the Huffman Tree Example II

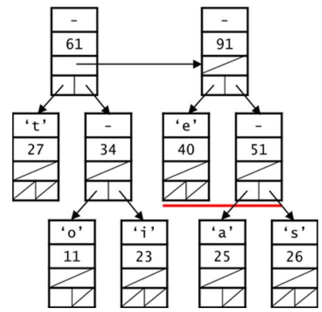
- Merge the next two nodes:



Completing the Huffman Tree Example II

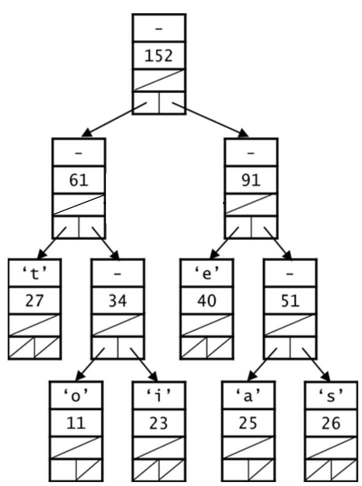
- Merge again:

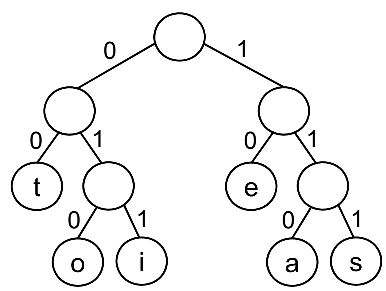




Completing the Huffman Tree Example IV

- The next merge creates the final tree:

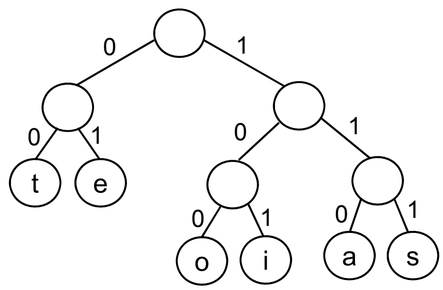




- Characters that appear more frequently end up higher in the tree, and thus their encodings are shorter.

The Shape of the Huffman Tree

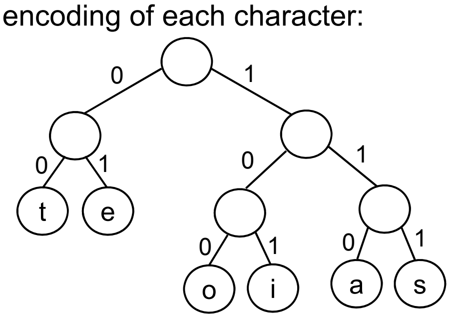
- The tree on the last slide is fairly symmetric.
- This won't always be the case!
  - depends on the frequencies of the characters in the document being compressed
- For example, changing the frequency of `'o'` from 11 to 21 would produce the tree shown below:



- This is the tree that we'll use in the remaining slides.

Using Huffman Encoding to Compress a File

1) Read through the input file and build its Huffman tree.
2) Write a file header for the output file.
   – include an array containing the frequencies so that the tree can be rebuilt when the file is decompressed.
3) Traverse the Huffman tree to create a table containing the encoding of each character:



a	?
e	?
i	101
o	100
s	111
t	00

4) Read through the input file a second time, and write the Huffman code for each character to the output file.

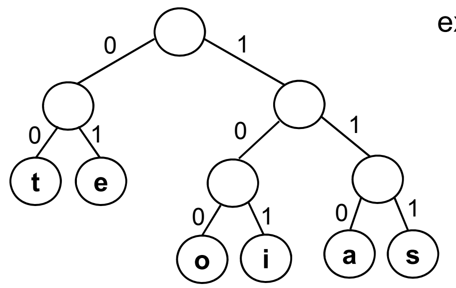
Using Huffman Decoding to Decompress a File

1) Read the frequency table from the header and rebuild the tree.

2) Read one bit at a time and traverse the tree, starting from the root:

    when you read a bit of 1, go to the right child
    when you read a bit of 0, go to the left child
    when you reach a leaf node, record the character,
        return to the root, and continue reading bits

The tree allows us to easily overcome the challenge of determining the character boundaries!

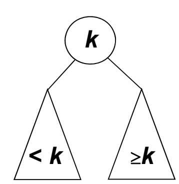


example: 101111110000111100

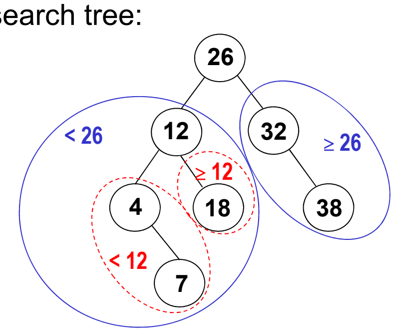
101 =	right,left,right	=	i
111 =	right,right,right	=	s
110 =	right,right,left	=	a
00 =	left,left	=	t
01 =	left,right	=	e
111 =	right,right,right	=	s
00 =	left,left	=	t

Binary *Search* Trees

- Search-tree property: for each node *k:*
    - all nodes in *k*'s left subtree are < *k*
    - all nodes in *k*'s right subtree are >= *k*



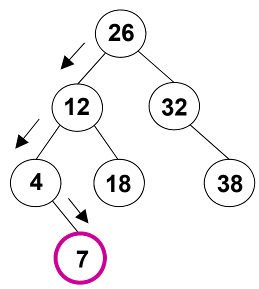
- Our earlier binary-tree example is a search tree:



Searching for an Item in a Binary Search Tree

- Algorithm for searching for an item with a key *k*:
  if *k* == the root node's key, you're done
  else if *k* < the root node's key, search the left subtree
  else search the right subtree

- Example: search for 7

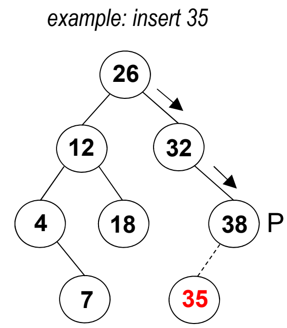


Implementing Binary-Tree Search

```
public class LinkedTree {    // Nodes have keys that are ints
    …
    private Node root;

    public LLList search(int key) {
        Node n = searchTree(root, key);
        if (n == null)
            return null;     // no such key
        else
            return n.data;   // return list of values for key
    }

    private static Node searchTree(Node root, int key) {
        // write together









    }
}
```

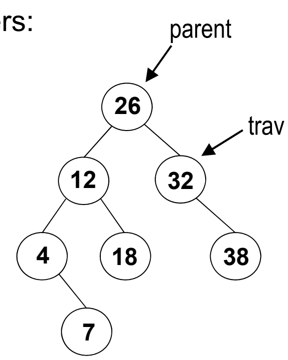
Inserting an Item in a Binary Search Tree

- We want to insert an item whose key is *k*.
- We traverse the tree as if we were searching for *k.*
- If we find a node with key *k*, we add the data item to the list of items for that node.
- If we don't find it, the last node we encounter will be the parent P of the new node.
  - if *k* < P's key, make the new node P's left child
  - else make the node P's right child
- *Special case:* if the tree is empty, make the new node the root of the tree.
- The resulting tree is still a search tree.



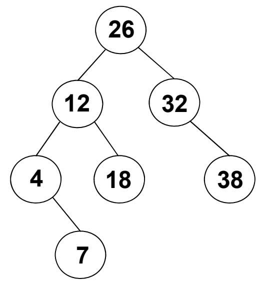
Implementing Binary-Tree Insertion

- We'll implement part of the `insert()` method together.
- We'll use iteration rather than recursion.
- Our method will use two references/pointers:
  - `trav`: performs the traversal down to the point of insertion
  - `parent`: stays one behind `trav`
    - like the `trail` reference that we sometimes use when traversing a linked list



Implementing Binary-Tree Insertion

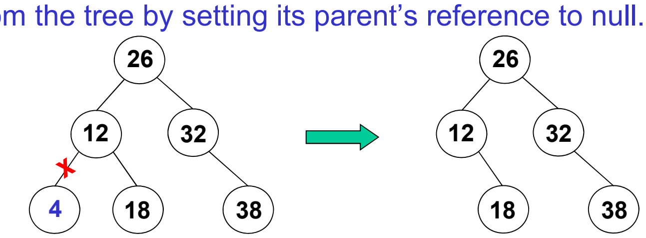
```java
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        }






    }
    Node newNode = new Node(key, data);
    if (root == null)    // the tree was empty
        root = newNode;
    else if (key < parent.key)
        parent.left = newNode;
    else
        parent.right = newNode;
}
```



Deleting Items from a Binary Search Tree

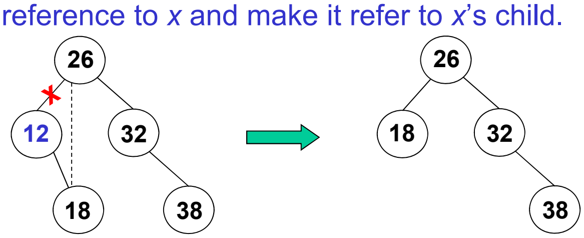
- Three cases for deleting a node *x*
- **Case 1:** *x* has no children.
  Remove *x* from the tree by setting its parent's reference to null.

ex: delete 4



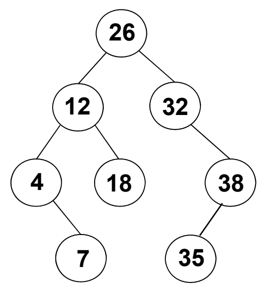
- **Case 2:** *x* has one child.
  Take the parent's reference to *x* and make it refer to *x*'s child.

ex: delete 12



Deleting Items from a Binary Search Tree (cont.)

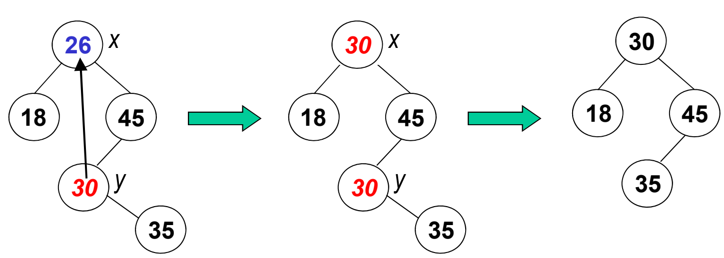
- **Case 3:** *x* has two children
  - we can't just delete *x*. why?
  - instead, we replace *x* with a node from elsewhere in the tree
  - to maintain the search-tree property, we must choose the replacement carefully
    - example: what nodes could replace 26 below?



Deleting Items from a Binary Search Tree (cont.)

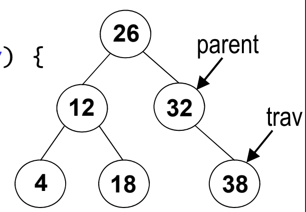
- **Case 3:** *x* has two children (continued):
  - replace *x* with the smallest node in *x*'s right subtree—call it *y*
  - – *y* will either be a leaf node or will have one right child. why?
- After copying *y*'s item into *x*, we delete *y* using case 1 or 2.

ex:
delete 26



Implementing Binary-Tree Deletion

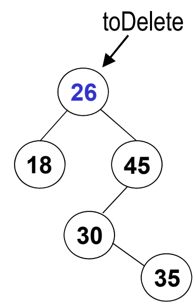
```
public LLList delete(int key) {
    // Find the node and its parent.
    Node parent = null;
    Node trav = root;
    while (trav != null && trav.key != key) {
        parent = trav;
        if (key < trav.key)
            trav = trav.left;
        else
            trav = trav.right;
    }

    // Delete the node (if any) and return the removed items.
    if (trav == null)    // no such key
        return null;
    else {
        LLList removedData = trav.data;
        deleteNode(trav, parent);
        return removedData;
    }
}
```



- This method uses a helper method to delete the node.

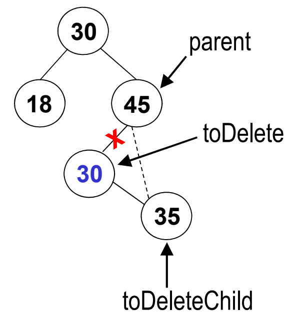
Implementing Case 3

```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        // Find a replacement – and
        // the replacement's parent.
        Node replaceParent = toDelete;

        // Get the smallest item
        // in the right subtree.
        Node replace = toDelete.right;
        // What should go here?



        // Replace toDelete's key and data
        // with those of the replacement item.
        toDelete.key = replace.key;
        toDelete.data = replace.data;

        // Recursively delete the replacement
        // item's old node. It has at most one
        // child, so we don't have to
        // worry about infinite recursion.
        deleteNode(replace, replaceParent);
    } else {
        ...
}
```



Implementing Cases 1 and 2

```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        ...
    } else {
        Node toDeleteChild;
        if (toDelete.left != null)
            toDeleteChild = toDelete.left;
        else
            toDeleteChild = toDelete.right;
        // Note: in case 1, toDeleteChild
        // will have a value of null.

        if (toDelete == root)
            root = toDeleteChild;
        else if (toDelete.key < parent.key)
            parent.left = toDeleteChild;
        else
            parent.right = toDeleteChild;
    }
}
```

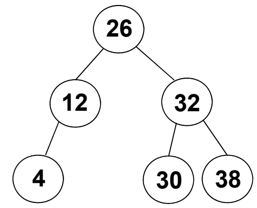


Efficiency of a Binary Search Tree

- The three key operations (search, insert, and delete) all have the same time complexity.
  - insert and delete both involve a search followed by a constant number of additional operations
- Time complexity of searching a binary search tree:
  - best case: $O(1)$
  - worst case: $O(h)$, where h is the height of the tree
  - average case: $O(h)$
- What is the height of a tree containing n items?
  - it depends! why?

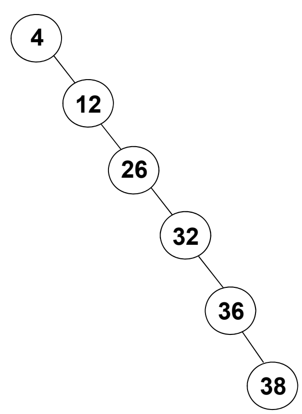
Balanced Trees

- A tree is *balanced* if, for each node, the node's subtrees have the same height or have heights that differ by 1.
- For a balanced tree with n nodes:
  - height = $O(\log_2 n)$.
  - gives a worst-case time complexity that is logarithmic ($O(\log_2 n)$)
    - the best worst-case time complexity for a binary tree



What If the Tree Isn't Balanced?

- Extreme case: the tree is equivalent to a linked list
  - height = n – 1
  - worst-case time complexity = $O(n)$
- We'll look next at search-tree variants that take special measures to ensure balance.



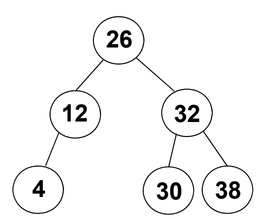
Balanced Search Trees

Computer Science E-22
Harvard Extension School

David G. Sullivan, Ph.D.

