What are Data Structures?

Data structures are ways to organize data (information). Examples:

- simple variables — primitive types
- objects — collection of data items of various types
- arrays — collection of data items of the same type, stored contiguously
- linked lists — sequence of data items, each one points to the next one

Typically, **algorithms** go with the data structures to manipulate the data (e.g., the methods of a class).

This course will cover some more complicated data structures:

- how to implement them efficiently
- what they are good for

Abstract Data Types

An **abstract data type** (ADT) defines

- a state of an object and
- operations that act on the object, possibly changing the state.

Similar to a Java class.

This course will cover

- specifications of several common ADTs
- pros and cons of different implementations of the ADTs (e.g., array or linked list? sorted or unsorted?)
- how the ADT can be used to solve other problems

Specific ADTs

The ADTs to be studied (and some sample applications) are:

- stack

  evaluate arithmetic expressions

- queue

  simulate complex systems, such as traffic

- general list

  AI systems, including the LISP language

- tree

  simple and fast sorting

- table

  database applications, with quick look-up

How Does C Fit In?

Although data structures are universal (can be implemented in any programming language), this course will use Java and C:

- non-object-oriented parts of Java are based on C
- C is *not* object-oriented

We will learn how to gain the advantages of data abstraction and modularity in C, by using self-discipline to achieve what Java forces you to do.

Reasons to learn C:

- learn proficiency with pointers and garbage collection
- useful in later courses and the real world
- ubiquitous and often free C software
- Unix is written in C
- C code can be very concise
- very efficient compilers, so resulting code can be very fast

Other Topics

Course will emphasize **good software development practice**:

- requirements analysis
- focus on design
- good documentation and self-documenting code
- testing

Course will touch on several more **advanced computer science topics** that appear later in the curriculum, and fit in with our topics this semester:

- file systems and databases
- artificial intelligence
- computability and complexity

Principles of Computer Science

Computer Science is like:

- engineering: build artifacts to work well
- science: search for fundamental laws and principles
- math: use formalisms to express fundamental laws.

However, CS studies artificial phenomena, computers and programs.

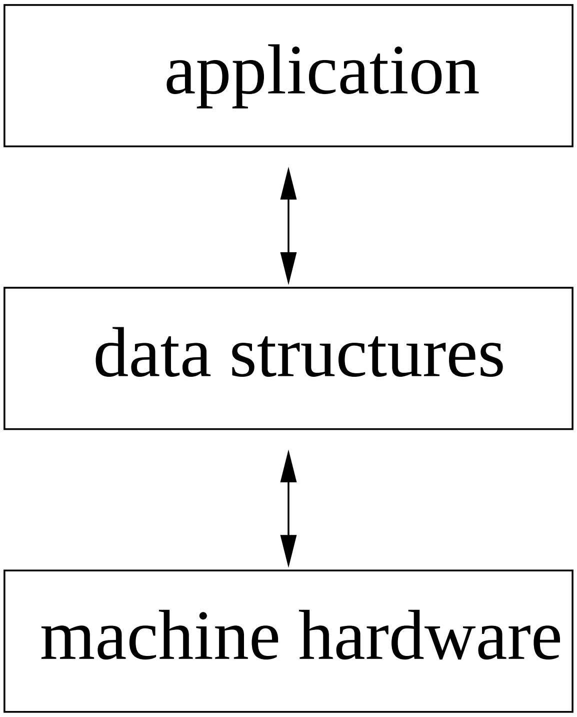
Recurring concepts in computer science are:

- layers, hierarchies, information-hiding, abstraction, interfaces (organizational principles to aid in the construction of large systems)
- efficiency, tradeoffs, resource usage (properties of algorithms and problems)
- reliability, affordability, correctness (acknowledging imperfectness and economic considerations)

Introduction to Data Structures

Data structures are one of the enduring principles in computer science. Why?

1. *Data structures are based on the notion of information hiding:*



   Changes in hardware require changes in data structure implementation but *not in the application*.

2. *A number of data structures are useful in a wide range of applications*.

   Promotes code reuse.

Efficiency Considerations

Since these data structures are so widespread, it's important to implement them efficiently. Measures of efficiency:

- running time
- space

in

- worst case
- average case

We will study tradeoffs, such as

- time vs. space
- the speed of one operation vs. the speed of another

Efficiency will be measured using

- asymptotic analysis and
- big-oh notation

Asymptotic Analysis

Actual (wall-clock) time of a program is affected by:

- size of the input
- programming language
- programming tricks
- compiler
- CPU speed
- multiprogramming level (other users)

Instead of wall-clock time, look at the *pattern* of the program's behavior *as the problem size increases*. This is called **asymptotic analysis**.

That is, look at the shape of the function $f(n)$ that gives the running time on inputs of size n, with more emphasis on what happens as n gets big.

Big-Oh Notation

Big-oh notation is used to capture the generic *shape of the curve.*

From a practical point of view, you can get the big-oh notation for a function by

1. ignoring multiplicative constants (these are due to pesky differences in compiler, CPU, etc.) and
2. discarding the lower order terms (as n gets larger, the largest term has the biggest impact)

Which terms are lower order than others? In increasing order: constant, $\log n$, n, $n \cdot \log n$, n^2, n^3, $\ldots$, 2^n.

Examples:

- $4302 = O(1)$
- $n^3 + n \log n + n^5 + n = O(n^5)$
- $34n^3 - 2n \log n + .0004n^5 + 5.2n = O(n^5)$.

See Appendix B, Section 4 of Standish, or CPSC 311, for mathematical definitions and justifications.

Why Multiplicative Constants are Unimportant

An example showing how multiplicative constants become unimportant as n gets very large:

n	$1000 \log n$	$.0001 \cdot n^2$
2	1000	.0004
256	8000	6.5
4096	12,000	1677.7
8192	13,000	6710.9
16,384	14,000	26,843.6
32,768	15,000	107,374.2
1,048,576	20,000	109,951,162.8

Big-oh notation is not always appropriate! If your program is working on small input sizes, the better algorithm may be one that has a worse big-oh analysis. Notice in the table above that below $n = 4096$, the $O(\log n)$ function is LARGER than the $O(n^2)$ function.

Generic Steps

How can you figure out the running time of an algorithm without implementing it, running it on various inputs, plotting the results, and fitting a curve to the data? And even if you did that, how would you know you fit the right curve?

We count **generic steps** of the algorithm. Each generic step that we count should be *an operation that can be performed in constant time in an actual implementation on a real computer.*

Classifying an assignment statement as a generic step is reasonable.

Classifying a statement “sort the entire array” as a generic step is unreasonable, since the time to sort the array will depend on the size of the array and will not be constant.

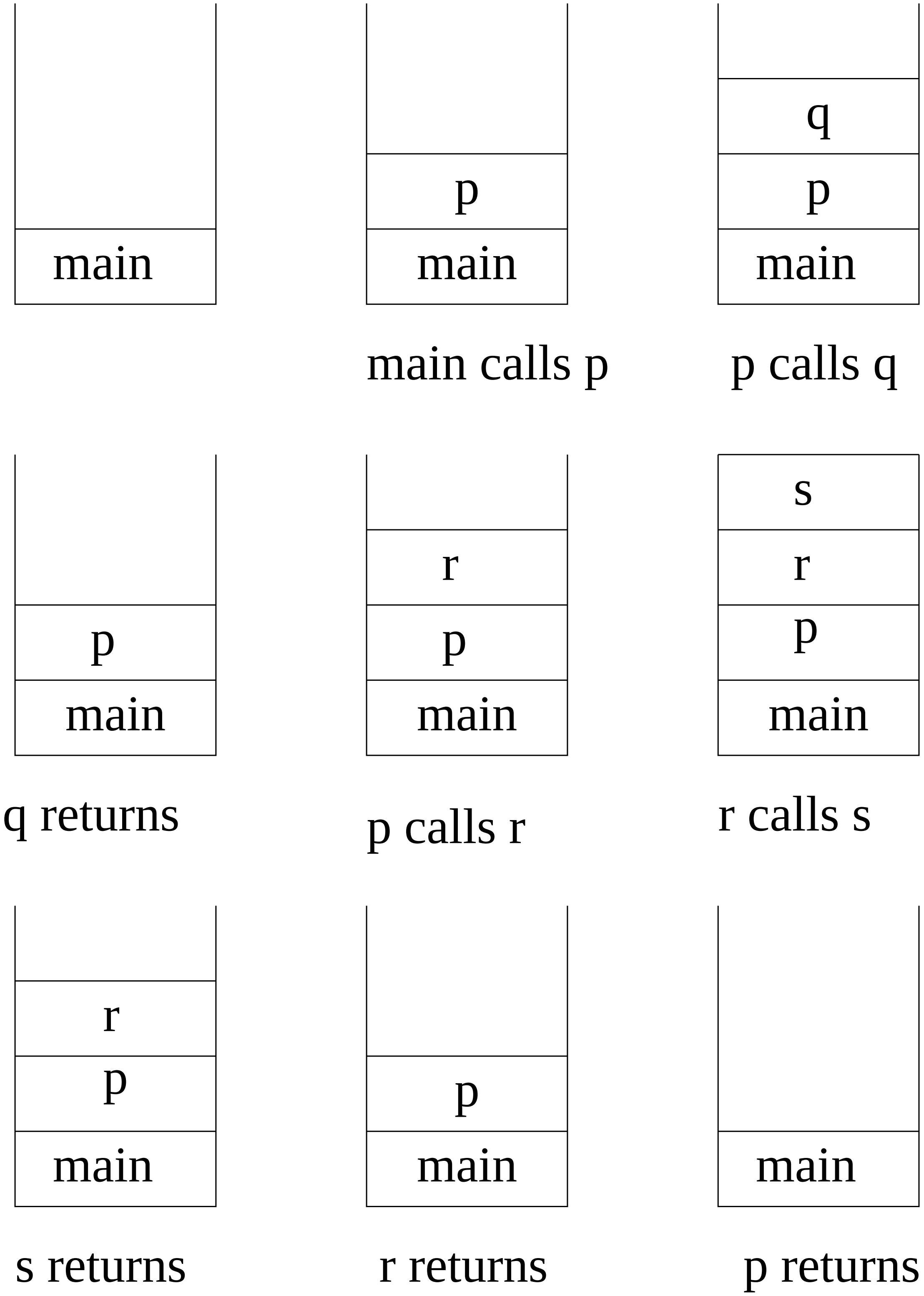
Stack vs. Heap

Memory used by an executing program is partitioned:

- **the stack:**
  - When a method begins executing, a piece of the stack (**stack frame**) is devoted to it.
  - There is an entry in the stack frame for
    - ∗ each formal parameter
    - ∗ the return value
    - ∗ every variable declared in the method
  - For variables of primitive type, the data itself is stored on the stack

    For variables of object type, only a pointer to the data is stored on the stack.
  - When the method finishes, the method's stack frame is discarded: its formal parameters and local variables are no longer accessible.
- **the heap:** Dynamically allocated memory goes here, including the actual data for objects. Lifetime is independent of method lifetimes.

