What is computer science?

> *"The discipline of computing is the systematic study of the algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation, and application. The fundamental question underlying all of computing is, 'What can be (efficiently) automated?' "*
>
> Denning et al., "Computing as a Discipline," *Communications of the ACM 32*, 1 (Jan 1989) pp. 9–23.

- a young discipline that arose from several more established fields (mathematics, science, engineering)
  - key words: algorithm, information ("informatics")
- term coined by George Forsythe, a numerical analyst and founding head (1965-1972) of Stanford University's CS Department
- CS at Waterloo: formally founded in 1967 as the Department of Applied Analysis and Computer Science

Aspects of computer science

- Design (from engineering)
  - establish requirements and specifications; create artifacts based on sound design principles
  - application: create hardware and software that is flexible, efficient, and usable
- Theory (from mathematics)
  - develop model; prove theorems
  - application: analyze the efficiency of algorithms before implementation; discover limits to computation
- Experimentation (from science)
  - form hypothesis, design experiments, and test predictions
  - application: simulate real-world situations; test effectiveness of programs whose behaviour cannot be modelled well

These aspects appear throughout CS, often concurrently.

Abstraction

"Fundamentally, computer science is a science of abstraction—creating the right model for a problem and devising the appropriate mechanizable techniques to solve it. Confronted with a problem, we must create an abstraction of that problem that can be represented and manipulated inside a computer. Through these manipulations, we try to find a solution to the original problem."

A. V. Aho & J. D. Ullman, *Foundations of Computer Science*

Object-oriented design concentrates on data abstraction:

1. identify entities occurring in problem domain
2. partition similar entities into sets
3. identify properties and operations common to all entities in each set
4. define interfaces to support operations on objects that belong to each entity set
5. define code modules to implement the entity sets
    - choose representations for objects
    - implement methods for constructing and manipulating objects conforming to the interfaces