Review: Balanced Trees

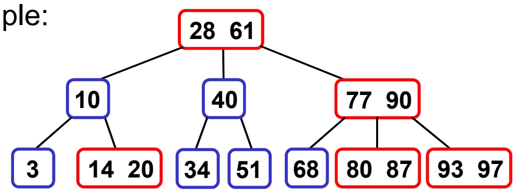
- A tree is *balanced* if, for each node, the node's subtrees have the same height or have heights that differ by 1.
- For a balanced tree with n nodes:
  - height = $O(\log_2 n)$.
  - gives a worst-case time complexity that is logarithmic ($O(\log_2 n)$)
    - the best worst-case time complexity for a binary search tree

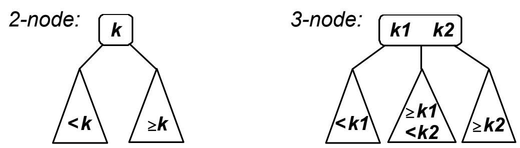


- With a binary search tree, there's no way to ensure that the tree remains balanced.
  - can degenerate to $O(n)$ time

2-3 Trees

- A 2-3 tree is a balanced tree in which:
  - *all* nodes have equal-height subtrees (perfect balance)
  - each node is either
    - a **2-node**, which contains one data item and 0 or 2 children
    - a **3-node**, which contains two data items and 0 or 3 children
  - the keys form a search tree
- Example:

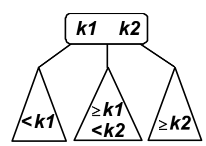




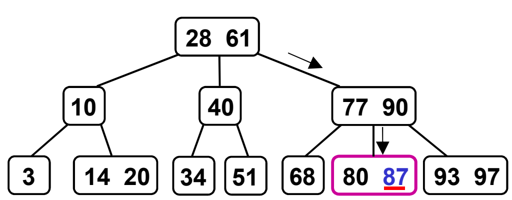
Search in 2-3 Trees

- Algorithm for searching for an item with a key *k*:

  if *k* == one of the root node's keys, you're done
  else if *k* < the root node's first key
    search the left subtree
  else if the root is a 3-node and k < its second key
    search the middle subtree
  else
    search the right subtree



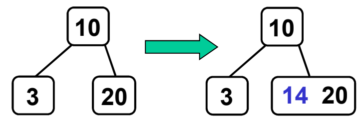
- Example: search for 87



Insertion in 2-3 Trees

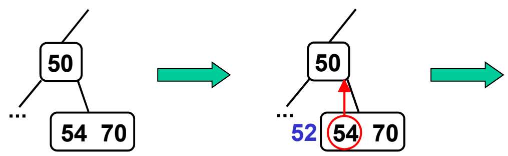
- Algorithm for inserting an item with a key *k*:

  search for *k,* but don't stop until you hit a leaf node
  let L be the leaf node at the end of the search
  if L is a 2-node
      add *k* to L, making it a 3-node



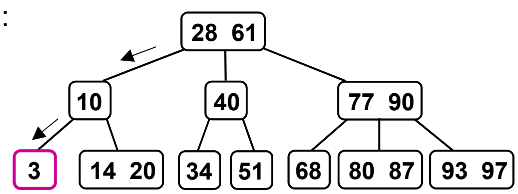
  else if L is a 3-node
      split L into two 2-nodes containing the items with the smallest and largest of: *k,* L's 1st key, L's 2nd key
      the middle item is "sent up" and inserted in L's parent

  example: add 52

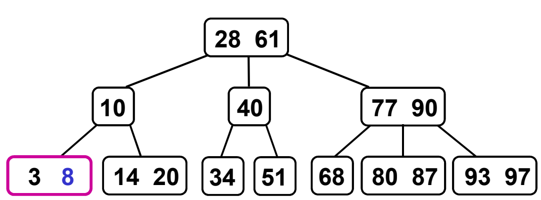


Example 1: Insert 8

- Search for 8:

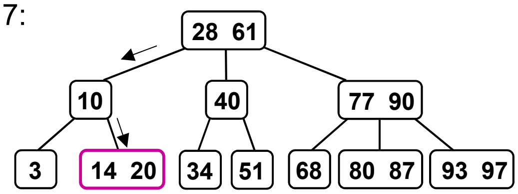


- Add 8 to the leaf node, making it a 3-node:

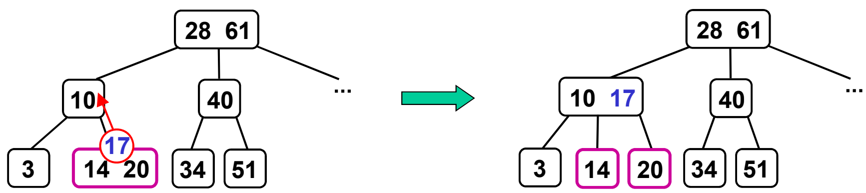


Example 2: Insert 17

- Search for 17:

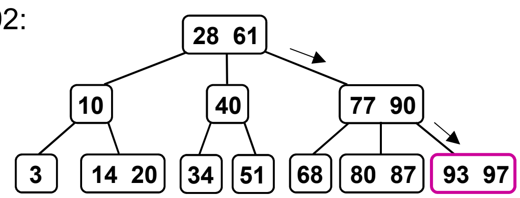


- Split the leaf node, and send up the middle of 14, 17, 20 and insert it the leaf node's parent:

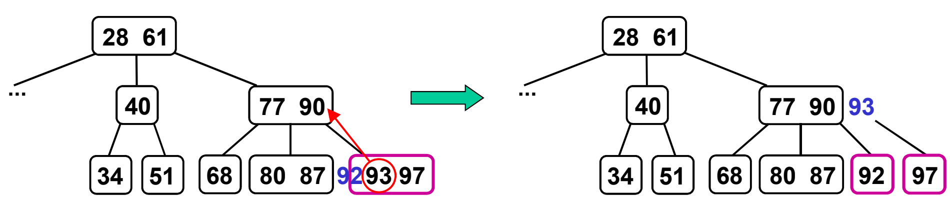


Example 3: Insert 92

- Search for 92:



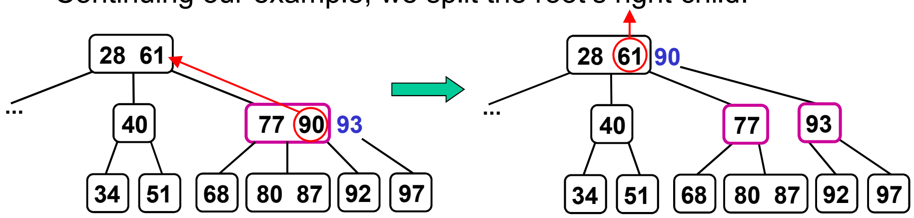
- Split the leaf node, and send up the middle of 92, 93, 97 and insert it the leaf node's parent:



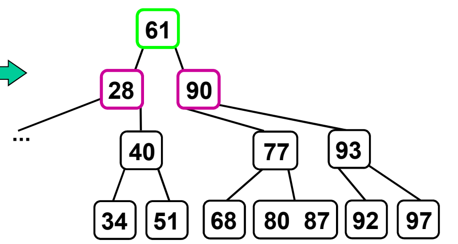
- In this case, the leaf node's parent is also a 3-node, so we need to split is as well

Splitting the Root Node

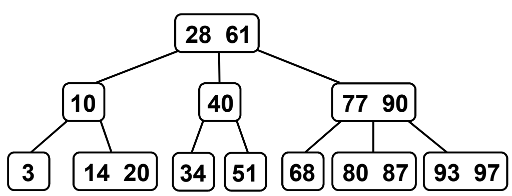
- If an item propagates up to the root node, and the root is a 3-node, we split the root node and create a new, 2-node root containing the middle of the three items.
- Continuing our example, we split the root's right child:



- Then we split the root, which increases the tree's height by 1, but the tree is still balanced.
- This is only case in which the tree's height increases.



Efficiency of 2-3 Trees



- A 2-3 tree containing n items has a height <= $\log_2 n$.
- Thus, search and insertion are both $O(\log n)$.
  - a search visits at most $\log_2 n$ nodes
  - an insertion begins with a search; in the worst case, it goes all the way back up to the root performing splits, so it visits at most $2\log_2 n$ nodes
- Deletion is tricky – you may need to coalesce nodes! However, it also has a time complexity of $O(\log n)$.
- Thus, we can use 2-3 trees for a $O(\log n)$-time data dictionary.

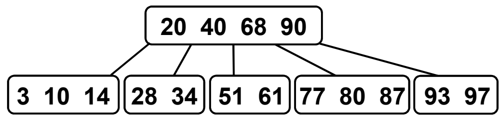
External Storage

- The balanced trees that we've covered don't work well if you want to store the data dictionary externally – i.e., on disk.
- Key facts about disks:
  - data is transferred to and from disk in units called *blocks,* which are typically 4 or 8 KB in size
  - disk accesses are slow!
    - reading a block takes ~10 milliseconds (10^{-3} sec)
    - vs. reading from memory, which takes ~10 nanoseconds
    - in 10 ms, a modern CPU can perform millions of operations!

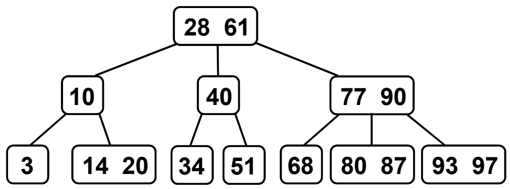
B-Trees

- A B-tree of order *m* is a tree in which each node has:
  - at most 2*m* entries (and, for internal nodes, 2*m* + 1 children)
  - at least *m* entries (and, for internal nodes, *m* + 1 children)
  - exception: the root node may have as few as 1 entry
  - a 2-3 tree is essentially a B-tree of order 1
- To minimize the number of disk accesses, we make *m* as large as possible.
  - each disk read brings in more items
  - the tree will be shorter (each level has more nodes), and thus searching for an item requires fewer disk reads
- A large value of *m* doesn't make sense for a memory-only tree, because it leads to many key comparisons per node.
- These comparisons are less expensive than accessing the disk, so large values of *m* make sense for on-disk trees.

Example: a B-Tree of Order 2



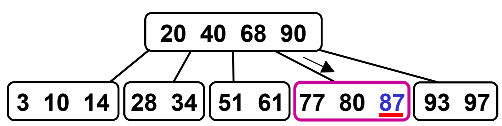
- Order 2: at most 4 data items per node (and at most 5 children)
- The above tree holds the same keys as one of our earlier 2-3 trees, which is shown again below:



- We used the same order of insertion to create both trees:
  51, 3, 40, 77, 20, 10, 34, 28, 61, 80, 68, 93, 90, 97, 87, 14
- For extra practice, see if you can reproduce the trees!

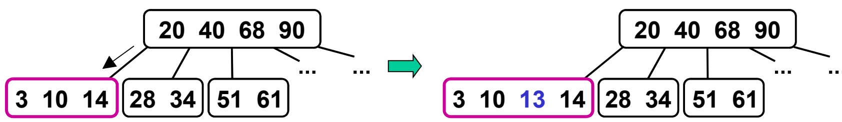
Search in B-Trees

- Similar to search in a 2-3 tree.
- Example: search for 87



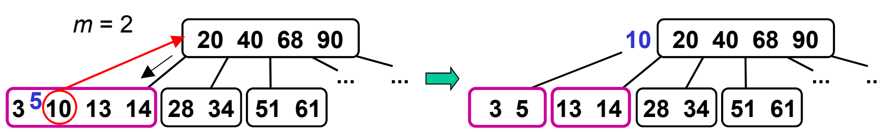
Insertion in B-Trees

- Similar to insertion in a 2-3 tree:
  - search for the key until you reach a leaf node
  - if a leaf node has fewer than 2*m* items, add the item to the leaf node
  - else split the node, dividing up the 2*m* + 1 items:
    - the smallest *m* items remain in the original node
    - the largest *m* items go in a new node
    - send the middle entry up and insert it (and a pointer to the new node) in the parent
- Example of an insertion without a split: insert 13

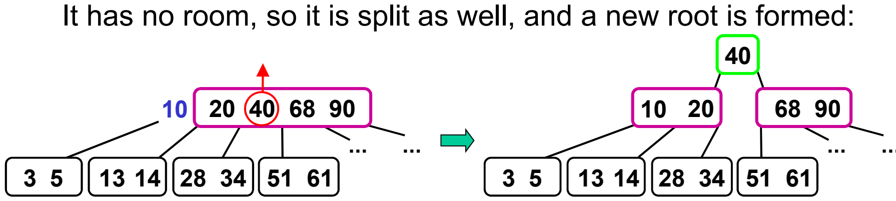


Splits in B-Trees

- Insert 5 into the result of the previous insertion:

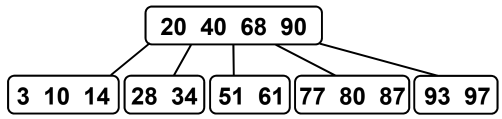


- The middle item (the 10) was sent up to the root.
  It has no room, so it is split as well, and a new root is formed:



- Splitting the root increases the tree's height by 1, but the tree is still balanced. This is only way that the tree's height increases.
- When an internal node is split, its 2*m* + 2 pointers are split evenly between the original node and the new node.

Analysis of B-Trees



- All internal nodes have at least *m* children (actually, at least *m*+1).
- Thus, a B-tree with n items has a height <= $\log_m n$, and search and insertion are both $O(\log_m n)$.
- As with 2-3 trees, deletion is tricky, but it's still logarithmic.

Search Trees: Conclusions

- Binary search trees can be $O(\log n)$, but they can degenerate to $O(n)$ running time if they are out of balance.
- 2-3 trees and B-trees are *balanced* search trees that guarantee $O(\log n)$ performance.
- When data is stored on disk, the most important performance consideration is reducing the number of disk accesses.
- B-trees offer improved performance for on-disk data dictionaries.

Heaps and Priority Queues

Computer Science E-22
Harvard Extension School

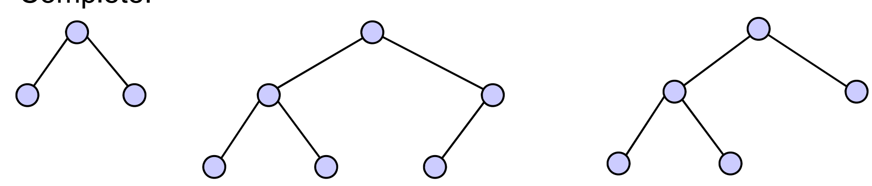
David G. Sullivan, Ph.D.

Priority Queue

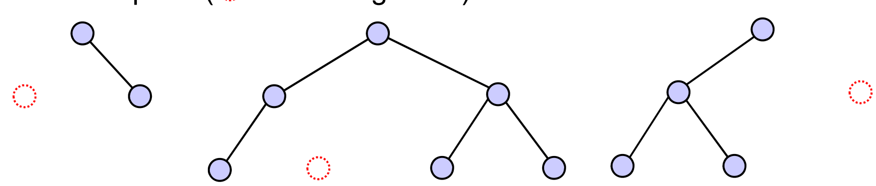
- A *priority queue* is a collection in which each item in the collection has an associated number known as a *priority*.
  - ("Henry Leitner", 10), ("Drew Faust", 15), ("Dave Sullivan", 5)
  - use a higher priority for items that are "more important"
- Example: scheduling a shared resource like the CPU
  - give some processes/applications a higher priority, so that they will be scheduled first and/or more often
- Key operations:
  - *insert:* add an item to the priority queue, positioning it according to its priority
  - *remove:* remove the item with the highest priority
- How can we efficiently implement a priority queue?
  - use a type of binary tree known as a *heap*

Complete Binary Trees

- A binary tree of height *h* is *complete* if:
  - levels 0 through *h* – 1 are fully occupied
  - there are no “gaps” to the left of a node in level *h*
- Complete:

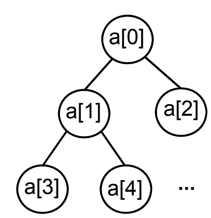


- Not complete (◌ = missing node):

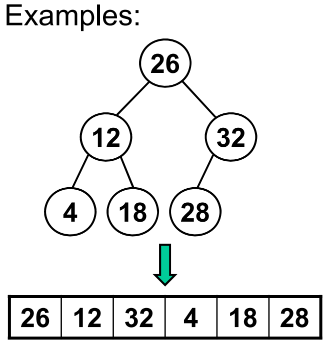


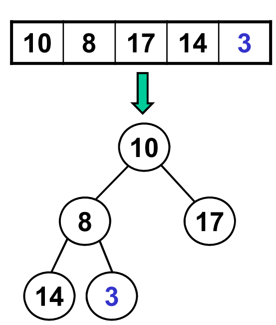
Representing a Complete Binary Tree

- A complete binary tree has a simple array representation.
- The nodes of the tree are stored in the array in the order in which they would be visited by a level-order traversal (i.e., top to bottom, left to right).