Stack Frames Example



main

q
p
main

main calls p

q
p
main

p calls q

p
main

q returns

r
p
main

p calls r

s
r
p
main

r calls s

r
p
main

s returns

p
main

r returns

main

p returns

Objects

An **object** is an entity (e.g., a ball) that has

- **state** — variables
- **behavior** — methods

A **class** is the model, or pattern, from which objects are created.

Analogy: a class is like an architectural blueprint
for a house;
an object is like an actual house.

- class defines important characteristics of the object
- construction is required to translate class into object
- many objects/houses can be created from the same class

Data Abstraction

The class concept supports **data abstraction**.

Similar principles apply as for procedural abstraction:

- group data that belongs together (Java's version of a record or struct)
- group data together with accompanying behavior
- separate the issue of how the data is implemented from the issue of how the data is used
- separate the issue of how the behavior is implemented from the issue of how the behavior is used.

References

The class of an object is its data type.

Objects are declared differently than are variables of primitive types.

Suppose there is a class called `Person`.

```
int total;
Person neighbor;
```

- Declaration of `total` allocates storage on the stack to hold an `int` and associates the name `total` with the address of that space.
- Declaration of `neighbor` allocates storage on the stack to hold a **reference** (or pointer) to an object of type `Person`, *but does not allocate any space for the Person object itself*.

Creating Objects

A **constructor** is a special method of the class that actually creates an object.

When a constructor is called,

- storage space is allocated *on the heap* for the object
- *each object gets its own space (own copy of the instance variables)*
- the object's state is initialized according to the (user-defined) code for that class

The name of the constructor for class `X` is `X()`. Ex:

```
neighbor = new Person();
```

The operator `new` *must be put in front of the call to the constructor.*

Summary: Declaring a variable of an object type produces a reference to the object, but not the object itself. To get the object itself, use `new` and the constructor for the class.

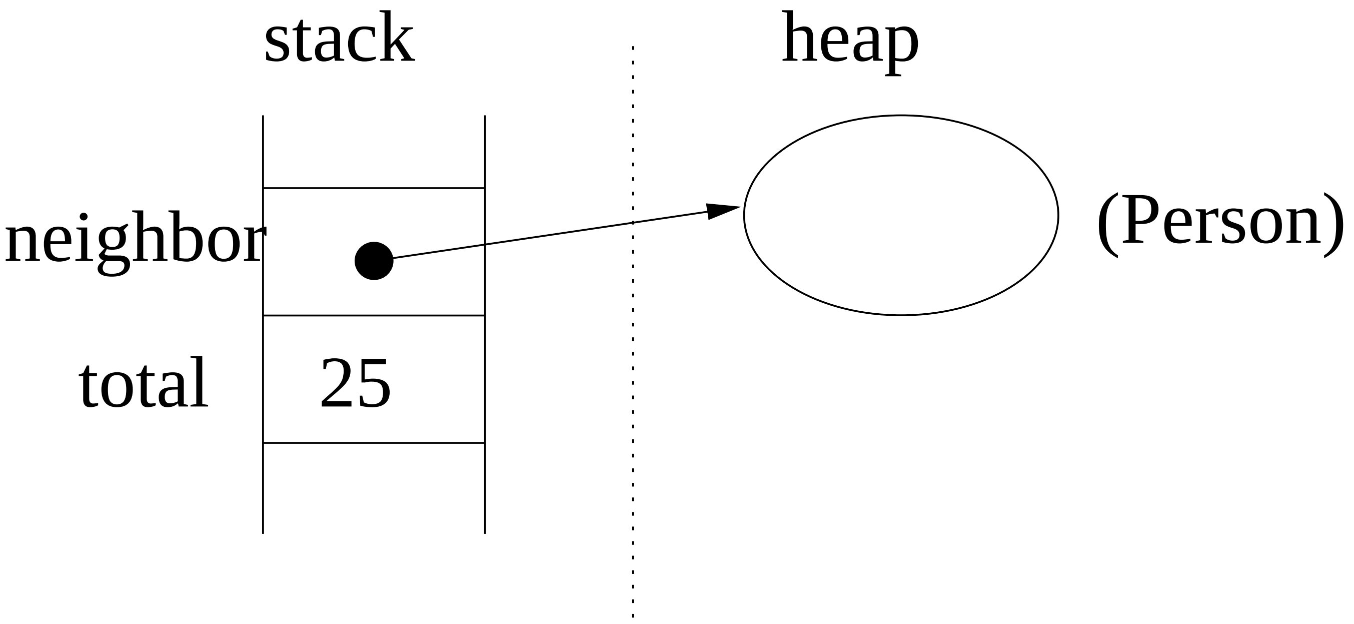
Creating Objects (cont’d)

You can combine the declaration and initialization:

```
Person neighbor = new Person();
```

just as you can for primitive types:

```
int total = 25;
```



Object Assignment & Aliases

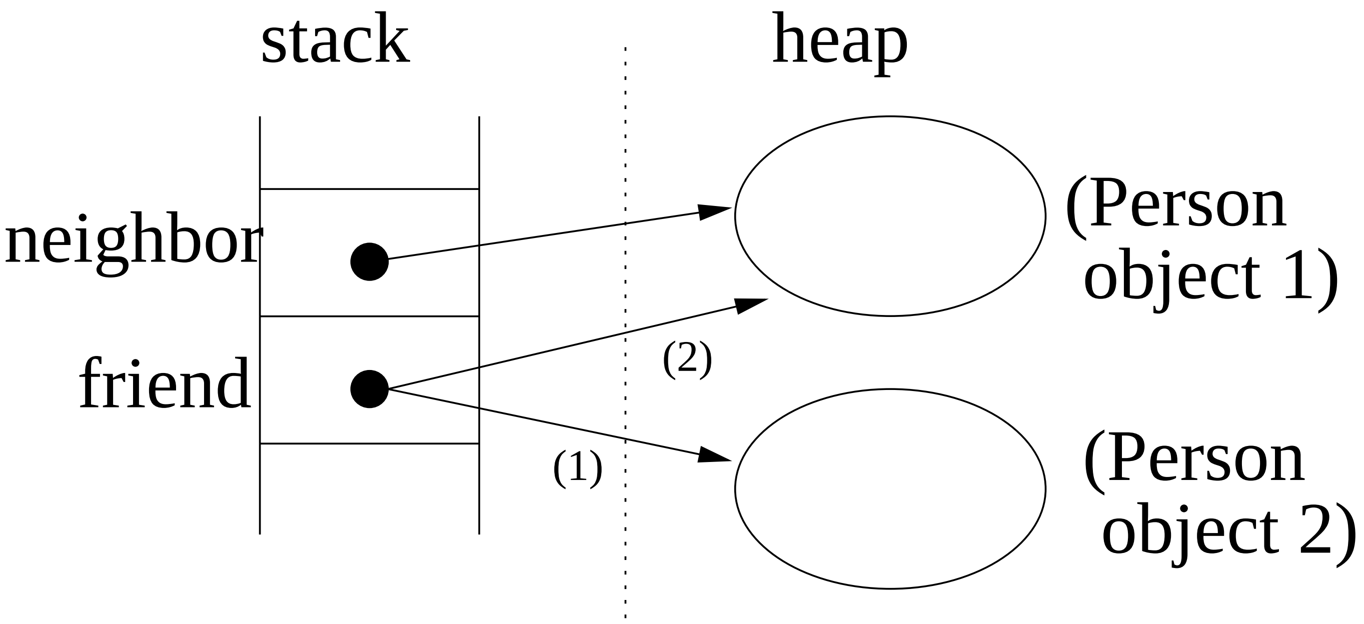
The meaning of assignment is *different* for objects than it is for primitive types.

```
int num1 = 5;
int num2 = 12;
num2 = num1;
```

At the end, `num2` holds 5.

```
Person neighbor = new Person(); // creates object 1
Person friend = new Person();   // creates object 2
friend = neighbor;
```

At the end, `friend` and `neighbor` both refer to object 1 (they are **aliases** of each other) and nothing refers to object 2 (it is **inaccessible**).



Java will **automatically garbage collect** object 2.

Data Abstraction Revisited

As a rule of thumb, referring to instance variables outside the class is NOT a good idea: it breaks the abstraction by exposing the internal workings of the class.

For instance, the implementor of the `Person` class might decide to store the age in months, instead of years.

In this case, `getAgeInYears` must change:

```
int getAgeInYears() {
   return age/12;
}
```

Code that got the age using this method need not change, but code that got the age using `.age` directly must be modified.

Moral: Separate specification (*what* a class does) from implementation (*how* a class does it)!

Public vs. Private

You can tailor the ability to access methods and variables from outside the class, using **visibility modifiers**.