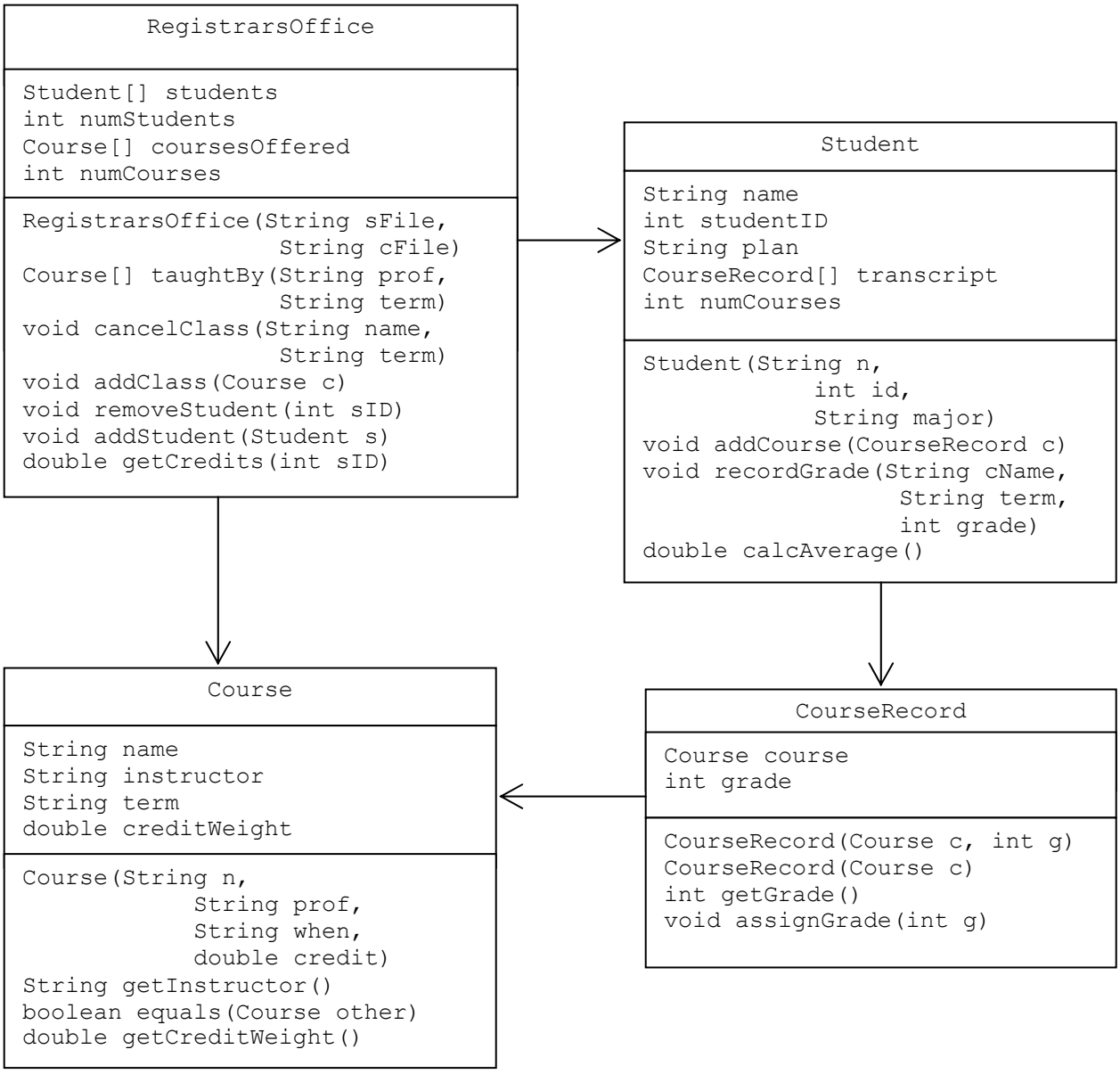
Problem Description

- Develop a program which models UW's Registrar's Office. The office maintains a list of all students and all courses offered by the university. For each student, maintain personal information (name, ID, plan and transcript). For each course, maintain general information (course title, instructor, term offered, and credit weight).
- Note: The goal today is **not** to implement this entire program. The goal is to give some suggestions on how to start the process of implementation.

Introduction to Design

- What classes are needed?
- For each class:
  - What methods are needed?
  - What information is needed? Not needed?
- Note: In CS 126, you will be given a design for all programming assignments. However, you should always understand why each class, method and instance variable is needed.

Class Diagram for Registrar's Office Program



Implementation Strategies (or, How Do I Start?)

- Start early!
- Read the description several times.
- Draw a picture of the data.
- Look at
  - Class dependencies
  - “Isolated” classes
    → Determine start class and implement.
  - Potential Problem here:
    * What if there is a cycle in the chart?
    →Program in "slices"
- Repeat until everything is done.

Implementing a Single Class

- Write constructor(s)
- Write stubs for all methods.
- Choose a non-trivial method
  - complete, and
  - test
- Continue until class is finished.

Testing a Class (or, Is it Correct?)

- Include a main method in each class.
- Create (instantiate) objects of the class type.
- Include code calling completed methods.
- Compare observed results to expected results.
- Note: In a subsequent lecture, we will discuss testing strategies in detail.