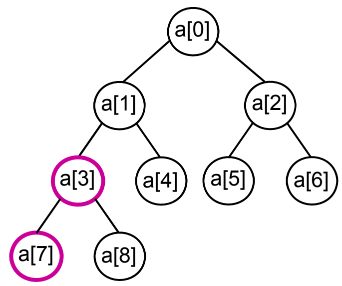
- Examples:





Navigating a Complete Binary Tree in Array Form

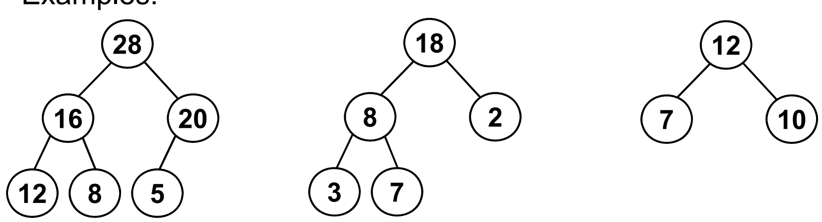
- The root node is in `a[0]`
- Given the node in `a[i]`:
  - its left child is in `a[2*i + 1]`
  - its right child is in `a[2*i + 2]`
  - its parent is in `a[(i - 1)/2]` (using integer division)



- Examples:
  - the left child of the node in `a[1]` is in `a[2*1 + 1]` = `a[3]`
  - the right child of the node in `a[3]` is in `a[2*3 + 2]` = `a[8]`
  - the parent of the node in `a[4]` is in `a[(4-1)/2]` = `a[1]`
  - the parent of the node in `a[7]` is in `a[(7-1)/2]` = `a[3]`

Heaps

- Heap: a complete binary tree in which each interior node is greater than or equal to its children
- Examples:



- The largest value is always at the root of the tree.
- The smallest value can be in *any* leaf node – there's no guarantee about which one it will be.
- Strictly speaking, the heaps that we will use are *max-at-top* heaps. You can also define a *min-at-top* heap, in which every interior node is less than or equal to its children.

How to Compare Objects

- We need to be able to compare items in the heap.
- If those items are objects, we can't just do something like this:

```
if (item1 < item2)
```

Why not?

- Instead, we need to use a method to compare them.

An Interface for Objects That Can Be Compared

- The `Comparable` interface is a built-in generic Java interface:

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

- It is used when defining a class of objects that can be ordered.
- Examples from the built-in Java classes:

```
public class String implements Comparable<String> {
    ...
    public int compareTo(String other) {
        ...
}
public class Integer implements Comparable<Integer> {
    ...
    public int compareTo(Integer other) {
        ...
}
```

An Interface for Objects That Can Be Compared (cont.)

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

- `item1.compareTo(item2)` should return:
  - a negative integer if `item1` "comes before" `item2`
  - a positive integer if `item1` "comes after" `item2`
  - 0 if `item1` and `item2` are equivalent in the ordering
- These conventions make it easy to construct appropriate method calls:

numeric comparison	comparison using `compareTo`
`item1 < item2`	`item1.compareTo(item2) < 0`
`item1 > item2`	`item1.compareTo(item2) > 0`
`item1 == item2`	`item1.compareTo(item2) == 0`

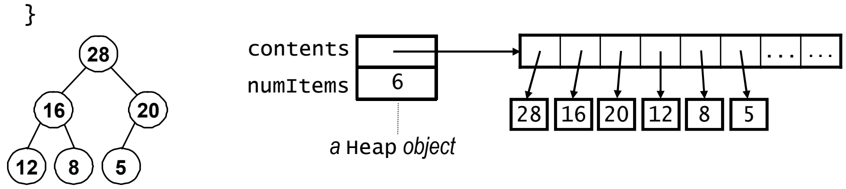
A Class for Items in a Priority Queue

```
public class PQItem implements Comparable<PQItem> {
    // group an arbitrary object with a priority
    private Object data;
    private int priority;
    ...

    public int compareTo(PQItem other) {
        // error-checking goes here…
        return (priority - other.priority);
    }
}
```

- Its `compareTo()` compares `PQItem`s based on their priorities.
- `item1.compareTo(item2)` returns:
  - a negative integer if `item1` has a lower priority than `item2`
  - a positive integer if `item1` has a higher priority than `item2`
  - 0 if they have the same priority

Heap Implementation

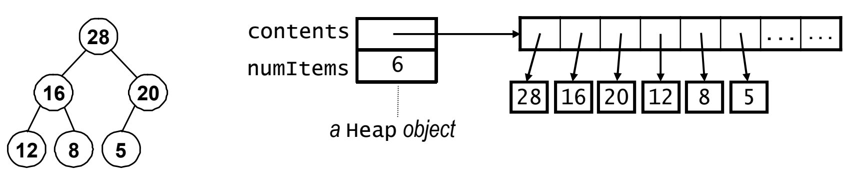
```
public class Heap<T extends Comparable<T>> {
    private T[] contents;
    private int numItems;

    public Heap(int maxSize) {
        contents = (T[])new Comparable[maxSize];
        numItems = 0;
    }
    …
}
```



- Heap is another example of a generic collection class.
  - as usual, T is the type of the elements
  - extends Comparable<T> specifies T must implement Comparable<T>
  - must use Comparable (not Object) when creating the array

Heap Implementation (cont.)

```
public class Heap<T extends Comparable<T>> {
    private T[] contents;
    private int numItems;

    …
}
```



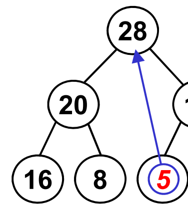
- The picture above is a heap of integers:

  ```
  Heap<Integer> myHeap = new Heap<Integer>(20);
  ```
  - works because Integer implements Comparable<Integer>
  - could also use String or Double
- For a priority queue, we can use objects of our PQItem class:

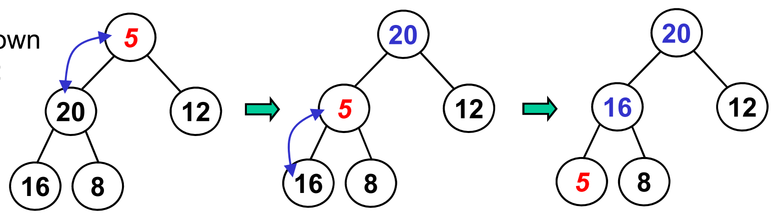
  ```
  Heap<PQItem> pqueue = new Heap<PQItem>(50);
  ```

Removing the Largest Item from a Heap

- Remove and return the item in the root node.
- In addition, we need to move the largest remaining item to the root, while maintaining a complete tree with each node >= children
- Algorithm:
  1. make a copy of the largest item
  2. move the last item in the heap to the root
  3. “sift down” the new root item until it is >= its children (or it’s a leaf)
  4. return the largest item



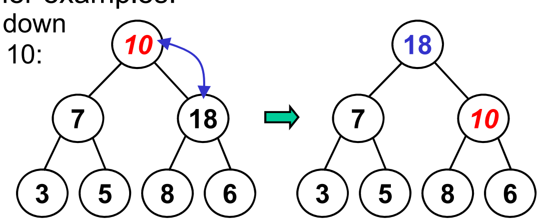
sift down the 5:



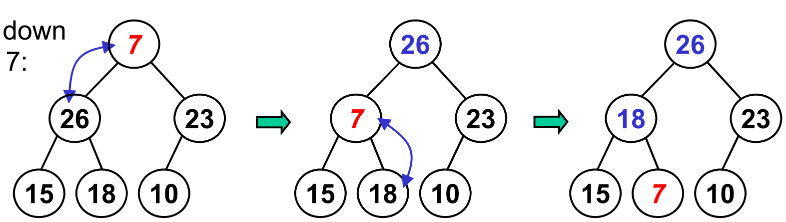
Sifting Down an Item

- To sift down item *x* (i.e., the item whose key is *x*):
  1. compare *x* with the larger of the item’s children, *y*
  2. if *x* < *y*, swap *x* and *y* and repeat
- Other examples:

sift down the 10:



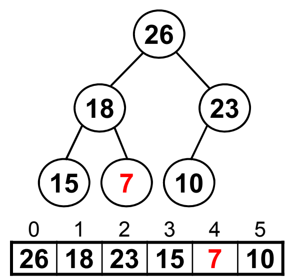
sift down the 7:



siftDown() Method

```
private void siftDown(int i) {
    T toSift = contents[i];

    int parent = i;
    int child = 2 * parent + 1;
    while (child < numItems) {
        // If the right child is bigger, compare with it.
        if (child < numItems - 1  &&
          contents[child].compareTo(contents[child + 1]) < 0)
            child = child + 1;

        if (toSift.compareTo(contents[child]) >= 0)
            break;  // we're done

        // Move child up and move down one level in the tree.
        contents[parent] = contents[child];
        parent = child;
        child = 2 * parent + 1;
    }

    contents[parent] = toSift;
}
```

- We don't actually swap items. We wait until the end to put the sifted item in place.

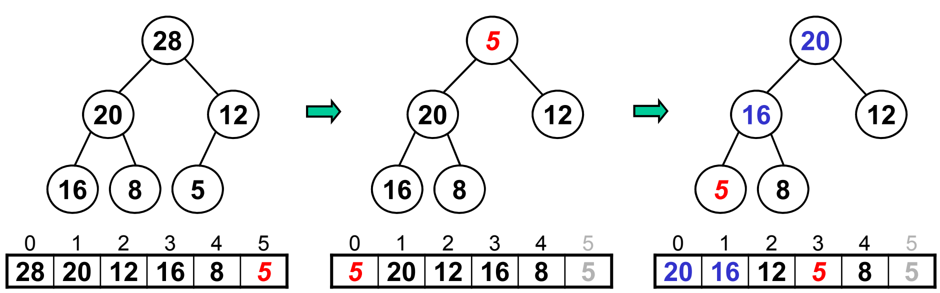


toSift: 7

parent	child
0	1
1	3
1	4
4	9

remove() Method

```
public T remove() {
    T toRemove = contents[0];

    contents[0] = contents[numItems - 1];
    numItems--;
    siftDown(0);

    return toRemove;
}
```



numItems: **6**
toRemove: **28**

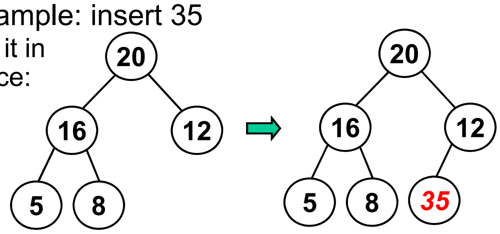
numItems: **5**
toRemove: **28**

numItems: **5**
toRemove: **28**

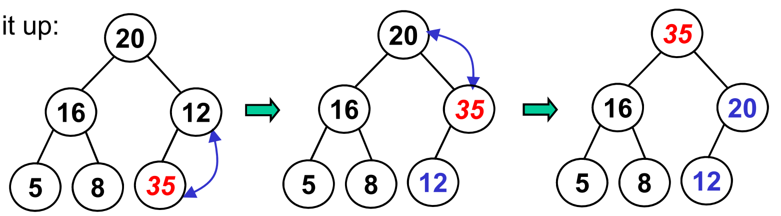
Inserting an Item in a Heap

- Algorithm:
  1. put the item in the next available slot (grow array if needed)
  2. “sift up” the new item
     until it is <= its parent (or it becomes the root item)
- Example: insert 35

put it in place:

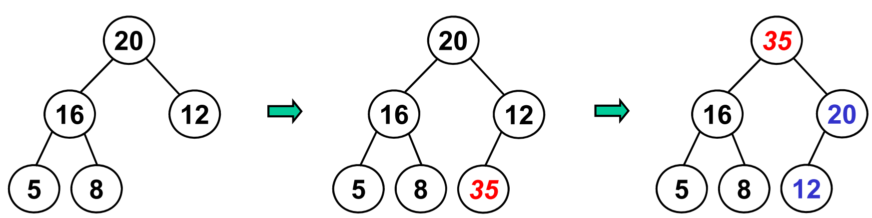


sift it up:



`insert()` Method

```
public void insert(T item) {
    if (numItems == contents.length) {
        // code to grow the array goes here…
    }
    contents[numItems] = item;
    siftUp(numItems);
    numItems++;
}
```



0	1	2	3	4	5
20	16	12	5	8	

numItems: **5**
item: **35**

0	1	2	3	4	5
20	16	12	5	8	35

numItems: **5**
item: **35**

0	1	2	3	4	5
35	16	20	5	8	12

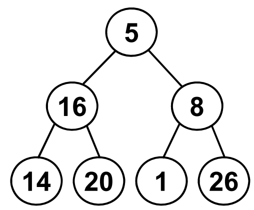
numItems: **6**

Converting an Arbitrary Array to a Heap

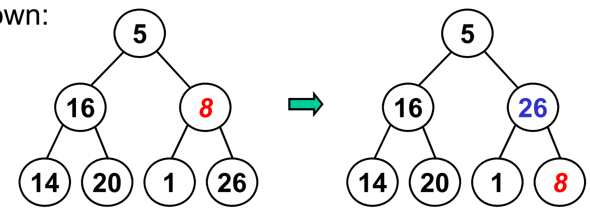
- Algorithm to convert an array with n items to a heap:
  1. start with the parent of the last element:
     `contents[i]`, where `i = ((n – 1) – 1)/2 = (n – 2)/2`
  2. sift down `contents[i]` and all elements to its left

- Example:

0	1	2	3	4	5	6
5	16	8	14	20	1	26

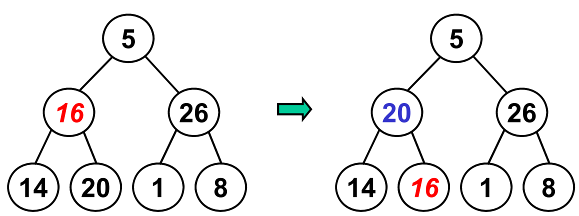


- Last element's parent = `contents[(7 – 2)/2]` = `contents[2]`.
  Sift it down:

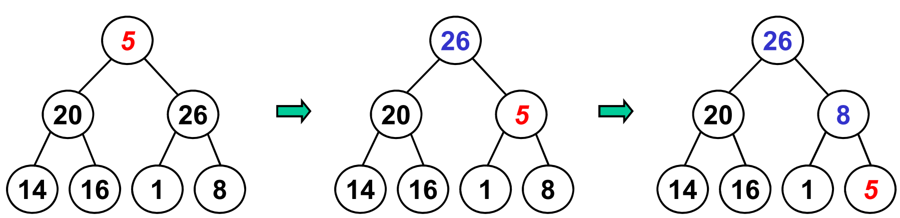


Converting an Array to a Heap (cont.)

- Next, sift down `contents[1]`:



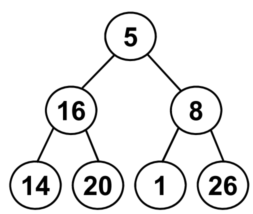
- Finally, sift down `contents[0]`:



Creating a Heap from an Array

```
public class Heap<T extends Comparable<T>> {
    private T[] contents;
    private int numItems;
    ...
    public Heap(T[] arr) {
        // Note that we don't copy the array!
        contents = arr;
        numItems = arr.length;
        makeHeap();
    }

    private void makeHeap() {
        int last = contents.length - 1;
        int parentOfLast = (last - 1)/2;
        for (int i = parentOfLast; i >= 0; i--)
            siftDown(i);
    }
    ...
}
```

Time Complexity of a Heap



- A heap containing n items has a height <= $\log_2 n$.
- Thus, removal and insertion are both $O(\log n)$.
  - remove: go down at most $\log_2 n$ levels when sifting down from the root, and do a constant number of operations per level
  - insert: go up at most $\log_2 n$ levels when sifting up to the root, and do a constant number of operations per level
- This means we can use a heap for a $O(\log n)$-time priority queue.
- Time complexity of creating a heap from an array?

Using a Heap to Sort an Array

- Recall selection sort: it repeatedly finds the smallest remaining element and swaps it into place:

0	1	2	3	4	5	6
5	16	8	14	20	1	26

0	1	2	3	4	5	6
1	16	8	14	20	5	26

0	1	2	3	4	5	6
1	*5*	8	14	20	16	26

- It isn't efficient ($O(n^2)$), because it performs a linear scan to find the smallest remaining element ($O(n)$ steps per scan).
- Heapsort is a sorting algorithm that repeatedly finds the *largest* remaining element and puts it in place.
- It *is* efficient ($O(n \log n)$), because it turns the array into a heap, which means that it can find and remove the largest remaining element in $O(\log n)$ steps.