- **public:** the variable or method can be referenced by any method
- **private:** the variable or method can only be referenced by methods belonging to the class

Visibility modifiers go at the beginning of the line that declares the variable or method. Ex:

```
public static void main(...
private int age;
```

Rules of thumb:

- make instance variables private
- make instance methods that are part of the public interface of the class public
- make instance methods that help with internal work of a class private

Public vs. Private (cont’d)

Instance variables should be accessible only *indirectly* via public ”get” and ”set” methods. Ex:

```
getAgeInYears()
```

Visibility modifiers are a very powerful feature which **enforces** *data encapsulation with the compiler, and thus helps with modularity and abstraction.*

Group together all the private variables/methods, and all the public ones when you format your program.

Specification vs. Implementation

Users of a class should rely only on the specification of the class. They are allowed to

- declare variables of the class type
- create instances of the class using constructors
- invoke public methods

Implementors of a class should

- define public interface for the class (names of the methods, their return types, and their parameter lists, and what do they do, but not how they do it)
- hide all details of the implementation from users
- protect internal data from access by users
- feel free to change implementation at any time, as long as public interface is unchanged

Inheritance

Inheritance lets a programmer derive a new class from an existing class. New class can

- use the variables and methods of the existing class
- modify the variables and methods of the existing class (called **overriding**)
- have additional variables and methods

Thus inheritance promotes *software reuse*. It is a defining characteristic of object-oriented programming.

Terminology:

- Class A is **derived** from (or, **inherits** from) another class B
- A is called **subclass** or **child class**.
- B is called **superclass** or **parent class**.

Benefits of Inheritance

Inheritance is particularly useful in *large* software projects:

- software reusability across programs:
  - saves time for development and maintenance
  - provides more reliable code
  - supports rapid prototyping
- code sharing within a program (like procedural abstraction); write code only once and decrease size of program
- consistent interfaces can be enforced — every class that inherits from the same parent class must conform to that interface
- information hiding / modularity

Costs of Inheritance

- Execution speed is decreased, to handle the generality of dealing with arbitrary subclasses.
  - Usually this disadvantage is outweighed by increased development speed.
  - Once system is working, then find bottlenecks and work to reduce them.
- Program size is larger if you use large libraries — not a problem these days for most applications.
- Program complexity can increase – understanding control flow requires studying the inheritance graph.

Inheritance in Java

To declare that a class is a subclass of another class:

```
class <child-class> extends <parent-class> {
   ... // define the child-class
}
```

- child class inherits all public variables of the parent class
- child class inherits all public methods of the parent class, except constructors
- child class does NOT inherit any private variables of the parent class
- child class does NOT inherit any private methods of the parent class

Inherited variables and methods can be used in the child class as if they had been declared in the child class.

Inheritance is one-way street!! An object of the parent class **cannot** access variables or methods of the child class.

Protected Visibility

- `private`: entity cannot be inherited but is not visible outside the class

- `public`: entity can be inherited but is visible outside the class

This makes it dangerous to inherit variables, since normally instance variables should not be made accessible outside the class.

The solution is an intermediate level of visibility:

- `protected:` entity can be inherited *and* is not visible outside the class (and its descendants)

Overriding Methods

When a child class defines a method with the same name *and signature* (sequence of parameters) as the parent, the child's version **overrides** the parent's version. Useful when the child class needs a different version than the parent.

Polymorphism means that the *type of the object*, not the type of the variable, decides which version of an overridden variable is executed.

These are not necessarily the same, since a variable can refer to any object whose class is a descendant of the variable's class.

When in doubt, draw a memory diagram!

Abstract Classes — Motivation

Consider a database for a veterinarian to keep track of medical and billing information for each patient.

- Each patient is someone's pet (e.g., dog, bird).
- Some aspects of the vet's business are independent of the particular species (e.g., billing, owner info).
- Some aspects depend critically on the species (e.g., the vaccination schedule, diet recommendations).

An obvious organization is to have a `Pet` superclass, and to have `Dog`, `Bird`, etc. subclasses.

Note that it does not make sense to create a `Pet` *object* — every pet is actually some particular species, that is, an instance of one of the subclasses.

The `Pet` class is used to group together common code, but is not complete by itself.

Rules for Abstract Classes and Methods

- Only instance methods can be declared abstract (not static methods).
- Any class with an abstract method must be declared as abstract.
- A class may be declared abstract even if it has no abstract methods.
- An abstract class cannot be instantiated (no objects created of that class).
- A non-abstract subclass of an abstract class must override each abstract method of the superclass and provide an implementation (method body) for it.
- If a subclass of an abstract class does not implement all of the abstract methods that it inherits, then it must also be an abstract class.

Since an abstract class cannot be instantiated, its variables and methods are not *directly* used. But they can be *indirectly* used via a (non-abstract) subclass.

Declaring an Interface

An **interface** is an abstract class taken to the extreme. It is like an abstract class in which everything is abstract — no methods have implementations.

```
interface <interface name> {
  <constant declarations>  // public final
  <abstract method declarations> // public abstract
}
```

An interface provides

- a collection of related constants, and
- a collection of method signatures

For example:

```
interface InterestBearing {
  double annualRate = .06;
  double calculateInterest(int period);
}
```

Implementing an Interface

The syntax for “inheriting from” (called **implementing**) an interface I is:

```
class B implements I { ... }
```

For example:

```
class Account implements InterestBearing {
  protected double balance;
  public double calculateInterest(int period) {
  // argument period is assumed to be in months
    return balance * annualRate * period/12;
  }
}
```

The class `Account`

- can access the constant `annualRate` in the interface `InterestBearing`, and
- must provide an implementation of the abstract method `calculateInterest` in the interface `InterestBearing`

Abstract Classes vs. Interfaces

- An abstract class can be used as a repository of **common code** that is shared among its subclasses, but an interface contains no code.

- A class can implement **many** interfaces, but can only inherit from one class (in Java).

- Both abstract classes and interfaces can be used to group together a collection of **related constants**.

Object-Oriented Design

The design of a software system is an iterative process.

- choose initial set of objects based on requirements
- develop behaviors and scenarios for the objects
- previous step may indicate that additional objects are needed
- develop behaviors and scenarios involving the additional objects
- etc.

As the design matures, objects are abstracted into classes:

- group common functionality into parent classes
- put unique functions into child classes
- determine what functionality will be public

Initial design effort focuses on the overall structure of the program. The algorithms for the methods are specified using pseudocode. Actual coding begins after most of the design structure and algorithm pseudocode are completed.

Deciding on Objects and Classes

Make some guesses about what the objects in the system are and try to arrange them into groups (which will be the classes). Although you should put serious thought into this, *don't try to do this perfectly on the first pass.*

Rule of Thumb: associate objects and classes with the physical entities they model.

Later you may need more intangible kinds of objects, such as an error message.

As you come up with the objects, some details (variables and methods) will be obvious. Document these and test them out with scenarios —

A **scenario** is a little sample execution of a piece of code. You walk through what you envision happening for a particular input, in terms of the order in which methods are invoked and what gets returned.

Linked List

Linked lists are useful when size and shape of data structure cannot be predicted in advance.

Linked lists are an example of **dynamic** data structures — their size, and even shape, can change during execution.

Separate blocks of storage are **linked** together using **pointers**. Blocks are *not necessarily contiguous*.

Linked representations are an important alternative to sequential representations (**arrays**).

Many key abstract data types (lists, stacks, queues, sets, trees, tables) can be represented with either linked structures or with arrays.

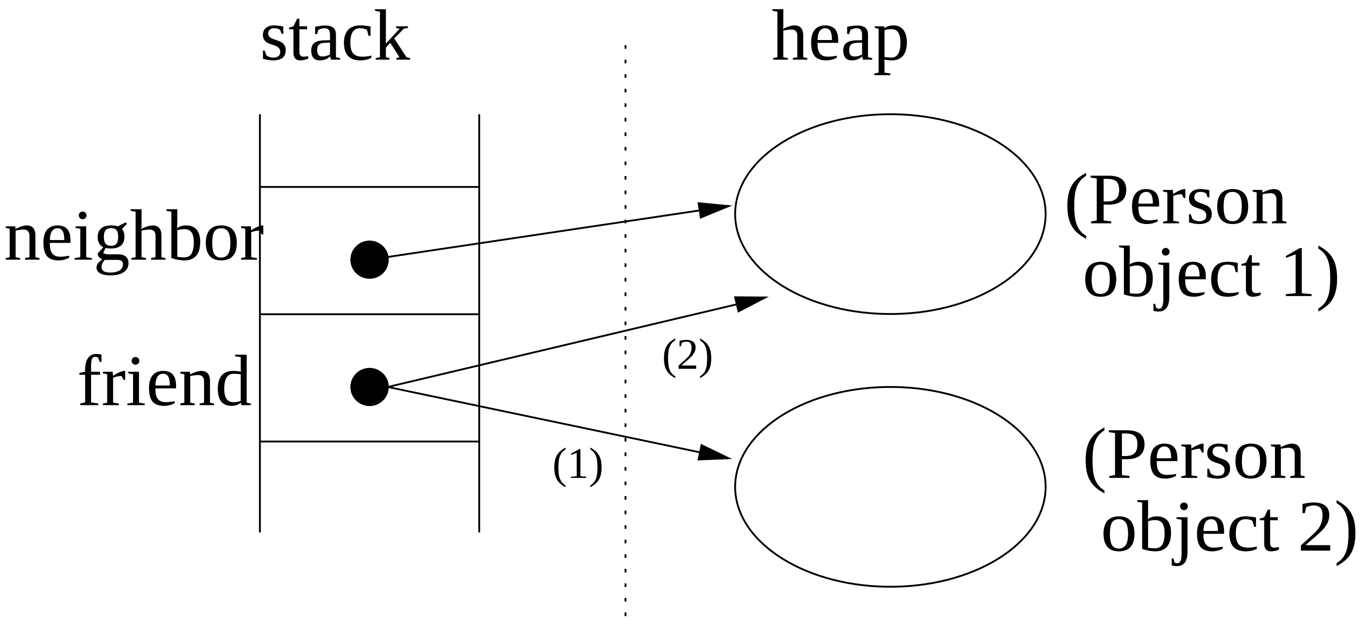
Important to understand the performance tradeoffs in the choice of representation.