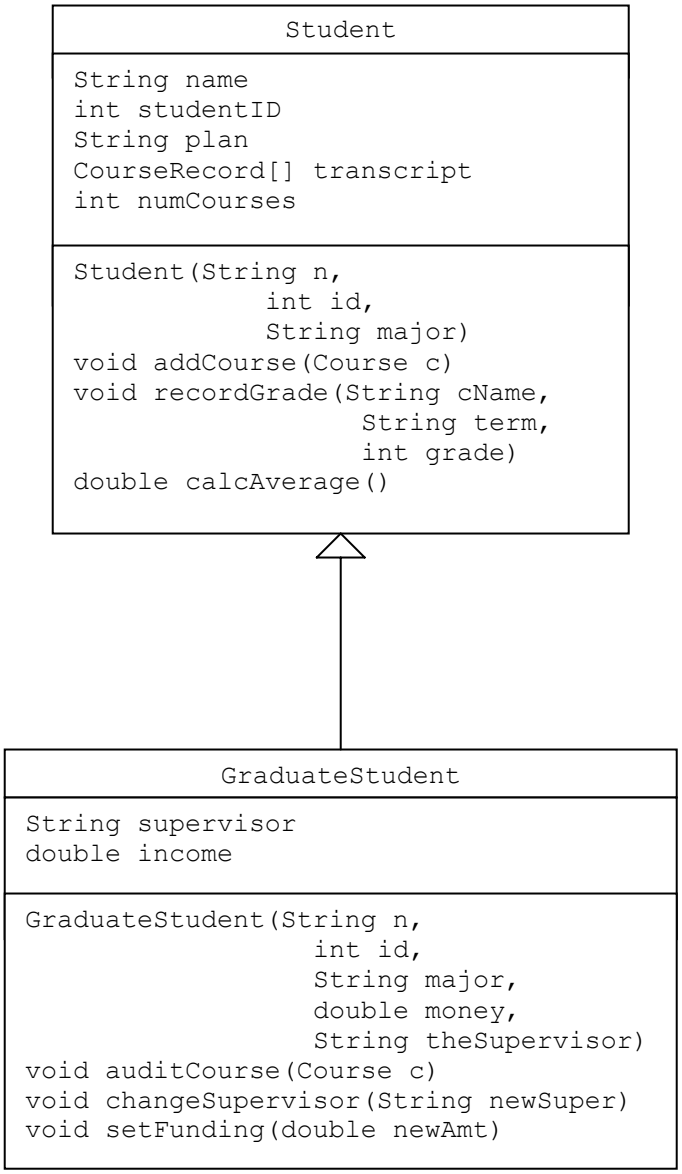
While doing all this …

- Document what each method does.
- Document blocks of code within each method, e.g.,
  - loops
  - if statements
  - major calculations
- Use step-wise refinement
  - use helper methods
- Re-use code, where appropriate
  - use helper methods
- Use constants to avoid “magic numbers”
- ***Start Early!!!***

Modified Problem Statement

- Develop a program which models UW's Registrar's Office. The office maintains a list of all students and all courses offered by the university. For each student, maintain personal information (name, ID, plan and transcript). **Graduate students (who receive financial support, have a supervisor, and can audit courses) are to be included as well.** For each course, maintain general information (course title, instructor, term offered, and credit weight).

Revised Partial Class Diagram



Programming with Inheritance

- Recall:
  - A `GraduateStudent` inherits all `Student` methods.
  - A `GraduateStudent` can override Student methods.
  - A `Student` variable can refer to a `GraduateStudent` object, but can only access Student methods.
  - Casting must be used to access `GraduateStudent` methods in this situation.

Abstract Data Types

- An abstract data type (ADT) includes
  - set of values (data space)
  - set of operators (methods)
- examples:
  - real numbers with addition, subtraction, absolute value, square root, ...
  - ordered sequences of characters with concatenate, substring, length, ...
  - sets of objects with union, intersection, size, ...
  - points in the plane with x-coordinate, distance, ...
- allows us to define further types tailored to applications' needs
- *independent of programming language*
- also applicable to procedural programming languages
- *independent* of data representation and operator implementation

ADTs (cont'd)

- Use the principle of *information hiding*

  coined by David Parnas in 1972

  "Every module [should be] characterized by its knowledge of a design decision which it hides from all others."

– particularly important to hide decisions that are likely to change (when they are found to be wrong or the environment has changed)

– benefits software maintenance and reuse

- ADT List

– collection of n objects indexed from 1 to n

  * $<i_1, i_2, i_3, \ldots, i_n>$

– can create an empty list

  * createList()

– can get the number of items in List s

  * s.size()

– can access element at an arbitrary index of List s

  * s.get(index)

– can insert new elements into List s

  * s.add(index, item)

– can remove elements from List s

  * s.remove(index) or s.removeAll()

– special check for empty List

  * s.isEmpty() ≡ (s.size() == 0)

Assertions in programs

- An assertion is:
  - a logical claim about the state of a program
  - located at a specific point in the program
  - assumed true whenever that point is reached

  Example: `0 ≤ i < a.length`
- precise, concise, and convincing documentation
- can be used to
  - guide the creation of a program (design)
  - reason formally about a program and verify its correctness (theory)
  - drive formulation of test cases and debugging strategy (experiment)

Properties of assertions

- comments
- snapshots of what is claimed to be true during execution at some specific point in the code
- noteworthy static statements
- *not* descriptions of action or change

Specification by assertions

- *preconditions*: assertions that *must be* true when the method is invoked
- *postconditions*: assertions that *will be* true when the method returns

Specify the syntax and semantics of each method:

- its signature :
  - name of method and type of result
  - names and types of parameters
- assumed properties of its inputs
- intended effects of its execution

```
public static int valueAt(int[] a, int i);
// pre: 0 ≤ i < a.length
// post: returns a[i]

public boolean equals(Object other);
// post: returns true iff this has the same value as other
```

- A method with preconditions and postconditions specifies a *contract* between the implementer and its users.
  - preconditions *need not* be checked *within* the method
- Pre- and postconditions give the basis of testing and of formal verification.

Specification of ADT List

```
public interface ListInterface {
/*  finite (possibly empty) collection of objects,
    indexed from 1 to size()
*/

  public void add(int index, Object item);
  // pre: 1 <= index <= size() + 1
  // post: Items from index to size have their index increased by one,
  //       item is inserted at index.

  public Object get(int index);
  // pre: 1 <= index <= size()
  // post: returns the item at position index in the list

  public boolean isEmpty();
  // post: returns true iff this is empty

  public Object remove(int index);
  // pre: 1 <= index <= size()
  // post: removes and returns the item at position index in the list
  //       and other items are renumbered accordingly

  public void removeAll();
  // post: this is empty

  public int size();
  // post: returns the number of items currently in this

}
```

What if preconditions not met?