Heapsort

```
public class HeapSort {
    public static <T extends Comparable<T>> void
    heapSort(T[] arr) {
        // Turn the array into a max-at-top heap.
        Heap<T> heap = new Heap<T>(arr);

        int endUnsorted = arr.length - 1;
        while (endUnsorted > 0) {
            // Get the largest remaining element and put it
            // at the end of the unsorted portion of the array.
            T largestRemaining = heap.remove();
            arr[endUnsorted] = largestRemaining;

            endUnsorted--;
        }
    }
}
```

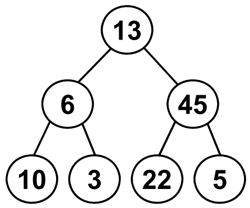
- We define a *generic method*, with a type variable in the method header. It goes right before the method's return type.
- T is a placeholder for the type of the array.
  - can be any type T that implements Comparable<T>.

Heapsort Example

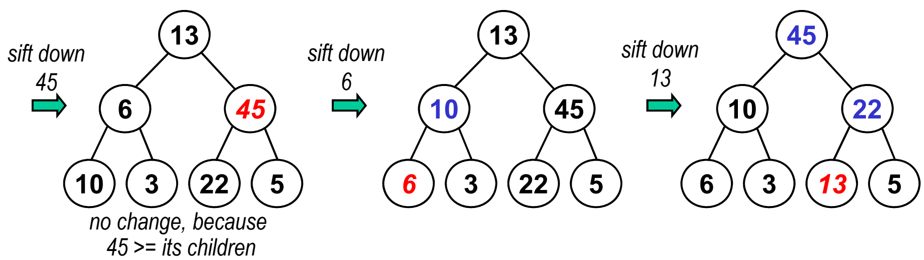
- Sort the following array:

0	1	2	3	4	5	6
13	6	45	10	3	22	5

- Here's the corresponding complete tree:

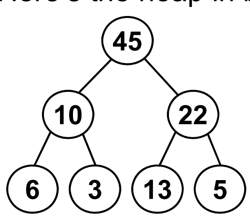


- Begin by converting it to a heap:



Heapsort Example (cont.)

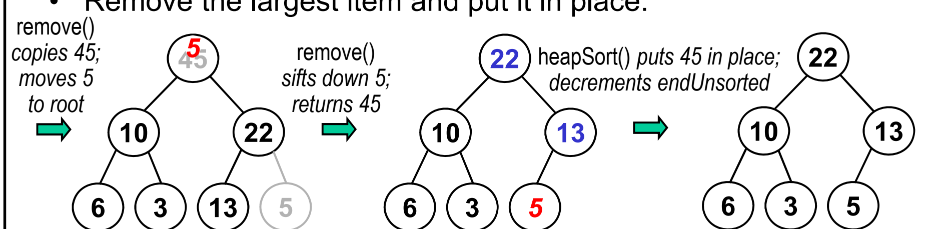
- Here's the heap in both tree and array forms:



0	1	2	3	4	5	6
45	10	22	6	3	13	5

endUnsorted: **6**

- Remove the largest item and put it in place:



toRemove: **45**

0	1	2	3	4	5	6
22	10	13	6	3	5	5

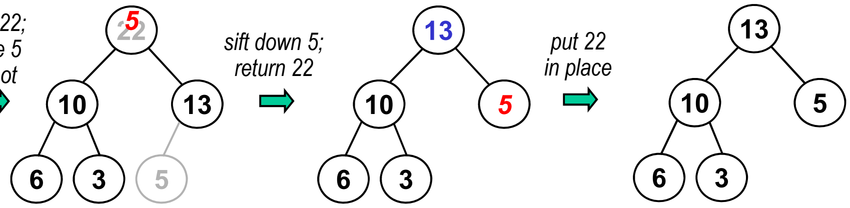
endUnsorted: **6**
largestRemaining: **45**

0	1	2	3	4	5	6
22	10	13	6	3	5	45

endUnsorted: **5**

Heapsort Example (cont.)

copy 22; move 5 to root → *sift down 5; return 22* → *put 22 in place*



toRemove: **22**

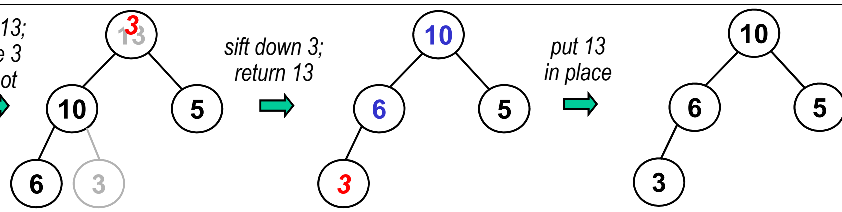
0	1	2	3	4	5	6
13	10	*5*	6	3	5	45

endUnsorted: **5**
largestRemaining: **22**

0	1	2	3	4	5	6
13	10	5	6	3	22	45

endUnsorted: **4**

copy 13; move 3 to root → *sift down 3; return 13* → *put 13 in place*



toRemove: **13**

0	1	2	3	4	5	6
10	6	5	*3*	3	22	45

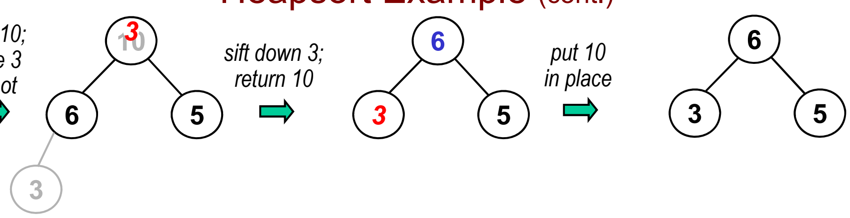
endUnsorted: **4**
largestRemaining: **13**

0	1	2	3	4	5	6
10	6	5	3	13	22	45

endUnsorted: **3**

Heapsort Example (cont.)

copy 10; move 3 to root → *sift down 3; return 10* → *put 10 in place*



toRemove: **10**

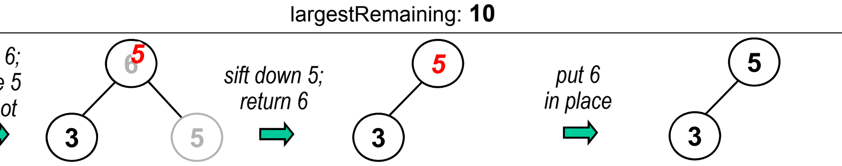
0	1	2	3	4	5	6
6	*3*	5	3	13	22	45

endUnsorted: **3**
largestRemaining: **10**

0	1	2	3	4	5	6
6	3	5	10	13	22	45

endUnsorted: **2**

copy 6; move 5 to root → *sift down 5; return 6* → *put 6 in place*



toRemove: **6**

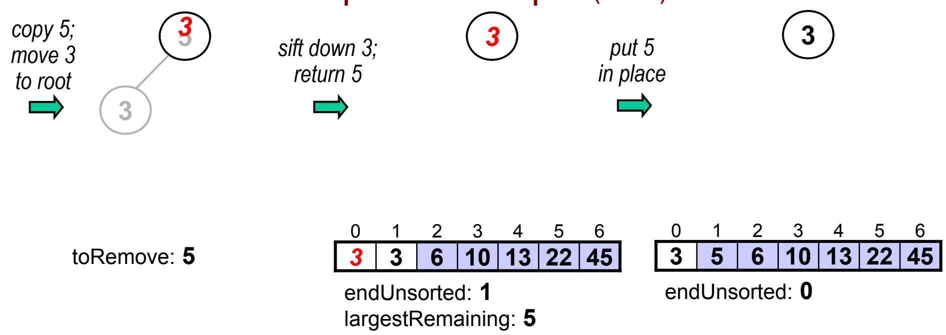
0	1	2	3	4	5	6
5	3	5	10	13	22	45

endUnsorted: **2**
largestRemaining: **6**

0	1	2	3	4	5	6
5	3	6	10	13	22	45

endUnsorted: **1**

Heapsort Example (cont.)



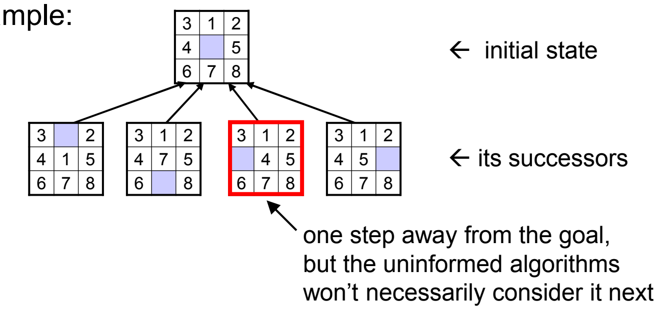
How Does Heapsort Compare?

algorithm	best case	avg case	worst case	extra memory
selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Shell sort	$O(n \log n)$	$O(n^{1.5})$	$O(n^{1.5})$	$O(1)$
bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(1)$
mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
heapsort	**$O(n \log n)$**	**$O(n \log n)$**	**$O(n \log n)$**	**$O(1)$**

- Heapsort matches mergesort for the best worst-case time complexity, but it has better space complexity.
- Insertion sort is still best for arrays that are almost sorted.
  - heapsort will scramble an almost sorted array before sorting it
- Quicksort is still typically fastest in the average case.

State-Space Search Revisited

- Earlier, we considered three algorithms for state-space search:
  - breadth-first search (BFS)
  - depth-first search (DFS)
  - iterative-deepening search (IDS)
- These are all *uninformed* search algorithms.
  - always consider the states in a certain order
  - do not consider how close a given state is to the goal
- 8 Puzzle example:



Informed State-Space Search

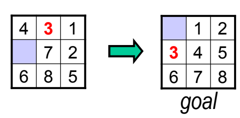
- *Informed* search algorithms attempt to consider more promising states first.
- These algorithms associate a *priority* with each successor state that is generated.
  - base priority on an estimate of nearness to a goal state
  - when choosing the next state to consider, select the one with the highest priority
- Use a priority queue to store the yet-to-be-considered search nodes.

State-Space Search: Estimating the Remaining Cost

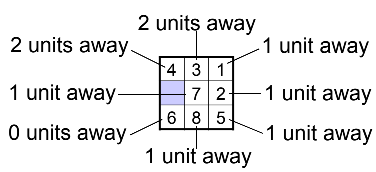
- The priority of a state is based on the *remaining cost* – i.e., the cost of getting from the state to the closest goal state.
  - for the 8 puzzle, remaining cost = # of steps to closest goal
- For most problems, we can't determine the exact remaining cost.
  - if we could, we wouldn't need to search!
- Instead, we estimate the remaining cost using a *heuristic function* h(x) that takes a state x and computes a cost estimate for it.
  - heuristic = rule of thumb
- To find optimal solutions, we need an *admissable* heuristic – one that never overestimates the remaining cost.

Heuristic Function for the Eight Puzzle

- Manhattan distance = horizontal distance + vertical distance
  - example: For the board at right, the Manhattan distance of the $\boxed{3}$ tile from its position in the goal state = 1 column + 1 row = 2



- Use h(x) = sum of the Manhattan distances of the tiles in x from their positions in the goal state
  - for our example:

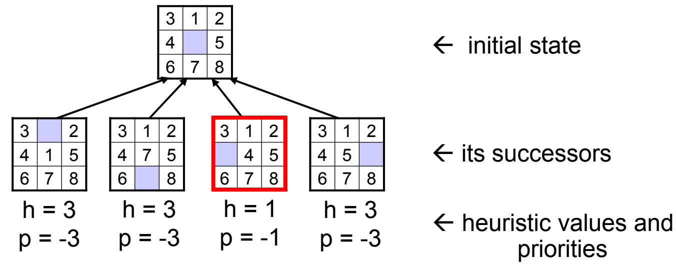


h(x) = 1 + 1 + 2 + 2
+ 1 + 0 + 1 + 1 = 9

- This heuristic is admissible because each of the operators (move blank up, move blank down, etc.) moves a single tile a distance of 1, so it will take at least h(x) steps to reach the goal.

Greedy Search

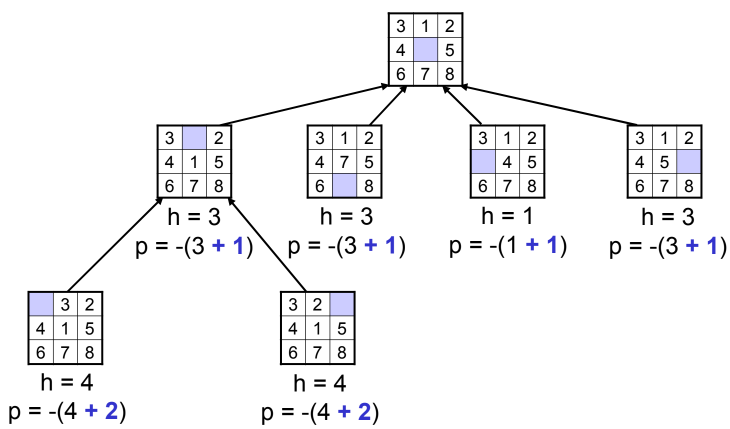
- Priority of state x, p(x) = −1 * h(x)
  - mult. by −1 so states closer to the goal have higher priorities



- Greedy search would consider the highlighted successor before the other successors, because it has the highest priority.
- Greedy search is:
  - incomplete: it may not find a solution
    - it could end up going down an infinite path
  - not optimal: the solution it finds may not have the lowest cost
    - it fails to consider the cost of getting *to* the current state

A* Search

- Priority of state x, p(x) = −1 * (h(x) + g(x))
  where g(x) = the cost of getting from the initial state to x



- Incorporating g(x) allows A* to find an optimal solution – one with the minimal *total* cost.

Characteristics of A*

- It is complete and optimal.
  - provided that h(x) is admissable, and that g(x) increases or stays the same as the depth increases
- Time and space complexity are still typically exponential in the solution depth, d – i.e., the complexity is $O(b^d)$ for some value b.
- However, A* typically visits far fewer states than other optimal state-space search algorithms.

solution depth	**iterative deepening**	**A* w/ Manhattan dist. heuristic**
4	112	12
8	6384	25
12	364404	73
16	did not complete	211
20	did not complete	676

Source: Russell & Norvig, *Artificial Intelligence: A Modern Approach*, Chap. 4.

The numbers shown are the average number of search nodes visited in 100 randomly generated problems for each solution depth.

The searches do *not* appear to have excluded previously seen states.

- Memory usage can be a problem, but it's possible to address it.

Implementing Informed Search

- Add new subclasses of the abstract `Searcher` class.
- For example:

```
public class GreedySearcher extends Searcher {
    private Heap<PQItem> nodePQueue;

    public void addNode(SearchNode node) {
        nodePQueue.insert(
          new PQItem(node, -1 * node.getCostToGoal()));
    }
    …
```

Hash Tables

Computer Science S-111
Harvard Extension School

David G. Sullivan, Ph.D.

Data Dictionary Revisited

- We've considered several data structures that allow us to store and search for data items using their keys fields:

data structure	***searching for an item***	***inserting an item***
a list implemented using an array	*O*(log n) using binary search	*O*(n)
a list implemented using a linked list	*O*(n) using linear search	*O*(n)
binary search tree		
balanced search trees (2-3 tree, B-tree, others)		

- Today, we'll look at hash tables, which allow us to do better than *O*(log n).