Pointers

Pointers in Java are called **references**.

References to objects are essentially pointers.

However, you cannot do arithmetic on pointers in Java (unlike C, for instance).



Linear Linked Lists

The list consists of a series of **nodes**, or storage blocks.

Each node contains

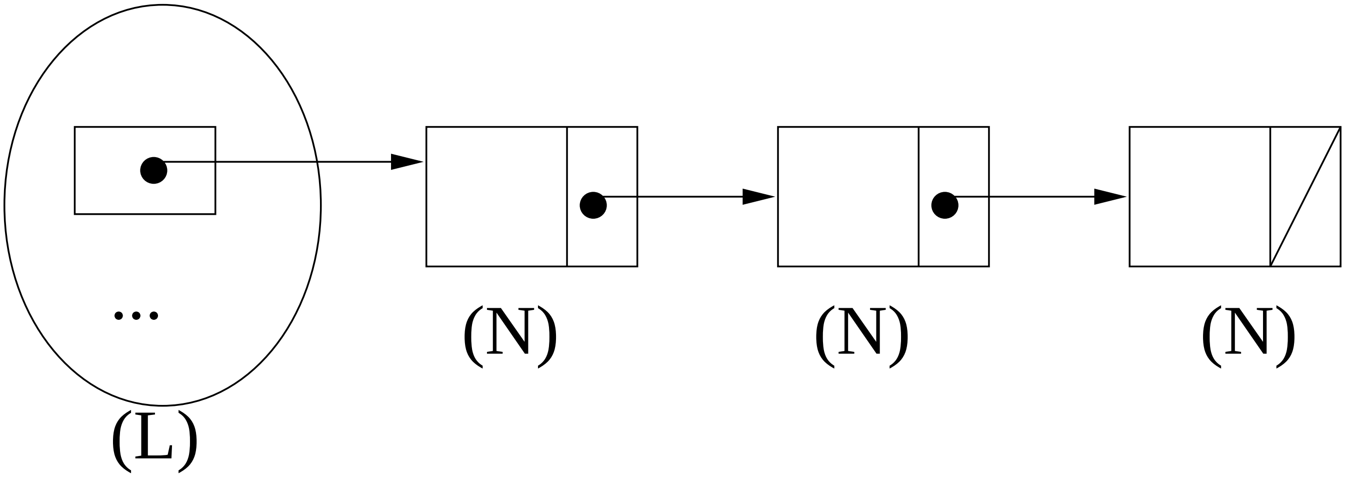
- **link** component, which points to next node in list
- other data components

To realize this idea in Java:

- each node is an object of some class N
- class N contains instance variables
  - `link`, whose type is N (reference to an object of type N)
  - other data components
- another class L contains a pointer to N object at the front of the list and other bookkeeping info about the entire list

Linear Linked Lists (cont’d)

Here is a diagram of the heap:



Space complexity: $O(n \cdot s)$ for a linked list of n nodes, each of size s.

Linked List Example — Node Class

For a linked list of books, first define a class that represents individual list elements (nodes).

```
class BookNode {
  String title;
  int numPages;
  BookNode link; // ptr to next book
                 // in linked list
  BookNode(String name, int pages) {
    title = name;
    numPages = pages;
    link = null; // points to nothing
  }
  ...
}
```

The type of the link variable is the *same* as the class being defined — **recursive data type**.

Linked List Example — List Class

Then define a class that represents the list itself. What should it contain?

- a pointer to the first node in the list
- length of the list
- other bookkeeping info...

It does NOT explicitly contain all the nodes of the list — they are part of this class indirectly, due to pointers.

```
class BookList {
  BookNode first;  // pointer to first
                   // elt. of list
  int size;        // # elts. in list

  BookList() {     // initially empty
    first = null;
    size = 0;
  }
  ...
}
```

Linked List Operations

What should be the operations on a linked list?

- insert another node — where?
  - front
  - end
  - before or after a specified node
- delete a node — which one?
  - first one
  - last one
  - the one containing certain data
- scan the whole list, doing something (like printing)

Add some instance methods to the `BookList` class:

```
void insertAtFront(BookNode node) ...
void insertAtEnd(BookNode node) ...
BookNode deleteFirst() ...
void printList() ...
...
```

Using a Linked List

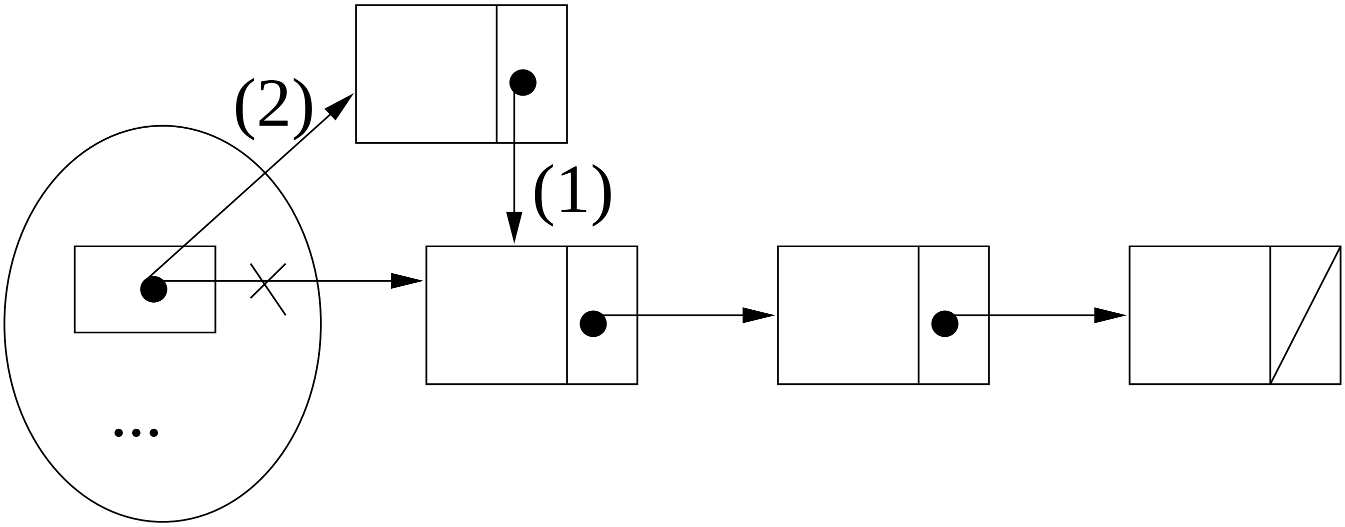
Example:

```
...
BookList myBooks = new BookList();
for (int i = 0; i < numBooks; i++) {
  BookNode bk = getBook();
  myBooks.insertAtEnd(bk);
}
myBooks.printList();
...
```

Inserting at the Front of a Linked List

Pseudocode:

```
1. make new node's link point to front of list
2. indicate that new node is first node in list
```

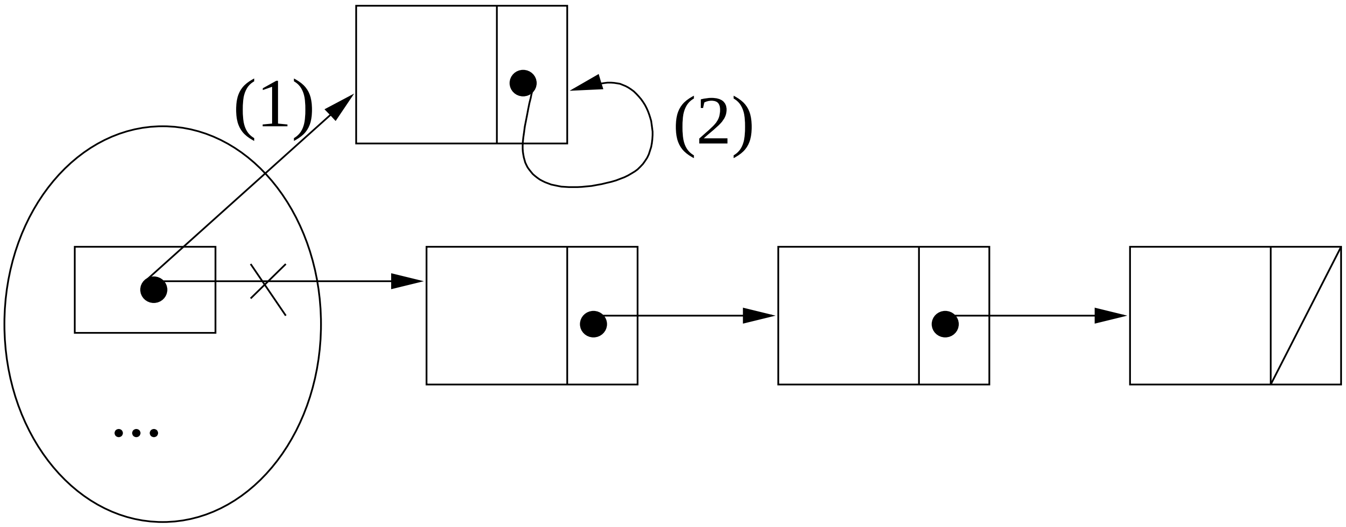


In Java (assuming the parameter is not null):

```
void insertAtFront(BookNode newNode) {
  newNode.link = first;  // step 1
  first = newNode;       // step 2
  size++;
}
```

Inserting at the Front of a Linked List (cont'd)

What happens if we do step 1 and step 2 in the opposite order?



We get a cycle, and the old list is LOST! **Be sure you don't lose access to your data!**

Time Complexity: $O(1)$, because we do a constant amount of work, no matter how many nodes are in the list.