- Calling routine is at fault
- Possible behaviour of technically correct implementations of called routine:

– Precondition not checked: unpredictable result
  - could return values that appear sensible
  - could cause abnormal program termination

– Precondition checked and error value returned
  - "unusual" value (e.g., `-1` for course mark)
  - some default value (e.g., `null` for item, where `null` might be a legitimate, although perhaps rare, item in the list)

– Precondition checked and exception raised
  - In Java, via statement:

```
throw new exceptionClass(msg);
```

- Dealing with exceptions in Java

– Possibility of throwing exception *must be declared* as part of method signature:

```
public void M() throws exceptionClass;
```

– Every routine calling such a method must
  - either *also* declare that the exception might be thrown (pass the buck)
  - or enclose method call in `try…catch` block

```
try { ... M(); ... }
catch (exceptionClass id) { ... }
```

Alternative specification of ADT List

- as defined by Carrano & Prichard (p. 440)

```
public interface ListInterface {
/*  finite (possibly empty) collection of objects,
    indexed from 1 to size()
*/

  public void add(int index, Object item) throws
       ListIndexOutOfBoundsException,ListException;
  //  post: If index >= 1, index <= size() + 1 and this is not full, then,
  //  item is at index and other items are renumbered accordingly;
  //  if index < 1 or index > size() + 1,
  //       throws ListIndexOutOfBoundsException;
  //  if list is full, throws ListException

  public Object get(int index) throws
       ListIndexOutOfBoundsException;
  // post: If 1 <= index <= size(), the item at position index in the list is
  //  returned; throws ListIndexOutOfBoundsException if index < 1 or
  //  index > size()

  public boolean isEmpty();
  // post: Returns true iff this is empty

  public void remove(int index) throws
       ListIndexOutOfBoundsException;
  // post: If 1 <= index <= size(), the item at position index in the list is
  //  deleted, and other items are renumbered accordingly; throws
  //  ListIndexOutOfBoundsException if index < 1 or index > size().

  public void removeAll();
  // post: this is empty

  public int size();
  // post: Returns the number of items that are currently in this

}
```

ADTs in Java

- Classes provide representations and implementations of an ADT allowing for their instantiation
- We would like to separate these details from the ADT's specification
- Java provides specific mechanism that it calls an *interface*.

– Java "interface" is a set of method signatures.
  * no code for the methods
  * no constructors

– Java interfaces can *extend* other Java interfaces.
  * interfaces form a hierarchy

– A *class* implements one or more interfaces.
  * For each method in an interface, such a class must provide a method that matches the interface method signature *exactly*.

```
interface Vehicle {...}
class MotorizedVehicle implements Vehicle {...}
class Car extends MotorizedVehicle {...}

interface TA extends Student, Employee {...}
class TeachAsst implements TA, Cloneable {...}
```

- syntax and semantics of the operators are defined by the interface

Java Data Types

- Program variables contain either
  - values of primitive types (**`int`**, **`char`**, etc.), *or*
  - references to arrays or to objects (values of subclasses of the class Object)

```
int temperature;
int[] dailyHighTemperatures;
Object highAndLowTemperatures;
ListInterface temperatureReadings;
```

    - Variable declared with a class name can refer to objects in any subclass.
    - Variables declared with an interface name can refer to objects in any class implementing that interface.

- Casting
  - When an object is constructed (using **`new`**), its type is established.

```
Square sq = new Square(3);
```

  - An object of type T can always be treated as if it were of a supertype of T (*upcast*).

```
Rectangle rec = sq;
```

    - Thus all objects can be treated as type Object.
    - If an object is cast to some supertype (upcast), it can later be *downcast* to its original type.

```
sq = (Square)rec; /* could raise exception */
```

Wrapper classes

To use primitive values in situations requiring objects, need to “wrap” them. For example:

```
public final class java.lang.Integer
                  extends java.lang.Number {
  ...
     public Integer(int value);
       // post: constructs wrapper object to hold value

     public boolean equals(Object obj);
       // post: returns true iff this.intValue() == obj.intValue()

     public int intValue();
       // post: returns value wrapped by this

     public String toString();
       // post: returns string representation of this.intValue();
    ...
}
```

- N.B., These wrapper classes define “immutable” objects: there are no “set” or “update” methods.
- Similar classes exist for booleans, doubles, etc.