Ideal Case: Searching = Indexing

- The optimal search and insertion performance is achieved when we can treat the key as an index into an array.
- Example: storing data about members of a sports team
  - key = jersey number (some value from 0-99).
  - class for an individual player's record:
    ```
    public class Player {
        private int jerseyNum;
        private String firstName;
        …
    }
    ```
  - store the player records in an array:
    ```
    Player[] teamRecords = new Player[100];
    ```
- In such cases, we can perform both search and insertion in *O*(1) time. For example:

```
public Player search(int jerseyNum) {
    return teamRecords[jerseyNum];
}
```

Hashing: Turning Keys into Array Indices

- In most real-world problems, indexing is not as simple as it is in the sports-team example. Why?
  -
  -
  -
- To handle these problems, we perform *hashing*:
  - use a *hash function* to convert the keys into array indices
    `"Sullivan"` → `18`
  - use techniques to handle cases in which multiple keys are assigned the same hash value
- The resulting data structure is known as a *hash table*.

Hash Functions

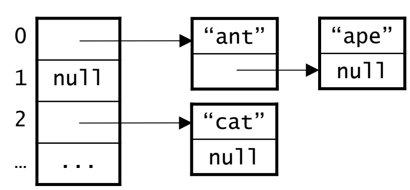
- A hash function defines a mapping from the set of possible keys to the set of integers.
- We then use the modulus operator to get a valid array index.

key value ⇨ **hash function** ⇨ integer $\xrightarrow{\%}$ integer in [0, n – 1]
(n = array length)

- Here's a very simple hash function for keys of lower-case letters:
  h(key) = Unicode value of first char – Unicode value of 'a'
  - examples:
    h("ant") = Unicode for 'a' – Unicode for 'a' = 0
    h("cat") = Unicode for 'c' – Unicode for 'a' = 2
- h(key) is known as the key's *hash code.*
- A *collision* occurs when items with different keys are assigned the same hash code.

Dealing with Collisions I: Separate Chaining

- If multiple items are assigned the same hash code, we "chain" them together.
- Each position in the hash table serves as a *bucket* that is able to store multiple data items.
- Two implementations:
  1. each bucket is itself an array
     - disadvantages:
       - large buckets can waste memory
       - a bucket may become full; *overflow* occurs when we try to add an item to a full bucket
  2. each bucket is a linked list
     - disadvantage:
       - the references in the nodes use additional memory



Dealing with Collisions II: Open Addressing

- When the position assigned by the hash function is occupied, find another open position.
- Example: “wasp” has a hash code of 22, but it ends up in position 23, because position 22 is occupied.
- We will consider three ways of finding an open position – a process known as *probing*.
- The hash table also performs probing to search for an item.
  - example: when searching for “wasp”, we look in position 22 and then look in position 23
  - we can only stop a search when we reach an empty position

0	“ant”
1	
2	“cat”
3	
4	“emu”
5	
6	
7	
…	...
22	“wolf”
23	“wasp”
24	“yak”
25	“zebra”

Linear Probing

- Probe sequence: h(key), h(key) + 1, h(key) + 2, ,
  wrapping around as necessary.
- Examples:
  - “ape” (h = 0) would be placed in position 1, because position 0 is already full.
  - “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - where would “zebu” end up?
- Advantage: if there is an open position, linear probing will eventually find it.
- Disadvantage: “clusters” of occupied positions develop, which tends to increase the lengths of subsequent probes.
  - probe length = the number of positions considered during a probe

0	“ant”
1	“ape”
2	“cat”
3	“bear”
4	“emu”
5	
6	
7	
…	...
22	“wolf”
23	“wasp”
24	“yak”
25	“zebra”

Quadratic Probing

- Probe sequence: h(key), h(key) + 1, h(key) + 4, h(key) + 9, ,
wrapping around as necessary.
  - the offsets are perfect squares: h + 1^2, h + 2^2, h + 3^2,
- Examples:
  - "ape" (h = 0): try 0, 0 + 1 – open!
  - "bear" (h = 1): try 1, 1 + 1, 1 + 4 – open!
  - "zebu"?
- Advantage: reduces clustering
- Disadvantage: it may fail to find an existing open position. For example:

table size = 10
x = occupied

trying to insert a key with h(key) = 0

offsets of the probe sequence in italics

0	x		5	x	*25*
1	x	*1 81*	6	x	*16 36*
2			7		
3			8		
4	x	*4 64*	9	x	*9 49*

0	"ant"
1	"ape"
2	"cat"
3	
4	"emu"
5	"bear"
6	
7	
…	...
22	"wolf"
23	"wasp"
24	"yak"
25	"zebra"

Double Hashing

- Use two hash functions:
  - h1 computes the hash code
  - h2 computes the increment for probing
  - probe sequence: h1, h1 + h2, h1 + 2*h2,
- Examples:
  - h1 = our previous h
  - h2 = number of characters in the string
  - "ape" (h1 = 0, h2 = 3): try 0, 0 + 3 – open!
  - "bear" (h1 = 1, h2 = 4): try 1 – open!
  - "zebu"?
- Combines the good features of linear and quadratic probing:
  - reduces clustering
  - will find an open position if there is one, provided the table size is a prime number

0	"ant"
1	"bear"
2	"cat"
3	"ape"
4	"emu"
5	
6	
7	
…	...
22	"wolf"
23	"wasp"
24	"yak"
25	"zebra"

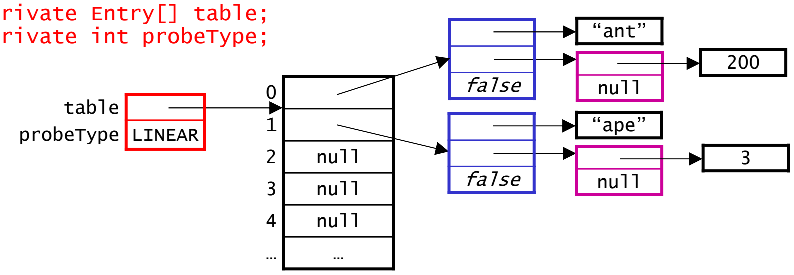
Removing Items Under Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”
  - search for “ape”: try 0, 0 + 1 – no item
  - search for “bear”: try 1 – no item, but “bear” is further down in the table
- When we remove an item from a position, we need to leave a special value in that position to indicate that an item was removed.
- Three types of positions: occupied, empty, “removed”.
- We stop probing when we encounter an empty position, but not when we encounter a removed position.
- We can insert items in either empty or removed positions.

0	“ant”
1	
2	“cat”
3	“bear”
4	“emu”
5	
…	...
22	“wolf”
23	“wasp”
24	“yak”
25	“zebra”

Implementation

```
public class HashTable {
    private class Entry {
        private String key;
        private LLList valueList;
        private boolean hasBeenRemoved;
        …
    }
    …
    private Entry[] table;
    private int probeType;
}
```



- We use a private inner class for the entries in the hash table.
- To handle duplicates, we maintain a list of values for each key.
- When we remove a key and its values, we set the `Entry`’s `hasBeenRemoved` field to `true`; this indicates that the position is a removed position.

Probing Using Double Hashing

```
private int probe(String key) {
    int i = h1(key);     // first hash function
    int h2 = h2(key);    // second hash function

    // keep probing until we get an empty position or match
    // (write this together)




    return i;
}
```

- We'll assume that removed positions have a key of null.
  - thus, for non-empty positions, it's always okay to compare the probe key with the key in the `Entry`

Avoiding an Infinite Loop

- The while loop in our probe method could lead to an infinite loop.

```
while (table[i] != null && !key.equals(table[i].key)) {
    i = (i + h2) % table.length;
}
```

- When would this happen?

- We can stop probing after checking n positions (n = table size), because the probe sequence will just repeat after that point.
  - for quadratic probing:
    $(h1 + n^2) \% n = h1 \% n$
    $(h1 + (n+1)^2) \% n = (h1 + n^2 + 2n + 1) \% n = (h1 + 1) \% n$
  - for double hashing:
    $(h1 + n*h2) \% n = h1 \% n$
    $(h1 + (n+1)*h2) \% n = (h1 + n*h2 + h2) \% n = (h1 + h2) \% n$

Avoiding an Infinite Loop (cont.)

```
private int probe(String key) {
    int i = h1(key);    // first hash function
    int h2 = h2(key);   // second hash function
    int positionsChecked = 1;

    // keep probing until we get an
    // empty position or a match
    while (table[i] != null && !key.equals(table[i].key)) {
        if (positionsChecked == table.length)
            return -1;
        i = (i + h2) % table.length;
        positionsChecked++;
    }

    return i;
}
```

Handling the Other Types of Probing

```
private int probe(String key) {
    int i = h1(key);    // first hash function
    int h2 = h2(key);   // second hash function
    int positionsChecked = 1;

    // keep probing until we get an
    // empty position or a match
    while (table[i] != null && !key.equals(table[i].key)) {
        if (positionsChecked == table.length)
            return -1;
        i = (i + probeIncrement(positionsChecked, h2))
              % table.length;
        positionsChecked++;
    }

    return i;
}
```

Handling the Other Types of Probing (cont.)

- The `probeIncrement()` method bases the increment on the type of probing:

```
private int probeIncrement(int n, int h2) {
    if (n <= 0)
       return 0;

    switch (probeType) {
    case LINEAR:
        return 1;
    case QUADRATIC:
        return (2*n - 1);
    case DOUBLE_HASHING:
        return h2;
    }
}
```

Handling the Other Types of Probing (cont.)

- For quadratic probing, `probeIncrement(n, h2)` returns
  `2*n - 1`
  Why does this work?
- Recall that for quadratic probing:
  - probe sequence = h1, h1 + 1^2, h1 + 2^2, …
  - nth index in the sequence = h1 + n^2
- The increment used to compute the nth index
  = nth index – (n – 1)st index
  = $(h1 + n^2) - (h1 + (n-1)^2)$
  = $n^2 - (n-1)^2$
  = $n^2 - (n^2 - 2n + 1)$
  = $2n - 1$

Search and Removal

- Both of these methods begin by probing for the key.

```
public LLList search(String key) {
    int i = probe(key);
    if (i == -1 || table[i] == null)
        return null;
    else
        return table[i].valueList;
}

public void remove(String key) {
    int i = probe(key);
    if (i == -1 || table[i] == null)
        return;

    table[i].key = null;
    table[i].valueList = null;
    table[i].hasBeenRemoved = true;
}
```

Insertion

- We begin by probing for the key.
- Several cases:
  1. the key is already in the table (we're inserting a duplicate)
     → add the value to the valueList in the key's Entry
  2. the key is not in the table: three subcases:
     a. encountered 1 or more removed positions while probing
        → put the (key, value) pair in the *first* removed position that we encountered while searching for the key.
        why does this make sense?

     b. no removed position; reached an empty position
        → put the (key, value) pair in the empty position

     c. no removed position or empty position encountered
        → overflow; throw an exception

Insertion (cont.)

- To handle the special cases, we give this method its own implementation of probing:

```
void insert(String key, int value) {
    int i = h1(key);
    int h2 = h2(key);
    int positionsChecked = 1;
    int firstRemoved = -1;

    while (table[i] != null && !key.equals(table[i].key)) {
        if (table[i].hasBeenRemoved && firstRemoved == -1)
            firstRemoved = i;
        if (positionsChecked == table.length)
            break;
        i = (i + probeIncrement(positionsChecked, h2))
              % table.length;
        positionsChecked++;
    }

    // deal with the different cases (see next slide)
}
```

- firstRemoved remembers the first removed position encountered

Insertion (cont.)

```
void insert(String key, int value) {
    ...
    int firstRemoved = -1;

    while (table[i] != null && !key.equals(table[i].key) {
        if (table[i].hasBeenRemoved && firstRemoved == -1)
            firstRemoved = i;
        if (++positionsChecked == table.length)
            break;
        i = (i + h2) % table.length;
    }

    // deal with the different cases
    if (table[i] != null && key.equals(table[i].key))  // 1
        table[i].valueList.addItem(value, 0);
    else if (firstRemoved != -1)                        // 2a
        table[firstRemoved] = new Entry(key, value);
    else if (table[i] == null)                          // 2b
        table[i] = new Entry(key, value);
    else throw an exception…                            // 2c
}
```

Tracing Through Some Examples

- Start with the hashtable at right with:
  - double hashing
  - our earlier hash functions h1 and h2
- Perform the following operations:
  - insert "bear"
  - insert "bison"
  - insert "cow"
  - delete "emu"
  - search "eel"
  - insert "bee"

0	“ant”
1	
2	“cat”
3	
4	"emu"
5	"fox"
6	
7	
8	
9	
10	

Dealing with Overflow

- Overflow = can't find a position for an item
- When does it occur?
  - linear probing:
  - quadratic probing:
    -
    -
  - double hashing:
    - if the table size is a prime number: same as linear
    - if the table size is not a prime number: same as quadratic
- To avoid overflow (and reduce search times), grow the hash table when the percentage of occupied positions gets too big.
  - problem: if we're not careful, we can end up needing to rehash **all** of the existing items
  - approaches exist that limit the number of rehashed items

Implementing the Hash Function

- Characteristics of a good hash function:
  1) efficient to compute
  2) uses the entire key
     - changing any char/digit/etc. should change the hash code
  3) distributes the keys more or less uniformly across the table
  4) must be a function!
     - a key must always get the same hash code
- In Java, every object has a `hashCode()` method.
  - the version inherited from `Object` returns a value based on an object's memory location
  - classes can override this version with their own

Hash Functions for Strings: version 1

- h_a = the sum of the characters' Unicode values
- Example: h_a("eat") = 101 + 97 + 116 = 314
- All permutations of a given set of characters get the same code.
  - example: h_a("tea") = h_a("eat")
  - could be useful in a Scrabble game
    - allow you to look up all words that can be formed from a given set of characters
- The range of possible hash codes is very limited.
  - example: hashing keys composed of 1-5 lower-case char's (padded with spaces)
  - 26*27*27*27*27 = over 13 million possible keys
  - smallest code = h_a("a   ") = 97 + 4*32 = 225
    largest code = h_a("zzzzz") = 5*122 = 610

    610 – 225 = 385 codes

Hash Functions for Strings: version 2

- Compute a *weighted* sum of the Unicode values:

  $$h_b = a_0 b^{n-1} + a_1 b^{n-2} + \quad + a_{n-2} b + a_{n-1}$$

  where a_i = Unicode value of the ith character
  b = a constant
  n = the number of characters

- Multiplying by powers of b allows the *positions* of the characters to affect the hash code.
  - different permutations get different codes

- We may get arithmetic overflow, and thus the code may be negative. We adjust it when this happens.

- Java uses this hash function with b = 31 in the `hashCode()` method of the `String` class.

Hash Table Efficiency

- In the best case, search and insertion are *O*(1).

- In the worst case, search and insertion are linear.
  - open addressing: *O*(m), where m = the size of the hash table
  - separate chaining: *O*(n), where n = the number of keys

- With good choices of hash function and table size, complexity is generally better than *O*(log n) and approaches *O*(1).

- *load factor* = # keys in table / size of the table.
  To prevent performance degradation:
  - open addressing: try to keep the load factor < 1/2
  - separate chaining: try to keep the load factor < 1

- Time-space tradeoff: bigger tables have better performance, but they use up more memory.

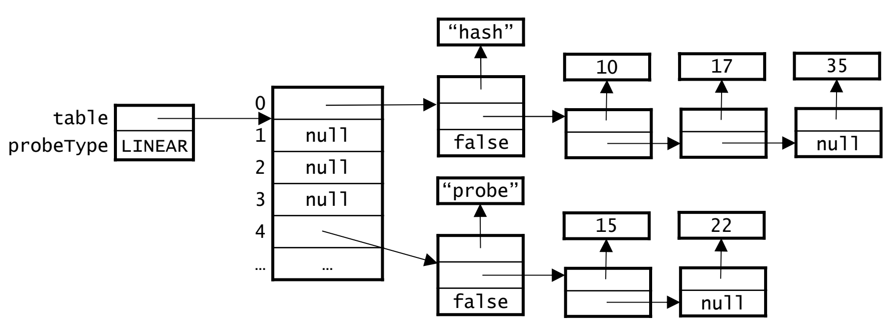
Hash Table Limitations

- It can be hard to come up with a good hash function for a particular data set.
- The items are not ordered by key. As a result, we can't easily:
  - print the contents in sorted order
  - perform a range search
  - perform a rank search – get the kth largest item

  We *can* do all of these things with a search tree.