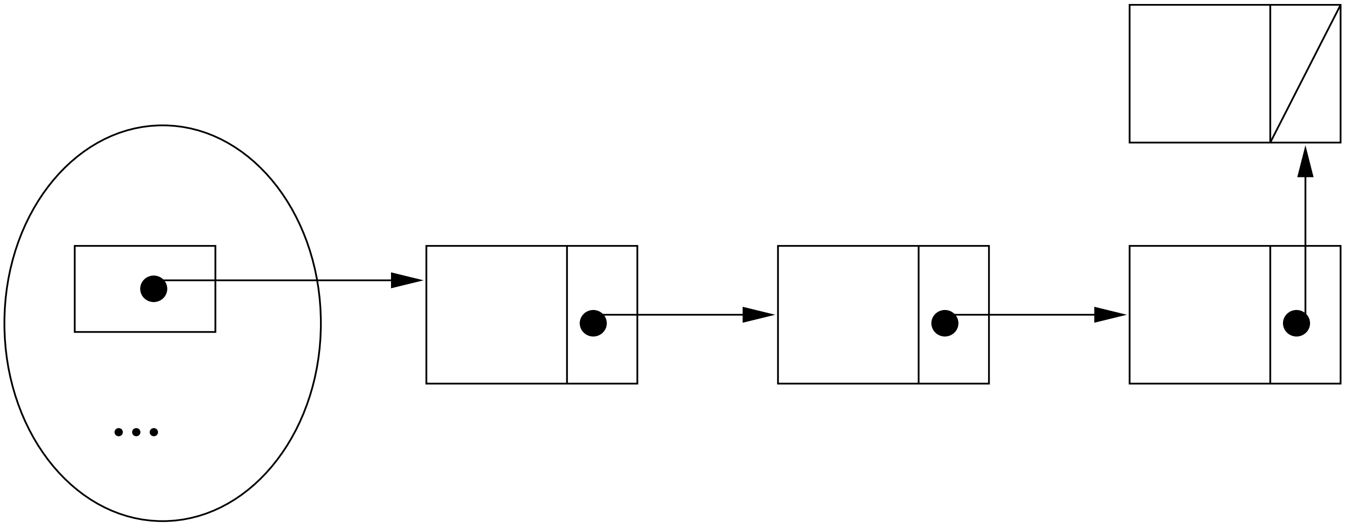
Inserting at the End of a Linked List

First, assume the list is empty (i.e., `first` equals `null`).

1. set the new node's link to `null`
2. set `first` to the new node.

Now, assume the list is not empty (i.e., `first` does not equal `null`).

1. find the last node, `node`, of the list.
2. set `node.link` to the new node.



How do we do step 1? Search through the list, starting with `first`, and following `link` pointers, until reaching the last node (i.e., the node whose `link` is `null`).

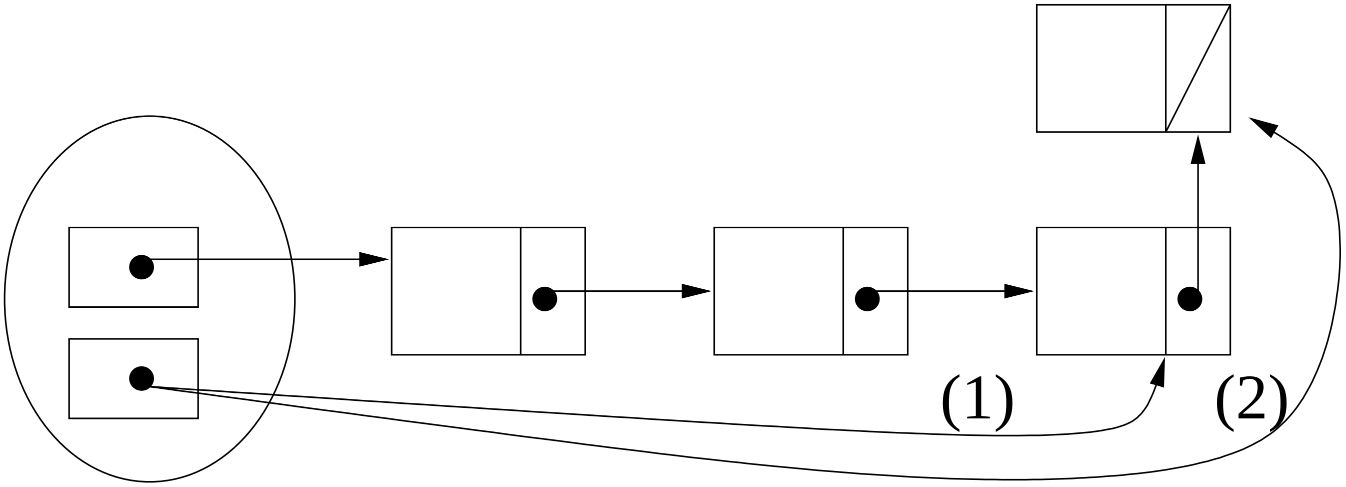
Inserting at the End of a Linked List (cont’d)

```
void insertAtEnd(BookNode newNode) {
  if (first == null) {
    first = newNode;
    newNode.link = null;
  }
  else {
    BookNode cur = first;
    while ( cur.link != null )
      cur = cur.link;
    // cur is last in list
    cur.link = newNode;
    newNode.link = null;
  }
  size++;
}
```

Time Complexity: Everything except the while loop takes constant time. Each iteration of the while loop takes constant time. There are n iterations of the while loop. So $O(1 + 1 \cdot n) = O(n)$.

Using a Last Pointer

To improve running time, keep a pointer to the last node in the list class, as well as the first node.



Time Complexity: O(1), independent of size of list.

```
class BookList {
  BookNode first;  // pointer to first
  BookNode last;   // pointer to last
  int size;        // # elts. in list
  // continued on next slide
```

Using a Last Pointer (cont'd)

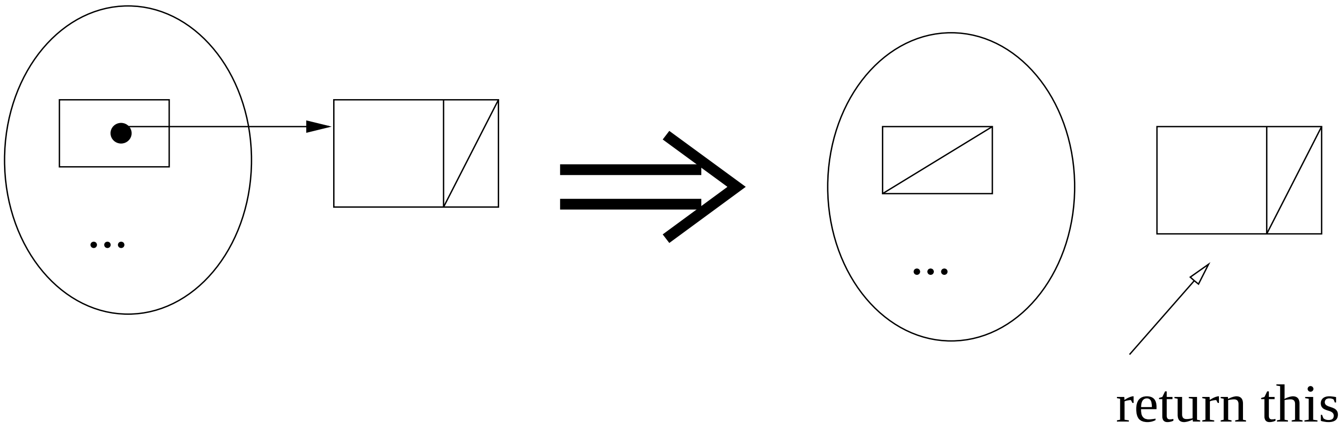
```
  // continued from previous slide
  BookList() {     // initially empty
    first = null;
    last = null;
    size = 0;
  }
  void insertAtEnd(BookNode newNode) {
    if (first == null) {
      first = newNode;
      last = newNode;
      newNode.link = null;
    }
    else {
      last.link = newNode;
      last = newNode;
      newNode.link = null;
    }
    size++;
  }
}
```

Deleting Last Node from Linked List

Suppose we want to delete the node at the end of the list and return the deleted node.

First, let's handle the **boundary conditions**:

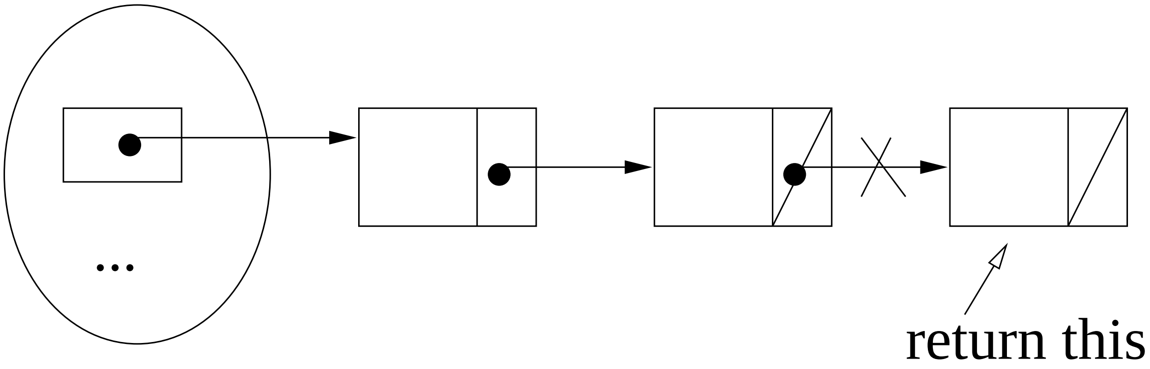
- If the list is empty, then nothing needs to be changed — just return `null`.
- If the list has only one element (i.e., if `first.link` is tt null), then set `first` to `null` and return the node that `first` used to point to (use a temp).



Deleting Last Node from Linked List (cont'd)

Suppose the list has at least two elements.
First attempt:

1. find the last element, z, of the list
2. change link of previous node to null
3. return z



Step 1 can be done as before.