ADT Hierarchy

- ADT SortedList is a finite collection of objects maintained in a specified order together with the operations:

– createSortedList, sortedIsEmpty, sortedSize, sortedAdd(item), sortedRemove(item), sortedGet(index), locateIndex(item)

- operations in common with ADT List can be put into a more general ADT
- BasicList ADT

– methods: size, isEmpty, removeAll, get

- hierarchy is reflected in the Java interfaces, not the Java classes
- classes that implement this interface must provide

– one or more constructors
  * basic constructor inherits default implementation

– correct implementation for each method
  * semantics of each method specified by pre-and postconditions
  * `equals` and `toString` inherit default implementations from `Object`

Representation Independence

- Define all variables using the name of the ADT's interface, *not* any of the classes that implement that interface.

```
ListInterface s;
...
s.add(i, x);
```

– ADT user need not be aware of representation.
– Implementation can be changed without altering any user code.

- Use a "data factory" to create all objects.

– cannot define a constructor in an interface
  * therefore cannot create instance of ADT without reference to its representation
– problem: every use of **new** exposes the representation
– solution: restrict **new** to special "creator" methods

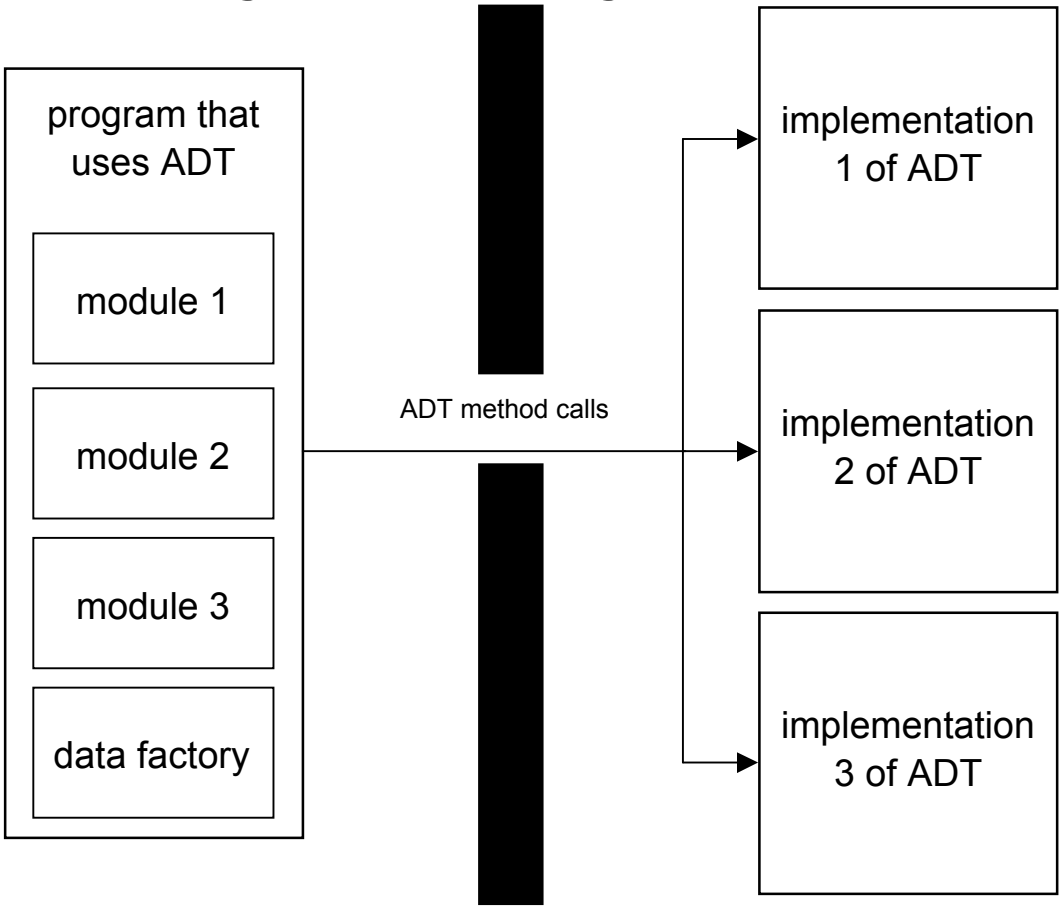
```
public static ListInterface makeList() {
   return new ListArrayBased();
}
```

and collect all such constructing methods (at least one per ADT) in a "data factory" class.