Application of Hashing: Indexing a Document

- Read a text document from a file and create an index of the line numbers on which each word appears.
- Use a hash table to store the index:
  - key = word
  - values = line numbers in which the word appears



- See `WordIndex.java`

Optional: Computing h_b More Efficiently

- Use Horner's method of evaluating a polynomial:
  - $a_0b^{n-1} + a_1b^{n-2} + \quad + a_{n-2}b^{n-1} + a_{n-1}$
    $= (\quad ((a_0b + a_1)b + a_2)b + \quad + a_{n-2})b + a_{n-1}$
  - example: $101*31^2 + 97*31 + 116 = ((101*31 + 97)*31 + 116$
  - here it is in Java for the string `s`:

```
int hash = 0;
for (int i = 0; i < s.length(); i++)
    hash = hash * b + s.charAt(i);
```

- Use the left-shift operator (<<) to multiply by 31:
  - `n << i` shifts the binary representation of `n` left by `i` places
  - `n << i = n * 2`^{i}
  - `n * 31 = n * (32 - 1) = (n * 32) - n = (n << 5) - n`
  - example: `n = 100 =` 0000000001100100_2
    `100 << 5 =` 0000110010000000_2 `= 3200`
    `100 * 31 = 3200 - 100 = 3100`

Optional: Hash Functions for Numeric Keys

- If the keys are `int`s (or a smaller numeric type – e.g., `byte`), we can use the keys themselves as the hash codes.
- If the keys are `long`s or `double`s (64 bits), we could cast the keys to `int`s, but that means that half of the bits in the keys won't contribute to the hash codes.
- Instead, use folding – as we did in h_a for strings.
  - break the 64 bits into two 32-bit pieces
  - combine the pieces by adding them or by applying the exclusive-or operator (^) – see the textbooks for more info.
- Example:

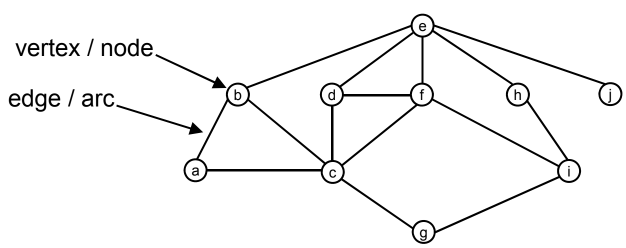
```
long key;
long leftMostBits = key >> 32;  // shift bits right by 32
int hash = (int)(key ^ leftMostBits);
```

Graphs

Computer Science E-22
Harvard Extension School

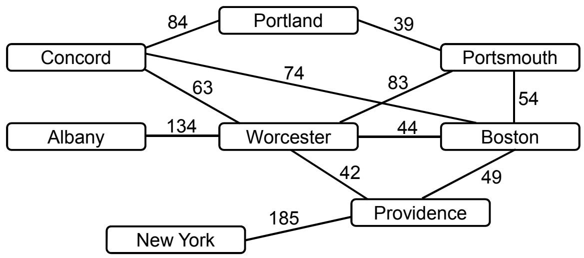
David G. Sullivan, Ph.D.

What is a Graph?



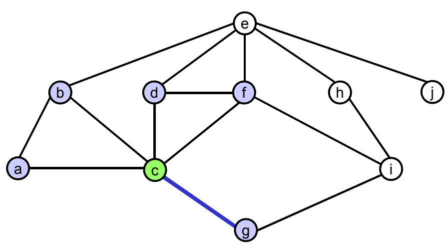
- A graph consists of:
  - a set of *vertices* (also known as *nodes*)
  - a set of *edges* (also known as *arcs*), each of which connects a pair of vertices

Example: A Highway Graph



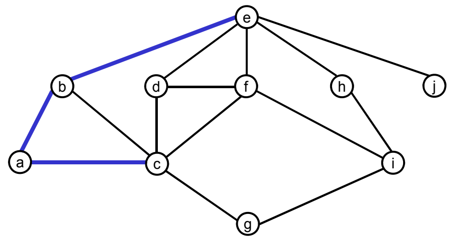
- Vertices represent cities.
- Edges represent highways.
- This is a *weighted* graph, because it has a *cost* associated with each edge.
  - for this example, the costs denote mileage
- We'll use graph algorithms to answer questions like "What is the shortest route from Portland to Providence?"

Relationships Among Vertices

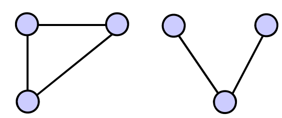


- Two vertices are *adjacent* if they are connected by a single edge.
  - *ex:* c and g are adjacent, but c and i are not
- The collection of vertices that are adjacent to a vertex v are referred to as v's *neighbors.*
  - *ex:* c's neighbors are a, b, d, f, and g

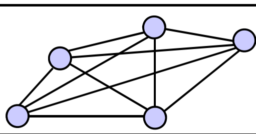
Paths in a Graph



- A *path* is a sequence of edges that connects two vertices.
  - *ex:* the path highlighted above connects c and e
- A graph is *connected* if there is a path between any two vertices.
  - *ex:* the six vertices at right are part of a graph that is *not* connected

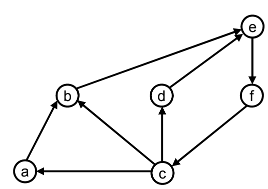


- A graph is *complete* if there is an edge between every pair of vertices.
  - *ex:* the graph at right is complete



Directed Graphs

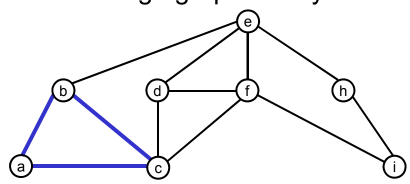
- A *directed* graph has a direction associated with each edge, which is depicted using an arrow:



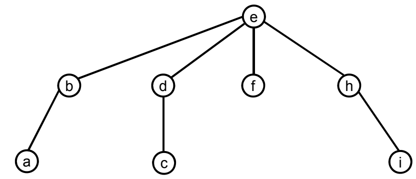
- Edges in a directed graph are often represented as ordered pairs of the form (start vertex, end vertex).
  - *ex:* (a, b) is an edge in the graph above, but (b, a) is not.
- A path in a directed graph is a sequence of edges in which the end vertex of edge i must be the same as the start vertex of edge i + 1.
  - *ex:* { (a, **b**), (**b**, **e**), (**e**, f) } is a valid path.
    { (a, **b**), (**c**, **b**), (**c**, a) } is not.

Trees vs. Graphs

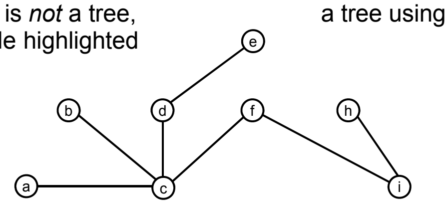
- A tree is a special type of graph.
  - it is connected and undirected
  - it is *acyclic:* there is no path containing distinct edges that starts and ends at the same vertex
  - we usually single out one of the vertices to be the root of the tree, although graph theory does not require this



a graph that is *not* a tree, with one cycle highlighted



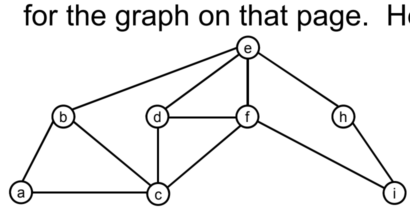
a tree using the same nodes



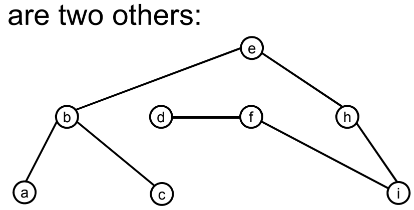
another tree using the same nodes

Spanning Trees

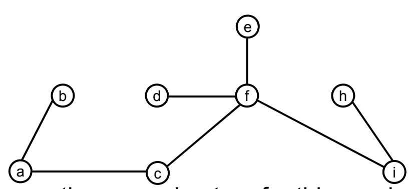
- A spanning tree is a subset of a connected graph that contains:
  - all of the vertices
  - a subset of the edges that form a tree
- The trees on the previous page were examples of spanning trees for the graph on that page. Here are two others:



the original graph



another spanning tree for this graph

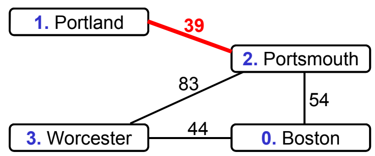


another spanning tree for this graph

Representing a Graph Using an Adjacency Matrix

- Adjacency matrix = a two-dimensional array that is used to represent the edges and any associated costs
  - edge[r][c] = the cost of going from vertex r to vertex c
- Example:

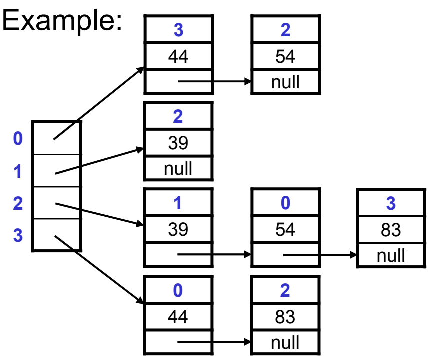
	0	1	2	3
0			54	44
1			39	
2	54	39		83
3	44		83	

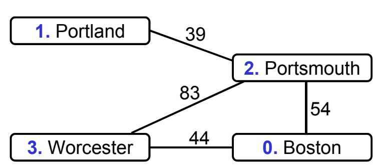


- Use a special value to indicate that you can't go from r to c.
  - either there's no edge between r and c, or it's a directed edge that goes from c to r
  - this value is shown as a shaded cell in the matrix above
  - we can't use 0, because we may have actual costs of 0
- This representation is good if a graph is *dense* – if it has many edges per vertex – but wastes memory if the graph is *sparse* – if it has few edges per vertex.

Representing a Graph Using an Adjacency List

- Adjacency list = a list (either an array or linked list) of linked lists that is used to represent the edges and any associated costs
- Example:

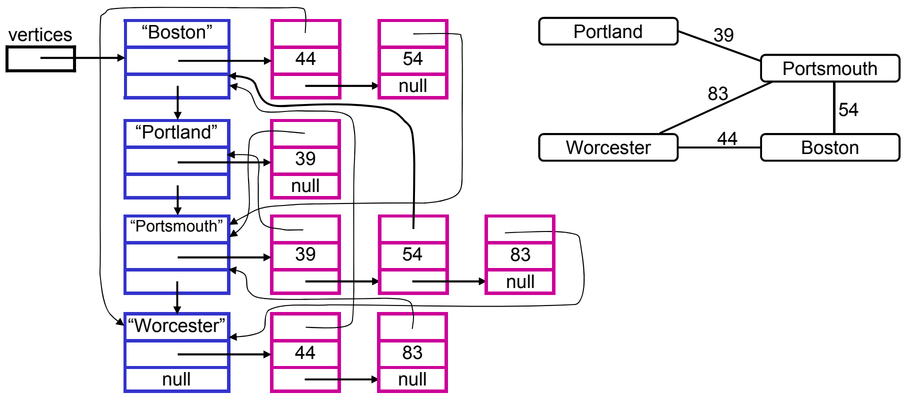




- No memory is allocated for non-existent edges, but the references in the linked lists use extra memory.
- This representation is good if a graph is sparse, but wastes memory if the graph is dense.

Our Graph Representation

- Use a linked list of linked lists for the adjacency list.
- Example:



`vertices` is a reference to a linked list of `Vertex` objects.
Each `Vertex` holds a reference to a linked list of `Edge` objects.
Each `Edge` holds a reference to the `Vertex` that is the end vertex.

Graph Class

```
public class Graph {
    private class Vertex {
        private String id;
        private Edge edges;            // adjacency list
        private Vertex next;
        private boolean encountered;
        private boolean done;
        private Vertex parent;
        private double cost;
        …
    }

    private class Edge {
        private Vertex start;
        private Vertex end;
        private double cost;
        private Edge next;
        …
    }

    private Vertex vertices;
    …
}
```

The highlighted fields are shown in the diagram on the previous page.

Traversing a Graph

- Traversing a graph involves starting at some vertex and visiting all of the vertices that can be reached from that vertex.
  - visiting a vertex = processing its data in some way
    - example: print the data
  - if the graph is connected, all of the vertices will be visited
- We will consider two types of traversals:
  - **depth-first**: proceed as far as possible along a given path before backing up
  - **breadth-first**: visit a vertex
    visit all of its neighbors
    visit all unvisited vertices 2 edges away
    visit all unvisited vertices 3 edges away, etc.
- Applications:
  - determining the vertices that can be reached from some vertex
  - state-space search
  - web crawler (vertices = pages, edges = links)

Depth-First Traversal

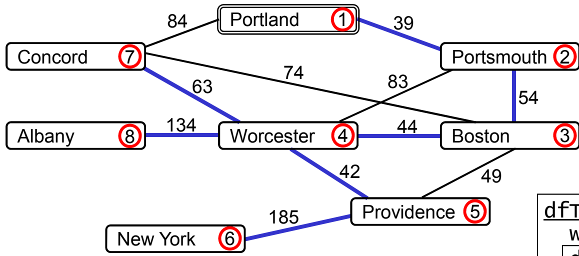
- Visit a vertex, then make recursive calls on all of its yet-to-be-visited neighbors:

```
dfTrav(v, parent)
    visit v and mark it as visited
    v.parent = parent
    for each vertex w in v's neighbors
        if (w has not been visited)
            dfTrav(w, v)
```

- Java method:

```
private static void dfTrav(Vertex v, Vertex parent) {
    System.out.println(v.id);    // visit v
    v.done = true;
    v.parent = parent;

    Edge e = v.edges;
    while (e != null) {
        Vertex w = e.end;
        if (!w.done)
            dfTrav(w, v);
        e = e.next;
    }
}
```

Example: Depth-First Traversal from Portland

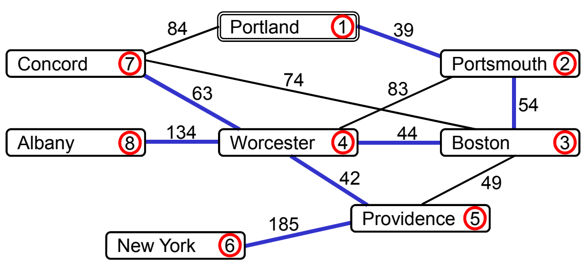


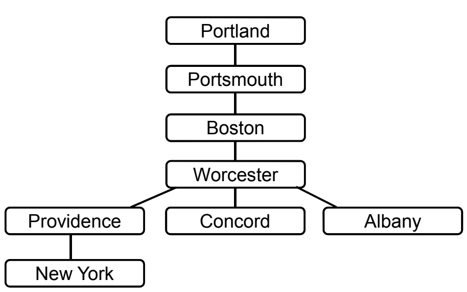
For the examples, we'll assume that the edges in each vertex's adjacency list are sorted by increasing edge cost.

```
void dfTrav(Vertex v, Vertex parent) {
    System.out.println(v.id);
    v.done = true;
    v.parent = parent;
    Edge e = v.edges;
    while (e != null) {
        Vertex w = e.end;
        if (!w.done)
            dfTrav(w, v);
        e = e.next;
    }
}
```

```
dfTrav(Ptl, null)
  w = Pts
  dfTrav(Pts, Ptl)
    w = Ptl, Bos
    dfTrav(Bos, Pts)
      w = Wor
      dfTrav(Wor, Bos)
        w = Pro
        dfTrav(Pro, Wor)
          w = Wor, Bos, NY
          dfTrav(NY, Pro)
            w = Pro
            return
          no more neighbors
          return
        w = Bos, Con
        dfTrav(Con, Wor)
          …
```

Depth-First Spanning Tree



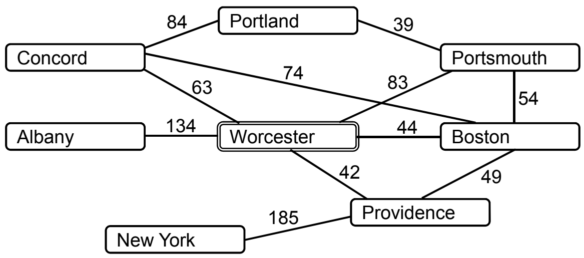


The edges obtained by following the `parent` references form a spanning tree with the origin of the traversal as its root.