What about step 2? How do we know which node is the preceding one? The links are *one-directional*. While traversing list, keep track of *next* node as well as current:

1. march down the list with two pointers, `cur` and `next`, until `next` is the last node in the list.
2. set `cur.link` to `null`

Deleting Last Node from Linked List (cont'd)

```
public BookNode deleteLast() {

  if (first == null) return null; // empty list

  if (first.link == null) {  // list w/ 1 elt.
    BookNode temp = first;
    first = null;
    size--;
    return temp;
  }

  BookNode cur = first;       // list w/ > 1 elt
  BookNode next = cur.link;

  while (next.link != null) {
    cur = next;
    next = cur.link;
  }

  cur.link = null;   // truncate list at 2nd-to-last
  size--;
  return next;
}
```

Time Complexity: The running time here, like the last example, is $O(n)$, proportional to size of list.

Would it help to keep a last pointer? No! We still can't follow the pointer backward.

Linked Lists Pitfalls

- **Check that a link is not null before following it!** Example:

  ```
  node.link = null;
  node = node.link;  // node is null
  node.link = null;  // ERROR!
  ```

- **Mark end of list** by setting the `link` field of the last node to `null`.
- **Be careful with boundary cases!** Examples of boundary cases are the empty list, the list with one element (in some cases), the first node, the last node, etc. (depending on what you are doing).
- **Draw memory diagrams!** These can usually make it clear what you need to do, and in what order.
- **Don't lose access to needed objects!** Make sure you change pointers in a safe order.

Linked Lists vs. Arrays

Space complexity: Linked list has overhead of one pointer per data item. Array has overhead of over-estimating required space, since size is not dynamic.

Time Complexity (n data items):

	singly linked	singly linked, last ptr	doubly linked	doubly linked, last ptr	array
insert front	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(n)$
insert end	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$
delete first	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(n)$
delete last	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
search	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Linked Lists vs. Arrays (cont'd)

Suppose the items in the sequence are in sorted order. Then data items must be inserted in the correct place. But perhaps this will make searching for an item easier. Break the insertion process into two parts:

1. search for correct place to insert, call the resulting place i
2. insert at current place

	singly linked	singly linked, last ptr	doubly linked	doubly linked, last ptr	array
search	$O(i)$	$O(i)$	$O(i)$	$O(i)$	$O(\log n)$
insert	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(n - i)$

Linked Lists vs. Arrays (cont'd)

Tradeoff:

- linked list:
  - insert is fast
  - search is slow

  because nodes are not contiguous in memory

- arrays:
  - insert is slow
  - search is fast (binary search)

  because nodes *are* contiguous in memory

Binary search cannot be used on linked lists because it relies on relationship between address of data (array index) and value of data.

Later we will see some other data structures that try to get the best of both worlds.

Other Linked Structures

We don't have to restrict ourselves to just having one link instance variable per node. We can get arbitrarily complicated linked structures.

Some of the more common and useful ones are:

- doubly linked list — have a forward link and a backward link per node. Reduces time for deleteLast from $O(n)$ to $O(1)$. Penalty is extra space for keeping backward links, which totals $O(n)$ space, a constant amount *per node*.
- rings — link of last node points to first node, circular.
- trees — we'll see more later about these.
- general graphs — arbitrary number of links per node that point to arbitrary other nodes

Recursion

Idea of **recursion** is closely related to the principle of *mathematical induction*.

- Figure out how to solve the problem for small problem instances.
- Assume you have a solution for smaller problem instances.
- Figure out how to do a little more work which, in combination with solution(s) to smaller instance(s), solves the larger problem instance.

This is also an application of *divide and conquer*.

Rules for recursive programs:

- There must be a stopping (base) case.
- Recursive call(s) must get you closer to a stopping case.

Stack Frames for Recursive Methods

When a recursive method is executed, *each invocation of the method gets a* **separate** *stack frame.* Thus each invocation has a separate copy of

- formal parameters
- local variables
- return value

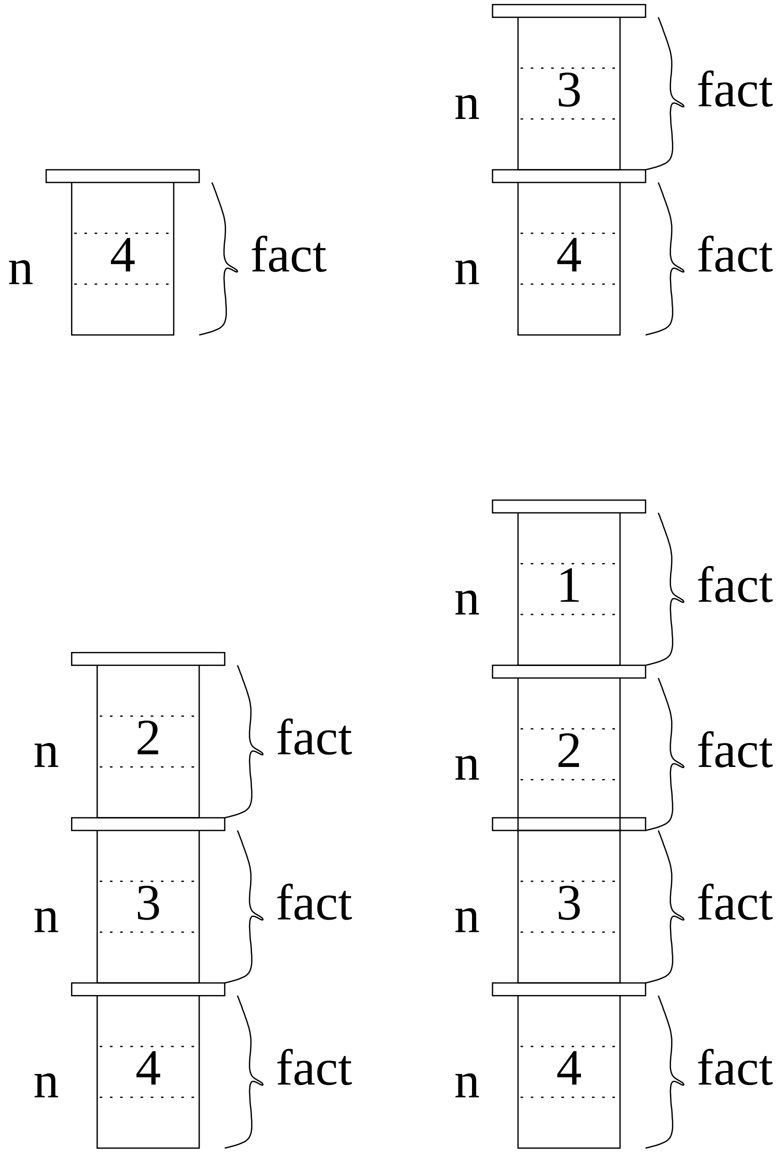
Example:
The factorial of n, represented $n!$, is calculated as $n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$.

To compute $n!$:

```
int fact(int n) {
  if (n == 1) return 1;
  else return n * fact(n-1);
}
```

Stack Frames for Factorial Example

Stack frames when calling `fact(4)` :



Towers of Hanoi

Towers of Hanoi is is an example of a problem that is *much* easier to solve using recursion than not using recursion.

- There are 3 pegs and n disks, all of different sizes
- Initially all disks are on the start peg, stacked in decreasing size, with largest on bottom and smallest on top.
- We must move all the disks to the end peg *one at a time and without ever putting a larger disk on top of a smaller disk.*
- The third peg can be used as a spare.

Example: $n = 2$. Solution is:

1. Move smaller disk from start peg to spare peg.
2. Move larger disk from start peg to end peg.
3. Move smaller disk from spare peg to end peg.

For larger n, it becomes difficult to figure out.

Recursive Solution to Towers of Hanoi

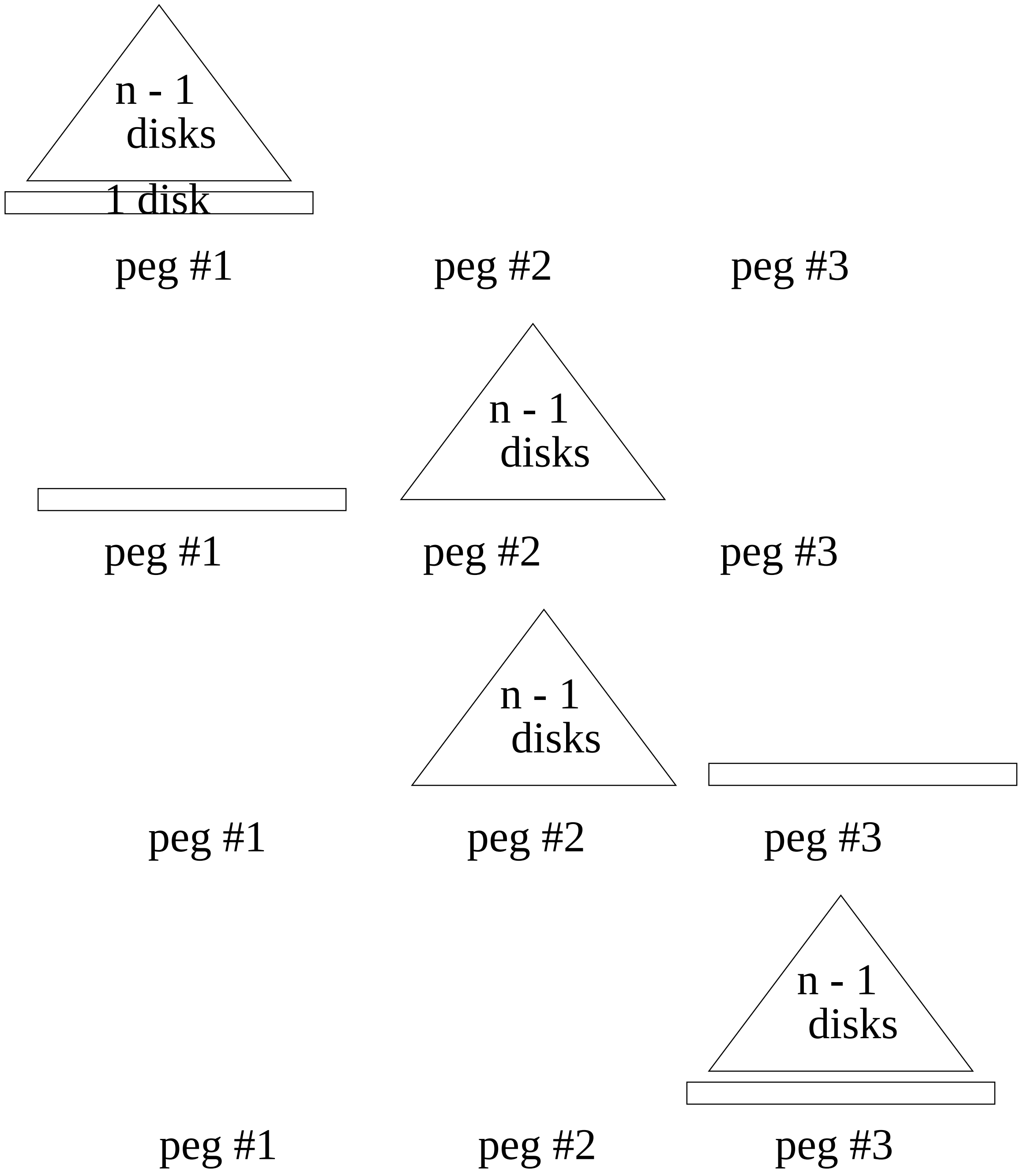
Using recursion can help. Suppose someone gives us a method M to move $n-1$ pegs. We can use it to solve the problem for n pegs as follows:

1. Move the top $n-1$ disks from the start peg to the spare peg using method M.
2. Move the bottom disk directly from the start peg to the end peg.
3. Move the $n-1$ disks from the spare peg to the end peg using method M.

Steps 1 and 3 will be done using recursion.

Stopping case? When $n = 1$, the peg can be moved directly.

Figure for Towers of Hanoi



Recursive Solution to Towers of Hanoi (cont’d)

The output of the program will be a list of instructions.

```
void Towers(int n, int start, int finish, int spare) {
  if (n == 1)
    S.o.p("move from " + start + " to " + finish);
  else {
    // move n-1 disks from start to spare:
    Towers(n-1, start, spare, finish)
    // move bottom disk directly to finish:
    S.o.p("move from " + start + " to " + finish);
    // move n-1 disks from spare to finish:
    Towers(n-1, spare, finish, start)
  }
}
```

To call this method, suppose you have 4 pegs and you want to use peg 1 as the start peg, peg 3 as the finish peg, and peg 2 as the spare peg:

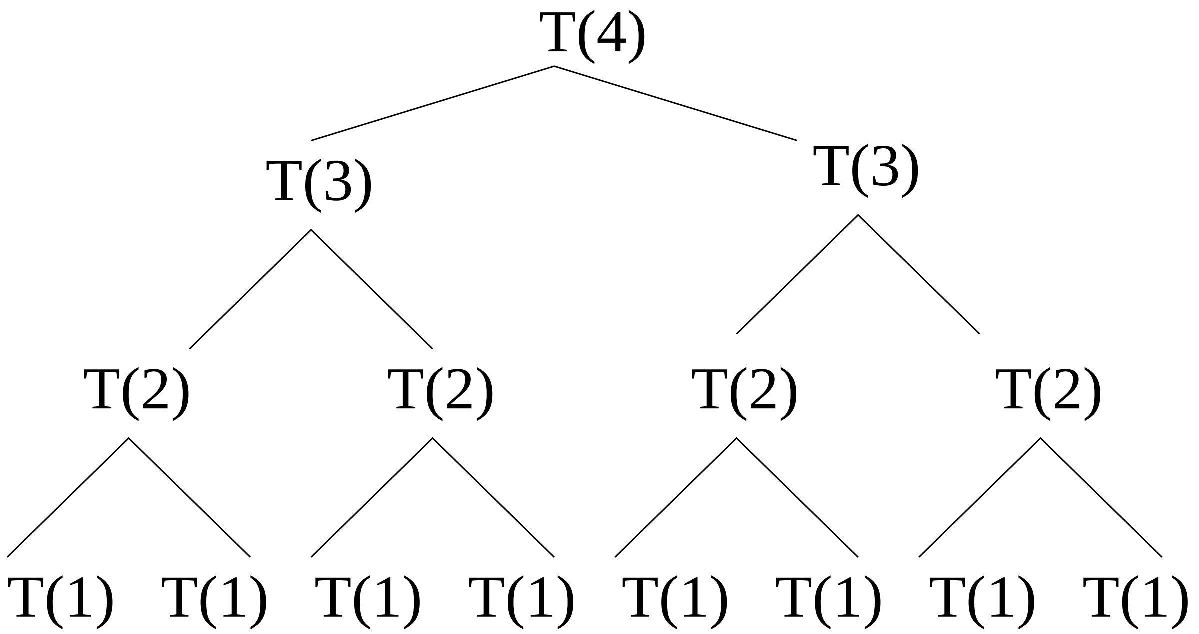
```
Towers(4, 1, 3, 2);
```

Time Complexity of Towers of Hanoi Solution

Time Complexity: Asymptotically proportional to the number of instructions output.

Each instantiation of the method prints one instruction.

To count the number of instantiations, draw a **call tree**:



Number of vertices in the tree is $2^n - 1$.

Therefore time complexity is $O(2^n)$.

Parsing Arithmetic Expressions

An important part of a compiler is the **parser**, which checks whether the input program conforms to the *grammar*, or *syntax*, of the programming language.

An important part of this problem is to check whether arithmetic expressions are well-formed. For example:

- $a + (b - (x/y))$ — OK
- $a + +b/z$ — BAD
- $(a)) * c$ — BAD

To simplify the problem:

- Assume that the operands are single-letter variables (a through z) or single digit numbers (0 through 9)
- Only consider operators $+$, $-$, $*$ and $/$

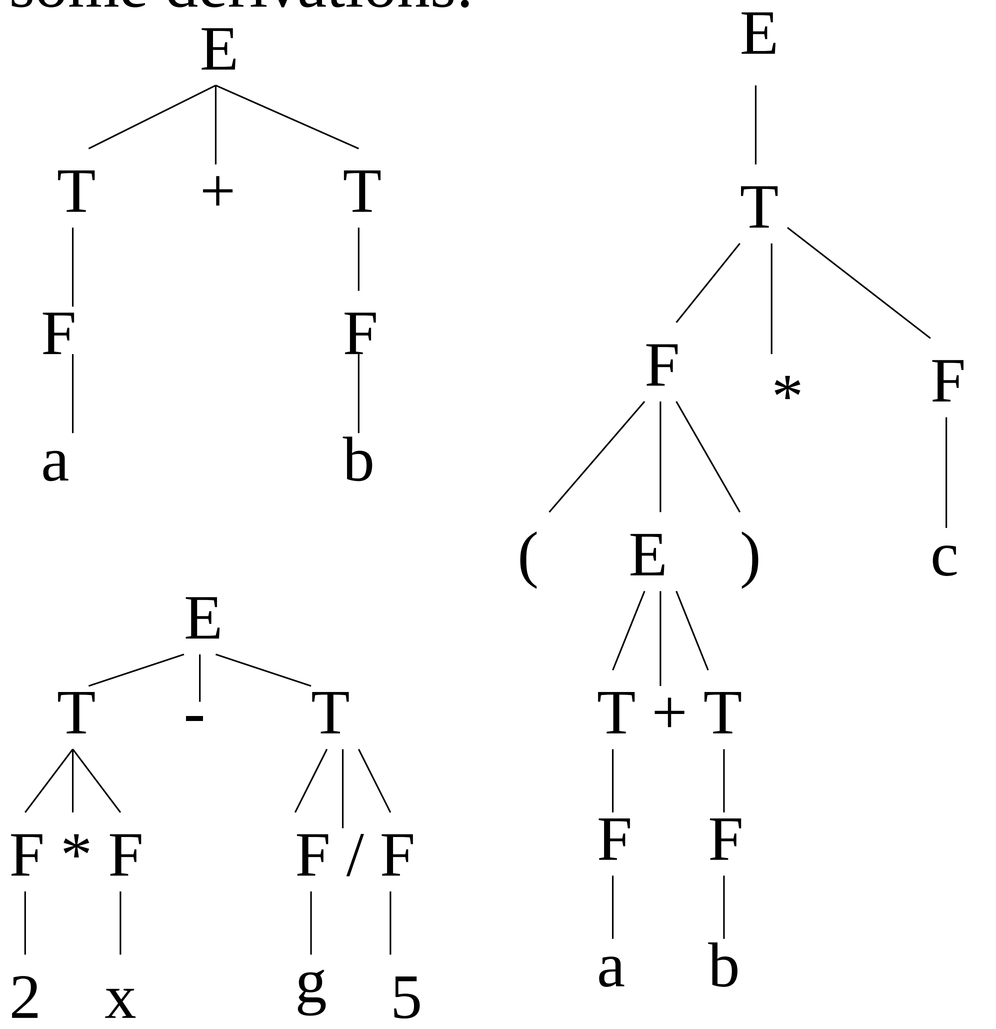
The correct syntax for arithmetic expressions can be described using **grammar rules**. Then a particular input can be checked to see whether it is derivable from the grammar rules.