Then, wherever needed in an application:

```
ListInterface s = DataFactory.makeList();
```

Advantages of using ADTs



- Choice of which implementation to use is isolated within data factory.
- System recognizes actual types of objects and connects method calls to correct classes implicitly.
- result: increased modularity
  - software understandable and reusable
  - supports programming in teams or incrementally
  - separation protects proprietary code

Software testing

"A man is his own easiest dupe, for what he wishes to be true he generally believes to be true." Demosthenes

- part of initial development effort *and* integral part of the maintenance cycle
- provides *some* evidence that software meets its specifications
  - Must be systematic, not haphazard
  - Must be well-documented
  - Must be repeatable
- *Unit testing*: purpose is to show that components meet their specifications
- *System testing*: … integrated whole meets its specifications

To be most effective, software testers must adopt the attitude that their goal is to show that the software does *not* meet its specifications.

"Whatever is only almost true is quite false, and among the most dangerous of errors, because being so near the truth, it is the more likely to lead astray." Henry Ward Beecher

Black-box testing

- Design test cases based on specifications
- Recall: pre- and postconditions define a contract.
  - test input should meet preconditions
  - test runs should compare output to expected results implied by postconditions
- "typical" input: values of data expected to be common in practice
- *boundary values*: data that makes the precondition(s) "barely" true

For example,

```
public void removeMe(Object[] array);
// pre:  array not null
// post: removes first occurrence of this, if any, closing
//       gap and setting the last entry to null
```

test cases involving `x.removeMe(a)`:

`[]`	`a.length` == 0
`[x]`	`x` is the only member
`[null]`	`null` is the only member
`[y,…,x,…,z]`	`x` is in the middle
`[x,…,y]`	`x` is at the start
`[y,…,x]`	`x` is at the end
`[y,…,z]`	`x` is not in the array
`[y,…,x,…,x,…,z]`	`x` repeats

White-box testing

- Design test cases based on the structure of the code.
- Execute every line of code.

– *Branch testing*: tests for each alternative
– *Loop testing*: tests to iterate
  * 0 times
  * exactly once
  * several times
  * as often as possible
– *Exit testing*: tests to cause each condition for loop or method exit
– *Exception testing*: test of exception handling

Continuing example:

```
public void removeMe(Object[] array) {
// pre:  array not null
// post: removes first occurrence of this, if any, closing
//       gap and setting the last entry to null
  int i;
  for (i = 0; i < array.length; i++) {
     if (array[i] == this) break;
  }
  if (i == array.length) return;
  while (i < array.length - 1) {
     array[i] = array[i + 1];
     i++;
  }
  array[i] = null;
}
```

Systematic testing

Test plan: collection of tests to perform; for each test:

- describe condition(s) being tested
- input data
- expected results

Test log: record of the results of running tests

- pairing test from test plan to output produced

Test harness: program that reads test plan from a data file, executes it, and writes test log to output file

Regression testing: testing modified code with identical inputs used to test that code previously

- Test each class implementing an ADT
  - include `main` method to invoke the test harness with suitable test plan
  - test each constructor and each method
  - rerun after each code change to ensure changes eliminated errors without introducing new ones
- Test application (system)
  - black-box: use system specifications only
  - white-box: use knowledge of components and their interactions
- Prepare system test report
  - test log for system as a whole
  - explain any deviations from specifications