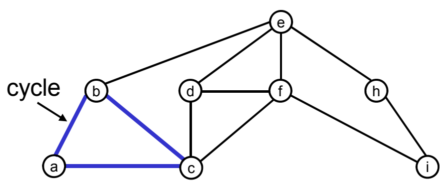
From any city, we can get to the origin by following the roads in the spanning tree.

Another Example: Depth-First Traversal from Worcester

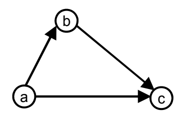
- In what order will the cities be visited?
- Which edges will be in the resulting spanning tree?



Checking for Cycles in an Undirected Graph



- To discover a cycle in an undirected graph, we can:
  - perform a depth-first traversal, marking the vertices as visited
  - when considering neighbors of a visited vertex, if we discover one already marked as visited, there must be a cycle
- If no cycles found during the traversal, the graph is acyclic.
- This doesn't work for directed graphs:
  - c is a neighbor of both a and b
  - there is no cycle

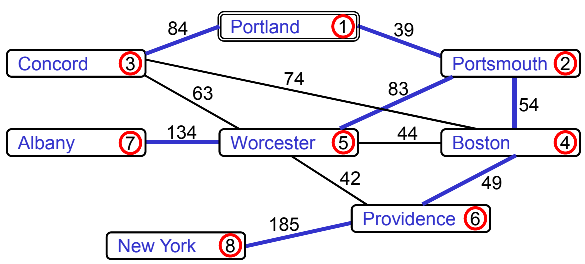


Breadth-First Traversal

- Use a queue, as we did for BFS and level-order tree traversal:

```
private static void bfTrav(Vertex origin) {
    origin.encountered = true;
    origin.parent = null;
    Queue<Vertex> q = new LLQueue<Vertex>();
    q.insert(origin);

    while (!q.isEmpty()) {
        Vertex v = q.remove();
        System.out.println(v.id);        // Visit v.

        // Add v's unencountered neighbors to the queue.
        Edge e = v.edges;
        while (e != null) {
            Vertex w = e.end;
            if (!w.encountered) {
                w.encountered = true;
                w.parent = v;
                q.insert(w);
            }
            e = e.next;
        }
    }
}
```

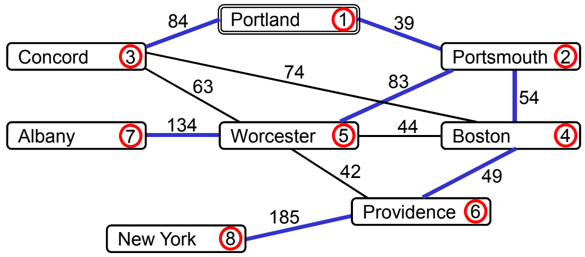
Example: Breadth-First Traversal from Portland



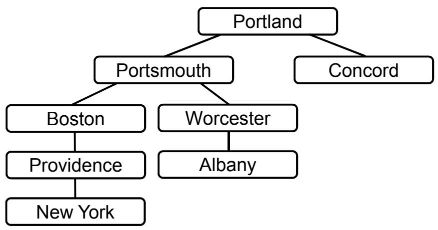
Evolution of the queue:

<u>remove</u>	<u>insert</u>	<u>queue contents</u>
	Portland	Portland
Portland	Portsmouth, Concord	Portsmouth, Concord
Portsmouth	Boston, Worcester	Concord, Boston, Worcester
Concord	*none*	Boston, Worcester
Boston	Providence	Worcester, Providence
Worcester	Albany	Providence, Albany
Providence	New York	Albany, New York
Albany	*none*	New York
New York	*none*	*empty*

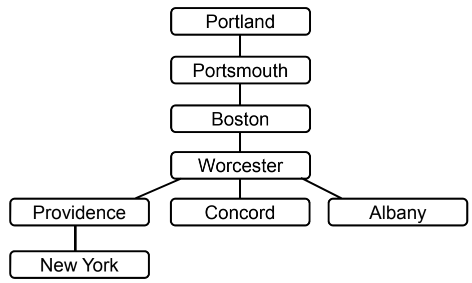
Breadth-First Spanning Tree



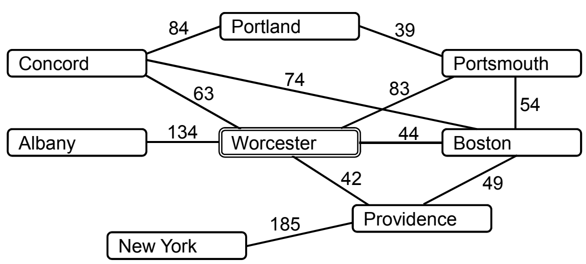
breadth-first spanning tree:



depth-first spanning tree:



Another Example: Breadth-First Traversal from Worcester



Evolution of the queue:

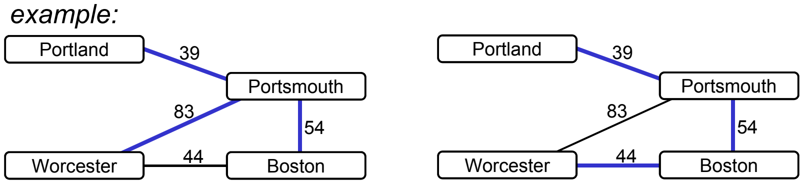
remove	insert	queue contents

Time Complexity of Graph Traversals

- let V = number of vertices in the graph
  E = number of edges
- If we use an adjacency matrix, a traversal requires $O(V^2)$ steps.
  - why?
- If we use an adjacency list, a traversal requires $O(V + E)$ steps.
  - visit each vertex once
  - traverse each vertex's adjacency list at most once
    - the total length of the adjacency lists is at most $2E = O(E)$
  - $O(V + E) << O(V^2)$ for a sparse graph
  - for a dense graph, $E = O(V^2)$, so both representations are $O(V^2)$
- In our implementations of the remaining algorithms, we'll assume an adjacency-list implementation.

Minimum Spanning Tree

- A minimum spanning tree (MST) has the smallest total cost among all possible spanning trees.
  - *example:*



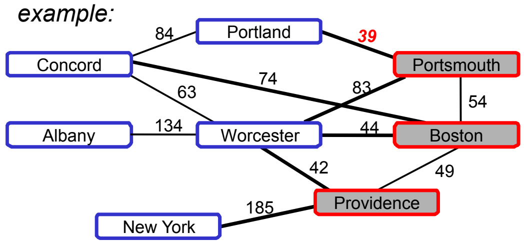
one possible spanning tree
(total cost = 39 + 83 + 54 = 176)

the minimal-cost spanning tree
(total cost = 39 + 54 + 44 = 137)

- If no two edges have the same cost, there is a unique MST. If two or more edges have the same cost, there may be more than one MST.
- Finding an MST could be used to:
  - determine the shortest highway system for a set of cities
  - calculate the smallest length of cable needed to connect a network of computers

Building a Minimum Spanning Tree

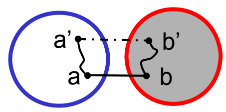
- Key insight: if you divide the vertices into two disjoint subsets A and B, then the lowest-cost edge joining a vertex in A to a vertex in B – call it (a, b) – must be part of the MST.
  - *example:*



The 6 bold edges each join an unshaded vertex to a shaded vertex.

The one with the lowest cost (Portland to Portsmouth) must be in the MST.

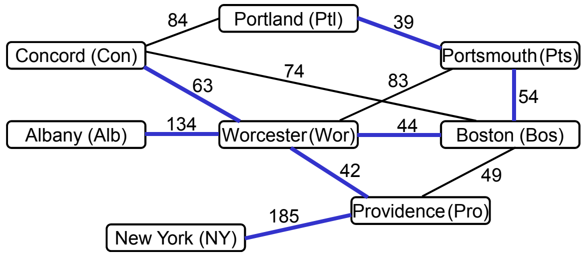
- Proof by contradiction:
  - assume there is an MST (call it T) that doesn't include (a, b)
  - T must include a path from a to b, so it must include one of the other edges (a', b') that spans subsets A and B, such that (a', b') is part of the path from a to b
  - adding (a, b) to T introduces a cycle
  - removing (a', b') gives a spanning tree with lower cost, which contradicts the original assumption.



Prim's MST Algorithm

- Begin with the following subsets:
  - A = any one of the vertices
  - B = all of the other vertices
- Repeatedly select the lowest-cost edge (a, b) connecting a vertex in A to a vertex in B and do the following:
  - add (a, b) to the spanning tree
  - update the two sets: A = A U {b}
B = B – {b}
- Continue until A contains all of the vertices.

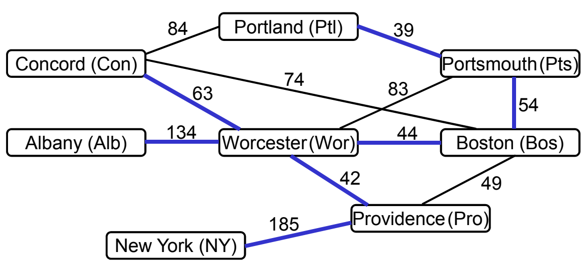
Example: Prim's Starting from Concord



- Tracing the algorithm:

edge added	set A	set B
	{Con}	{Alb, Bos, NY, Ptl, Pts, Pro, Wor}
(Con, Wor)	{Con, Wor}	{Alb, Bos, NY, Ptl, Pts, Pro}
(Wor, Pro)	{Con, Wor, Pro}	{Alb, Bos, NY, Ptl, Pts}
(Wor, Bos)	{Con, Wor, Pro, Bos}	{Alb, NY, Ptl, Pts}
(Bos, Pts)	{Con, Wor, Pro, Bos, Pts}	{Alb, NY, Ptl}
(Pts, Ptl)	{Con, Wor, Pro, Bos, Pts, Ptl}	{Alb, NY}
(Wor, Alb)	{Con, Wor, Pro, Bos, Pts, Ptl, Alb}	{NY}
(Pro, NY)	{Con, Wor, Pro, Bos, Pts, Ptl, Alb, NY}	{}

MST May Not Give Shortest Paths



- The MST is the spanning tree with the minimal *total* edge cost.
- It does not necessarily include the minimal cost path between a pair of vertices.
- Example: shortest path from Boston to Providence is along the single edge connecting them
  - that edge is not in the MST

Implementing Prim's Algorithm in our `Graph` class

- Use the `done` field to keep track of the sets.
  - if `v.done == true`, v is in set A
  - if `v.done == false`, v is in set B

- Repeatedly scan through the lists of vertices and edges to find the next edge to add.
  ➔ *O*(EV)

- We can do better!
  - use a heap-based priority queue to store the vertices in set B
  - priority of a vertex x = –1 * cost of the lowest-cost edge connecting x to a vertex in set A
    - why multiply by –1?

  - somewhat tricky: need to update the priorities over time
  ➔ *O*(E log V)

The Shortest-Path Problem

- It's often useful to know the shortest path from one vertex to another – i.e., the one with the minimal total cost
  - example application: routing traffic in the Internet

- For an *unweighted* graph, we can simply do the following:
  - start a breadth-first traversal from the origin, v
  - stop the traversal when you reach the other vertex, w
  - the path from v to w in the resulting (possibly partial) spanning tree is a shortest path

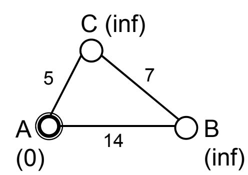
- A breadth-first traversal works for an unweighted graph because:
  - the shortest path is simply one with the fewest edges
  - a breadth-first traversal visits cities in order according to the number of edges they are from the origin.

- Why might this approach fail to work for a *weighted* graph?

Dijkstra's Algorithm

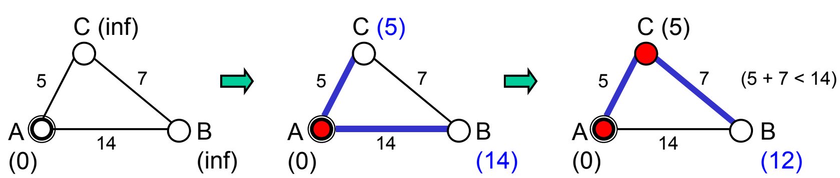
- One algorithm for solving the shortest-path problem for weighted graphs was developed by E.W. Dijkstra.
- It allows us to find the shortest path from a vertex v (the origin) to *all other vertices* that can be reached from v.
- Basic idea:
  - maintain estimates of the shortest paths from the origin to every vertex (along with their costs)
  - gradually refine these estimates as we traverse the graph
- Initial estimates:

	path	cost
the origin itself:	stay put!	0
all other vertices:	unknown	infinity



Dijkstra's Algorithm (cont.)

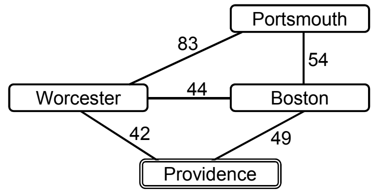
- We say that a vertex w is *finalized* if we have found the shortest path from v to w.
- We repeatedly do the following:
  - find the unfinalized vertex w with the lowest cost estimate
  - mark w as finalized (shown as a filled circle below)
  - examine each unfinalized neighbor x of w to see if there is a shorter path to x that passes through w
    - if there is, update the shortest-path estimate for x
- Example:



Another Example: Shortest Paths from Providence

- Initial estimates:

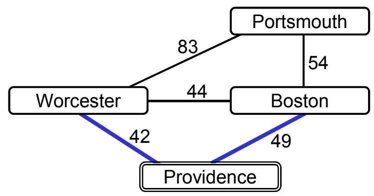
Boston	infinity
Worcester	infinity
Portsmouth	infinity
Providence	***0***



- Providence has the smallest unfinalized estimate, so we finalize it.

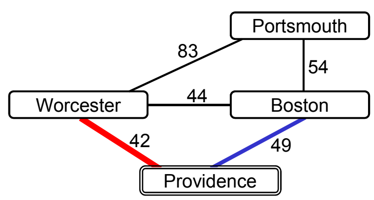
- We update our estimates for its neighbors:

Boston	*49* (< infinity)
Worcester	*42* (< infinity)
Portsmouth	infinity
Providence	0



Shortest Paths from Providence (cont.)

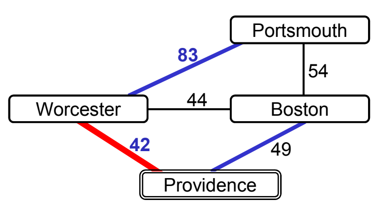
Boston	49
Worcester	***42***
Portsmouth	infinity
Providence	0



- Worcester has the smallest unfinalized estimate, so we finalize it.
  - any other route from Prov. to Worc. would need to go via Boston, and since (Prov → Worc) < (Prov → Bos), we can't do better.

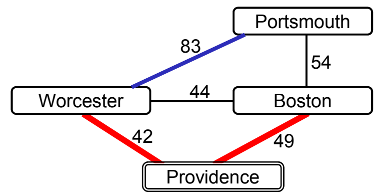
- We update our estimates for Worcester's unfinalized neighbors:

Boston	49 (no change)
Worcester	42
Portsmouth	*125* (42 + 83 < infinity)
Providence	0



Shortest Paths from Providence (cont.)

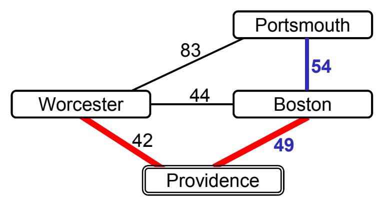
Boston	***49***
Worcester	42
Portsmouth	125
Providence	0



- Boston has the smallest unfinalized estimate, so we finalize it.
  - we'll see later why we can safely do this!