A Grammar for Arithmetic Expressions

Sample Rules: (| means “or”)

1. expression $\rightarrow$ term $+$ term $|$ term $-$ term $|$ term
2. term $\rightarrow$ factor $*$ factor $|$ factor $/$ factor $|$ factor
3. factor $\rightarrow$ (expression) $|$ letter $|$ digit

Here are some derivations:



Recursive Parsing Algorithm

Idea is to try to obtain an expression from the input. To do this, try to obtain from the input

- a term, or
- a term, followed by a $+$, followed by another term, or
- a term, followed by a $-$, followed by another term.

To obtain a term from the input (starting at the current position), try to obtain

- a factor
- a factor, followed by a $*$, followed by another factor, or
- a factor, followed by a $/$, followed by another factor.

To obtain a factor from the input (starting at the current position), try to obtain

- a letter, or a digit, or
- a (, followed by an expression, followed by a).

Note the **indirect recursion**.

Recursive Parsing Algorithm (cont'd)

At the top level:

```
boolean valid(String input) {
  String remainder = getExpr(input);
  return ((remainder != null) &&
          (remainder.length() == 0));
}
```

`getExpr` recognizes an expression at the beginning of `input` and returns the rest of the string, which will be the empty string if nothing is left over. If a syntax error is encountered, it returns `null`. (Does not handle white space in the input.)

```
String getExpr(String input) {
  String remainder = getTerm(input);
  if ((remainder != null) &&
      (remainder.startsWith("+") ||
       remainder.startsWith("-")))
    // lop off + or - before looking for a term
    remainder = getTerm(remainder.substring(1));
  return remainder;
}
```

Recursive Parsing Algorithm (cont’d)

```
String getTerm(String input) {
  String remainder = getFactor(input);
  if ((remainder != null) &&
      (remainder.startsWith("*") ||
       remainder.startsWith("/")))
    // lop off * or / before looking for a factor
    remainder = getFactor(remainder.substring(1));
  return remainder;
}
String getFactor(String input) {
  if (Character.isLetterOrDigit(input.charAt(0)))
    // lop off digit or char before returning
    return input.substring(1);
  else if (input.startsWith("(")) {
    // lop off ( before looking for expression
    String remainder = getExpr(input.substring(1));
    if ((remainder == null) ||
        !remainder.startsWith(")"))
      return null;        // syntax error
    // lop off ) before returning
    else return remainder.substring(1);
  }
  else return null;       // syntax error
}
```

Abstract Data Types

An **abstract data type** (ADT) defines entities that have

- state and
- operations that can change the state and return information

ADTs provide the benefits of abstraction, modularity, and information hiding.

There is a *strict separation* between the public interface, or specification, and the private implementation of the ADT.

This separation facilitates correctness proofs of programs/algorithms that *use* entities of the ADT.

ADTs are easily achieved in Java using classes and the appropriate visibility modifiers.

ADT Example: Priority Queue Specification

The **priority queue** ADT is useful in many situations. Here is its specification:

- The state is a set of elements that can be compared to each other according to some "priority".
- The operations on a priority queue are:
  - make a new empty priority queue
  - insert a new element into the priority queue
  - remove the *highest priority element* from the priority queue

Note that *there is no operation* to remove any other element.

Example applications:

- Pay the bill among all your outstanding bills that has the closest deadline.
- Provide medical treatment to the sickest person in the hospital's emergency room.

Using a Priority Queue to Sort a List of Integers

Even without knowing anything about *how* a priority queue might be implemented, we can take advantage of its operations to solve other problems.

For example, to sort a list of numbers:

- Insert each number in the list into a priority queue; (*smallest* number has highest priority)
- Successively remove the highest priority (i.e., smallest) number until the priority queue is empty.
- Store the removed numbers in order.

```
void sortPQ (int[] A) {
  int n = A.length;
  PriorityQueue pq = new PriorityQueue();
  for (int i = 0; i < n; i++) pq.insert(A[i]);
  for (int i = 0; i < n; i++) A[i] = pq.remove();
}
```

This method correctly sorts the number, for ANY (correct) implementation of the PriorityQueue class.

Time Complexity: $O(n(f(n) + g(n)))$, where $f(n)$ is the time to insert and $g(n)$ is the time to remove.

Implementing a Priority Queue with an Array

```
class PriorityQueue {
  private int[] A = new int[100];
  private int next; // next inserted element goes here
  PriorityQueue() {
    next = 0;
  }
  public void insert(int x) { // no overflow check
    A[next] = x;
    next++;
    }
  public int remove() { // no underflow check
    int high = A[0];
    int highLoc = 0;
    for (int cur = 1; cur < next; cur++) {
      if (high < A[cur]) {  // find highest priority elt
        high = A[cur];
        highLoc = cur;
      }
    }
    A[highLoc] = A[next-1]; // reorder array
    next--;
    return high;
  }
}
```

Time Complexity: insert $O(1)$, remove $O(n)$, sort $O(n^2)$.

Implementing a Priority Queue with a Linked List

Pseudocode:

- To insert an element: Insert the element at the head of the linked list. Time is $O(1)$.
- To remove the highest priority element:
  - Scan through the entire linked list, maintaining a pointer to the highest priority item found so far.
  - When the entire list has been scanned, splice the highest priority node out of the linked list.

  Time is $O(n)$.

Asymptotic running times are same as for the array.

Time to sort is again $O(n^2)$.

Can we do things faster by keeping the array, or linked list, elements in sorted order?

Warning: Do not confuse the implementation of the priority queue with a possible application of it (e.g., sorting).

Implementing a PQ with a Sorted Array

Keep the array elements in increasing order of priority. (If highest priority is smallest element, then elements will be in *decreasing* order).

Pseudocode:

- To insert an element: Starting at the end, search for correct location for new element while simultaneously shifting elements down to make room. Time is $O(n)$, due to the shifting.
- To remove the highest priority element:
  - Indicate that the effective size of the array has been decreased by one.
  - Return the element at the end of the effective part of the array.

  Time is $O(1)$, an improvement.

However, time to sort is still $O(n^2)$.

Implementing a PQ with a Sorted Linked List

Pseudocode:

- To insert an element: Scan down the linked list until finding the correct spot to insert the new element. Insert it there. Time is $O(n)$, due to the scan.
- To remove the highest priority element: Remove the last element of the list. Time is $O(1)$.

Asymptotic times are the same as for a sorted array; time to sort is still $O(n^2)$.

Generic PQ Implementation Using Java

To avoid rewriting the priority queue implementation for every different kind of element (integer, double, String, user-defined classes, etc.), we can use Java's `interface` feature.

All that is required is a way to compare two elements.

```
interface ComparisonKey {
  int HIGHER = -1;
  int LOWER = 1;
  int EQUAL = 0;

// k1.compareTo(k2) returns
// EQUAL if k1 "equals" k2
// HIGHER if k1 "is higher than" k2
// LOWER if k1 "is lower than" k2

    int compareTo(ComparisonKey k);
}
```

Using the `ComparisonKey` Interface

- Change the specification of the `PriorityQueue` class to consist of a collection of `ComparisonKey`'s, with the methods
  - `insert`, which takes a `ComparisonKey` as a parameter (instead of an int)
  - `remove`, which returns a `ComparisonKey` (instead of an int)
- Any class that implements `ComparisonKey` can be used in place of `ComparisonKey`.
- Define a class called `PQItem` that implements `ComparisonKey`.
- `sortPQ`, the sorting algorithm that uses a priority queue, can also be generalized to work on an array of `ComparisonKey`'s.

Generic Implementation of PQ with Array

```
class PriorityQueue {
  private ComparisonKey[] A =
          new ComparisonKey[100]; // int -> CK
  private int next;
  PriorityQueue() {
    next = 0;
  }
  public void insert(ComparisonKey x) { // int -> CK
    A[next] = x;
    next++;
  }
  public ComparisonKey remove() { // int -> CK
    ComparisonKey high = A[0];    // int -> CK
    int highLoc = 0;
    for (int cur = 1; cur < next; cur++) {
      if (high.compareTo(A[cur]) ==
          ComparisonKey.LOWER) { // use compareTo method
        high = A[cur];
        highLoc = cur;
      }
    }
    A[highLoc] = A[next-1];
    next--;
    return high;
  }
}
```

Implementing the Generic `PQItem`

Here is a possible `PQItem` class for integers. Note overhead vs. flexibility.

```
class PQItem implements ComparisonKey {
  private int key;
  PQItem(int value) { // constructor
    key = value;
  }
  public int compareTo(ComparisonKey k) {
    int otherKey = ( (PQItem) k).key; // extract int
    if (key < otherKey) return HIGHER;
    if (key > otherKey) return LOWER;
    return EQUAL;
  }
}
```

For a `PQItem` class for strings:

- make `key` a string
- make parameter to the constructor a string
- the method `compareTo` can use the `compareTo` method for strings

Generic `PQItem`'s (cont'd)

This approach is particularly powerful since we can define the priority any way we want to for our own user-defined class.

Suppose the items are student records.

One form of priority might be according to GPA, breaking ties according to number of hours completed.

Another form might be alphabetical order of name, breaking ties according to year of birth.

All those decisions will be encapsulated inside the `compareTo` method of the `PQItem` class.

Sorting with Generic PQ

Finally, here is the sorting algorithm:

```
void sortPQ (ComparisonKey[] A) {
  int n = A.length;
  PriorityQueue pq =
             new PriorityQueue();
  for (int i = 0; i < n; i++)
             pq.insert(A[i]);
  for (int i = 0; i < n; i++)
             A[i] = pq.remove();
}
```

The only difference from before is the type of A.

IMPORTANT TO NOTICE:

- The `PriorityQueue` class will NOT be changed even if the `PQItem` class changes.
- The `sortPQ` method will NOT be changed even if the `PQItem` class or `PriorityQueue` class changes.

Importance of Modularity and Information Hiding

Why is it valuable to be able to do these kinds of things?

The public/private visibility modifiers of Java, and the discipline of not making the internal details be available outside are forms of **information hiding**.

Information hiding promotes **modular** programming — you can switch the implementation of one class without affecting (correctness of) other classes.

The key to abstraction is separating WHAT (the specification) from HOW (the implementation).