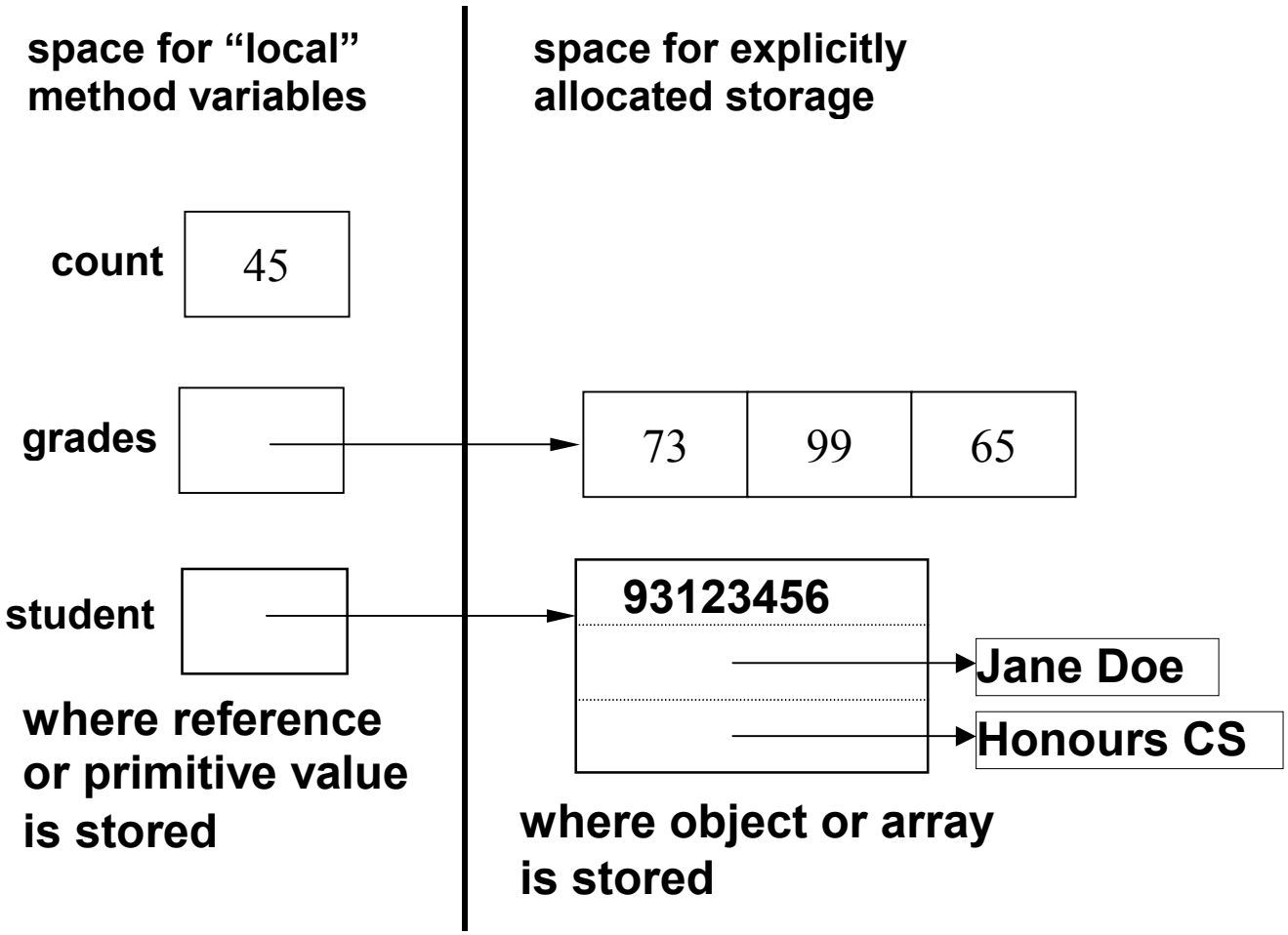
Implementation of List

- might choose a data structure that will allow random access to elements at a given index
- a partially filled array of a fixed size might be a reasonable choice if an upper bound on the number of items to be inserted into the List is known
- What if we do not know an upper bound on the size?
  - use a partially filled array which is resized when full is a data structure that will grow and shrink gracefully as objects are inserted into and removed from a List
  - this is precisely what a `Vector` or `ArrayList` is (`java.util`)
  - Carrano & Prichard only discuss this briefly on pp. 228-229
- How can data structures grow dynamically?

Dynamic storage in Java

- Space to hold the value of a variable "local" to a method is allocated automatically.
- Space to hold the value of an object or array is explicitly allocated and initialized upon construction.

– in response to use of **`new`** or array initializer

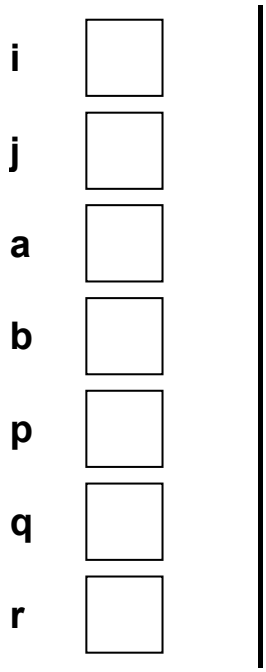


Using references in Java

Assume `Student` is a class that includes:

```
int id;   String name;    String major;
public int silly(int k, int[] c, Student s) {
  k++;    c[1]++; this.setID(k);
  if (s != null) s.setID(c[0]);
  if (!s.equals(this)) s = this;
  return(k);
}
public void setID(int newID) {id = newID;}
```