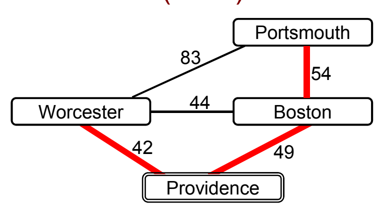
- We update our estimates for Boston's unfinalized neighbors:

Boston	49
Worcester	42
Portsmouth	*103* (49 + 54 < 125)
Providence	0



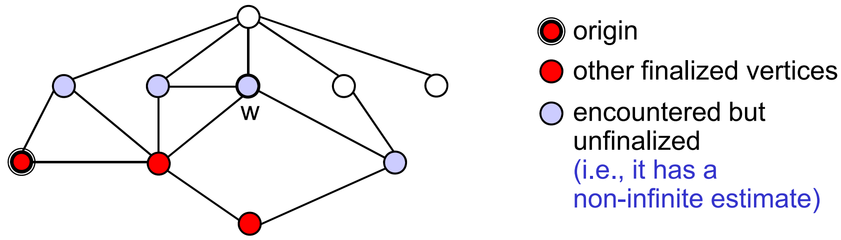
Shortest Paths from Providence (cont.)

Boston	49
Worcester	42
Portsmouth	***103***
Providence	0



- Only Portsmouth is left, so we finalize it.

Finalizing a Vertex



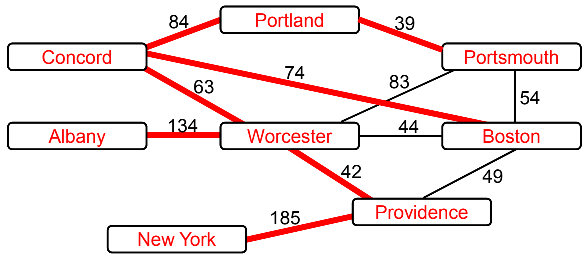
- Let w be the unfinalized vertex with the smallest cost estimate. Why can we finalize w, before seeing the rest of the graph?
- We know that w's current estimate is for the shortest path to w that passes through only *finalized* vertices.
- Any shorter path to w would have to pass through one of the other encountered-but-unfinalized vertices, but we know that they're all further away from the origin than w is.
  - their cost estimates may decrease in subsequent stages of the algorithm, but they can't drop below w's current estimate!

Pseudocode for Dijkstra's Algorithm

```
dijkstra(origin)
   origin.cost = 0
   for each other vertex v
        v.cost = infinity;

   while there are still unfinalized vertices with cost < infinity
        find the unfinalized vertex w with the minimal cost
        mark w as finalized

        for each unfinalized vertex x adjacent to w
            cost_via_w = w.cost + edge_cost(w, x)
            if (cost_via_w < x.cost)
                x.cost = cost_via_w
                x.parent = w
```

- At the conclusion of the algorithm, for each vertex v:
  - v.cost is the cost of the shortest path from the origin to v; if v.cost is infinity, there is no path from the origin to v
  - starting at v and following the parent references yields the shortest path

Example: Shortest Paths from Concord

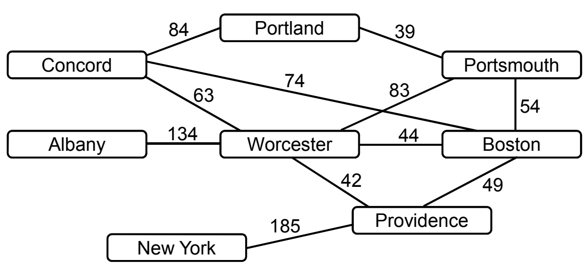


Evolution of the cost estimates (costs in bold have been finalized):

Albany	inf	inf	197	197	197	197	***197***	
Boston	inf	74	***74***					
Concord	***0***							
New York	inf	inf	inf	inf	inf	290	290	***290***
Portland	inf	84	84	***84***				
Portsmouth	inf	inf	146	128	123	***123***		
Providence	inf	inf	105	105	***105***			
Worcester	inf	***63***						

Note that the Portsmouth estimate was improved three times!

Another Example: Shortest Paths from Worcester



Evolution of the cost estimates (costs in bold have been finalized):

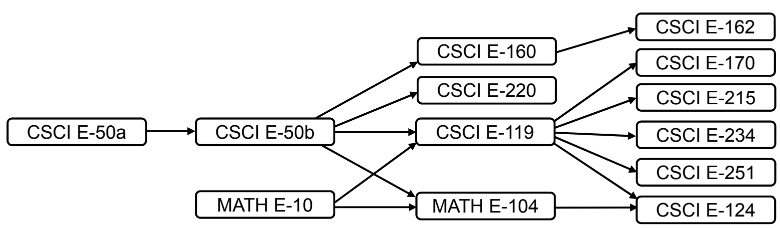
Albany								
Boston								
Concord								
New York								
Portland								
Portsmouth								
Providence								
Worcester								

Implementing Dijkstra's Algorithm

- Similar to the implementation of Prim's algorithm.
- Use a heap-based priority queue to store the unfinalized vertices.
  - priority = ?
- Need to update a vertex's priority whenever we update its shortest-path estimate.
- Time complexity = $O(E\log V)$

Topological Sort

- Used to order the vertices in a directed acyclic graph (a DAG).
- Topological order: an ordering of the vertices such that, if there is directed edge from a to b, a comes before b.
- Example application: ordering courses according to prerequisites



  - a directed edge from a to b indicates that a is a prereq of b
- There may be more than one topological ordering.

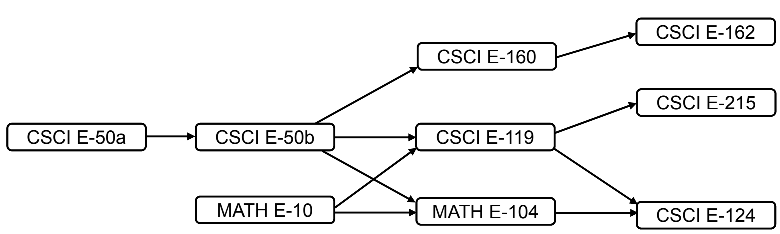
Topological Sort Algorithm

- A *successor* of a vertex v in a directed graph = a vertex w such that (v, w) is an edge in the graph (v → w)
- Basic idea: find vertices that have no successors and work backward from them.
  - there must be at least one such vertex. why?
- Pseudocode for one possible approach:

```
topolSort
    S = a stack to hold the vertices as they are visited
    while there are still unvisited vertices
        find a vertex v with no unvisited successors
        mark v as visited
        S.push(v)
    return S
```

- Popping the vertices off the resulting stack gives one possible topological ordering.

Topological Sort Example

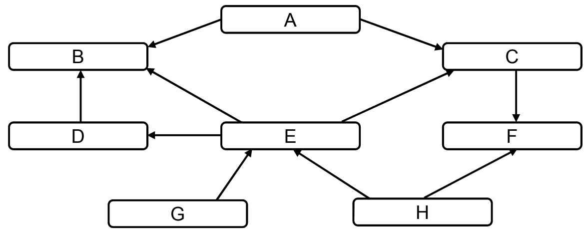


Evolution of the stack:

push	stack contents (top to bottom)
E-124	E-124
E-162	E-162, E-124
E-215	E-215, E-162, E-124
E-104	E-104, E-215, E-162, E-124
E-119	E-119, E-104, E-215, E-162, E-124
E-160	E-160, E-119, E-104, E-215, E-162, E-124
E-10	E-10, E-160, E-119, E-104, E-215, E-162, E-124
E-50b	E-50b, E-10, E-160, E-119, E-104, E-215, E-162, E-124
E-50a	**E-50a, E-50b, E-10, E-160, E-119, E-104, E-215, E-162, E-124**

one possible topological ordering

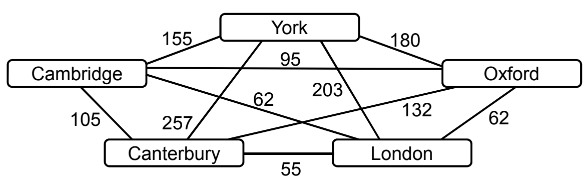
Another Topological Sort Example



Evolution of the stack:

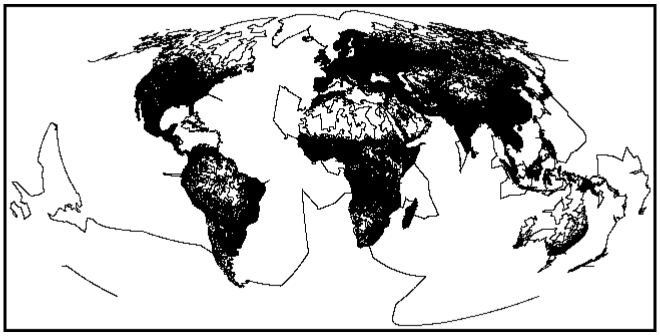
<u>push</u> <u>stack contents (top to bottom)</u>

Traveling Salesperson Problem (TSP)



- A salesperson needs to travel to a number of cities to visit clients, and wants to do so as efficiently as possible.
- As in our earlier problems, we use a weighted graph.
- A *tour* is a path that begins at some starting vertex, passes through every other vertex *once and only once*, and returns to the starting vertex. (The actual starting vertex doesn't matter.)
- TSP: find the tour with the lowest total cost
- TSP algorithms assume the graph is complete, but we can assign infinite costs if there isn't a direct route between two cities.

TSP for Santa Claus



source: http://www.tsp.gatech.edu/world/pictures.html

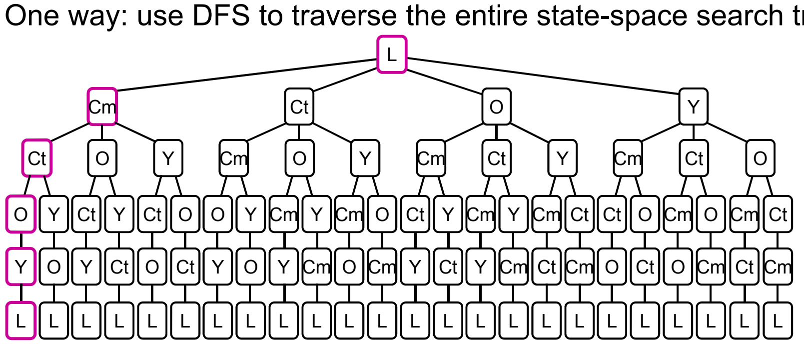
A "world TSP" with 1,904,711 cities.

The figure at right shows a tour with a total cost of 7,516,353,779 meters – which is at most 0.068% longer than the optimal tour.

- Other applications:
  - coin collection from phone booths
  - routes for school buses or garbage trucks
  - minimizing the movements of machines in automated manufacturing processes
  - many others

Solving a TSP: Brute-Force Approach

- Perform an exhaustive search of all possible tours.
- One way: use DFS to traverse the entire state-space search tree.



- The leaf nodes correspond to possible solutions.
  - for n cities, there are (n – 1)! leaf nodes in the tree.
  - half are redundant (e.g., L-Cm-Ct-O-Y-L = L-Y-O-Ct-Cm-L)
- Problem: exhaustive search is intractable for all but small n.
  - example: when n = 14, ((n – 1)!) / 2 = over 3 billion

Solving a TSP: Informed State-Space Search

- Use A* with an appropriate heuristic function for estimating the cost of the remaining edges in the tour.
- This is much better than brute force, but it still uses exponential space and time.

Algorithm Analysis Revisited

- Recall that we can group algorithms into classes (n = problem size):

name	example expressions	big-O notation
constant time	1, 7, 10	$O(1)$
logarithmic time	$3\log_{10}n$, $\log_2 n + 5$	$O(\log n)$
linear time	$5n$, $10n - 2\log_2 n$	$O(n)$
n log n time	$4n \log_2 n$, $n \log_2 n + n$	$O(n \log n)$
quadratic time	$2n^2 + 3n$, $n^2 - 1$	$O(n^2)$
n^c ($c > 2$)	$n^3 - 5n$, $2n^5 + 5n^2$	$O(n^c)$
exponential time	2^n, $5e^n + 2n^2$	$O(c^n)$
factorial time	$(n - 1)!/2$, $3n!$	$O(n!)$

- Algorithms that fall into one of the classes above the dotted line are referred to as *polynomial-time* algorithms.
- The term *exponential-time algorithm* is sometimes used to include *all* algorithms that fall below the dotted line.
  - algorithms whose running time grows as fast or faster than c^n

Classifying Problems

- Problems that can be solved using a polynomial-time algorithm are considered “easy” problems.
  - we can solve large problem instances in a reasonable amount of time
- Problems that don’t have a polynomial-time solution algorithm are considered “hard” or "intractable" problems.
  - they can only be solved exactly for small values of n
- Increasing the CPU speed doesn't help much for intractable problems:

	CPU 1	CPU 2 (1000x faster)
max problem size for $O(n)$ alg:	N	1000N
$O(n^2)$ alg:	N	31.6 N
$O(2^n)$ alg:	N	**N + 9.97**

Classifying Problems (cont.)

- The class of problems that can be solved using a polynomial-time algorithm is known as the class P.
- Many problems that don’t have a polynomial-time solution algorithm belong to a class known as NP
  - for *non-deterministic polymonial*
- If a problem is in NP, it’s possible to guess a solution and verify if the guess is correct in polynomial time.
  - example: a variant of the TSP in which we attempt to determine if there is a tour with total cost <= some bound b
  - given a tour, it takes polynomial time to add up the costs of the edges and compare the result to b

Classifying Problems (cont.)

- If a problem is *NP-complete*, then finding a polynomial-time solution for it would allow you to solve a large number of other hard problems.
  - thus, it's extremely unlikely such a solution exists!
- The TSP variant described on the previous slide is NP-complete.
- Finding the optimal tour is at least as hard.
- For more info. about problem classes, there is a good video of a lecture by Prof. Michael Sipser of MIT available here:

  http://claymath.msri.org/sipser2006.mov

Dealing With Intractable Problems

- When faced with an intractable problem, we resort to techniques that quickly find solutions that are "good enough".
- Such techniques are often referred to as *heuristic* techniques.
  - heuristic = rule of thumb
  - there's no guarantee these techniques will produce the optimal solution, but they typically work well

Iterative Improvement Algorithms

- One type of heuristic technique is what is known as an *iterative improvement algorithm.*
  - start with a randomly chosen solution
  - gradually make small changes to the solution in an attempt to improve it
    - e.g., change the position of one label
  - stop after some number of iterations
- There are several variations of this type of algorithm.

Hill Climbing

- Hill climbing is one type of iterative improvement algorithm.
  - start with a randomly chosen solution
  - repeatedly consider possible small changes to the solution
  - if a change would improve the solution, make it
  - if a change would make the solution worse, don't make it
  - stop after some number of iterations
- It's called hill climbing because it repeatedly takes small steps that improve the quality of the solution.
  - "climbs" towards the optimal solution

Simulated Annealing

- *Simulated annealing* is another iterative improvement algorithm.
  - start with a randomly chosen solution
  - repeatedly consider possible small changes to the solution
  - if a change would improve the solution, make it
  - if a change would make the solution worse, *make it some of the time* (according to some probability)
  - the probability of doing so reduces over time

Take-Home Lessons

- Object-oriented programming allows us to capture the abstractions in the programs that we write.
  - creates reusable building blocks
  - key concepts: encapsulation, inheritance, polymorphism
- Abstract data types allow us to organize and manipulate collections of data.
  - a given ADT can be implemented in different ways
  - fundamental building blocks: arrays, linked nodes
- Efficiency matters when dealing with large collections of data.
  - some solutions can be *much* faster or more space efficient than others!
  - what's the best data structure/algorithm for the specific instances of the problem that you expect to see?
    - example: sorting an almost sorted collection

Take-Home Lessons (cont.)

- Use the tools in your toolbox!
  - generic data structures
  - lists/stacks/queues
  - trees
  - heaps
  - hash tables
  - recursion
  - recursive backtracking
  - divide-and-conquer
  - state-space search
  - ...

From the Introductory Lecture

- We will study fundamental *data structures.*
  - ways of imposing order on a collection of information
  - sequences: lists, stacks, and queues
  - trees
  - hash tables
  - graphs
- We will also:
  - study *algorithms* related to these data structures
  - learn how to *compare* data structures & algorithms
- Goals:
  - learn to think more intelligently about programming problems
  - acquire a set of useful tools and techniques