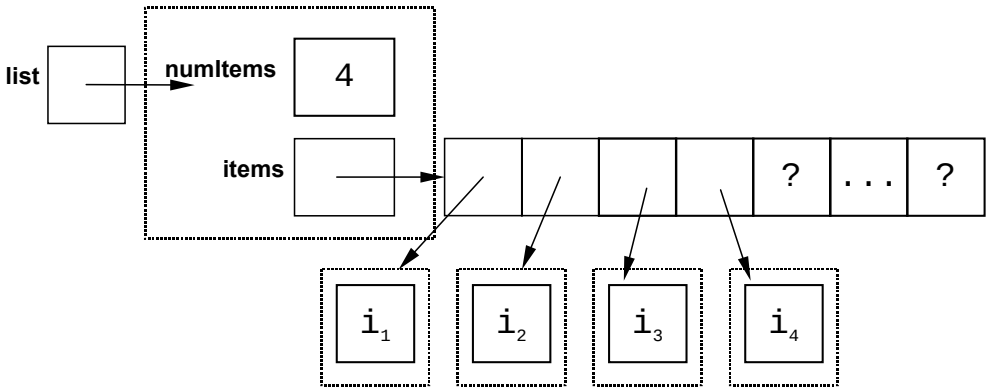
Line #	Code
1,2,3	`int i, j;  int[] a, b;  Student p, q, r;`
4,5,6	`i = 5; j = 6; j = i;`
7,8,9,10	`a = new int[3]; b = a; b[0] = 5; a[1]++;`
11	`p = new Student(8, "Jo", "CS");`
12	`q = new Student(9, "Lee", "C&O");`
13	`r = p;`
14	`j = p.silly(i, a, q);`



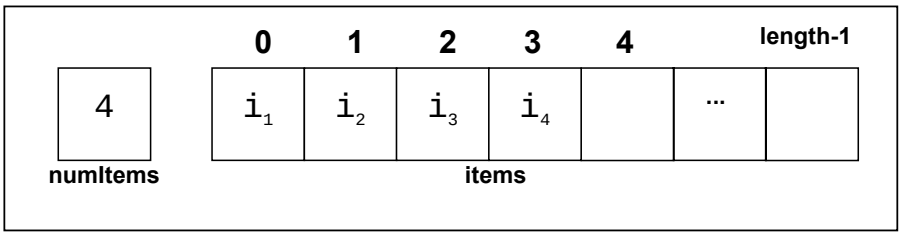
Implementation of List as a partially-filled array

- Uses two instance variables:

```
int numItems;        // number of elements in List
Object[] items;      // storage for List's elements
```



or more simply diagrammed as



- Straightforward implementations for `size()`, `isEmpty()`, and `get(index)` using

```
private int translate(int position) {
// pre: 1 ≤ position ≤ numItems+1
// post: position - 1
   return position - 1;
}
```

Updating an array-based List

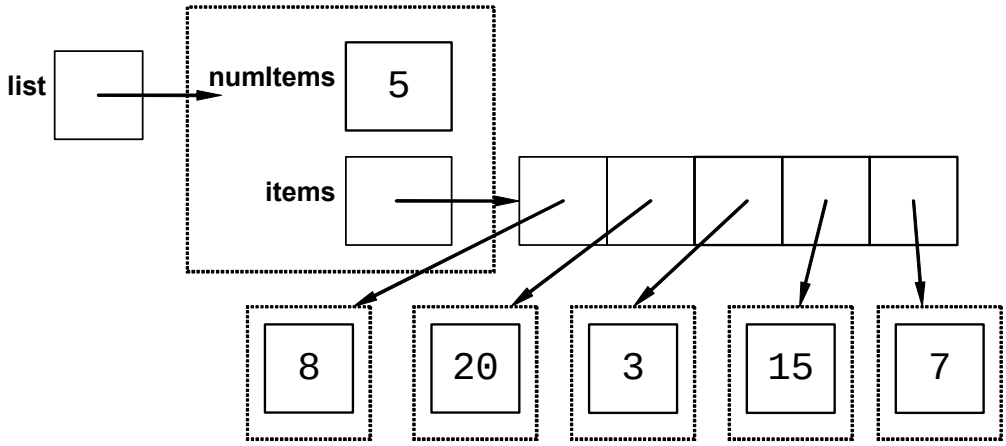
- simple loop needed to close gap

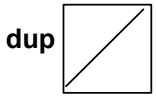
```
public Object remove(int index) {
// pre: 1 <= index <= size()
// post: removes and returns the item at position index in the list
//         and other items are renumbered accordingly

  Object toRemove = items[translate(index)];

  for(int pos = index; pos < size(); pos++) {
    items[translate(pos)] =
                    items[translate(pos+1)];
  }

  items[translate(numItems)] = null;
  numItems--;

  return toRemove;
}
```

- `removeAll()` simply requires setting items to a new empty array

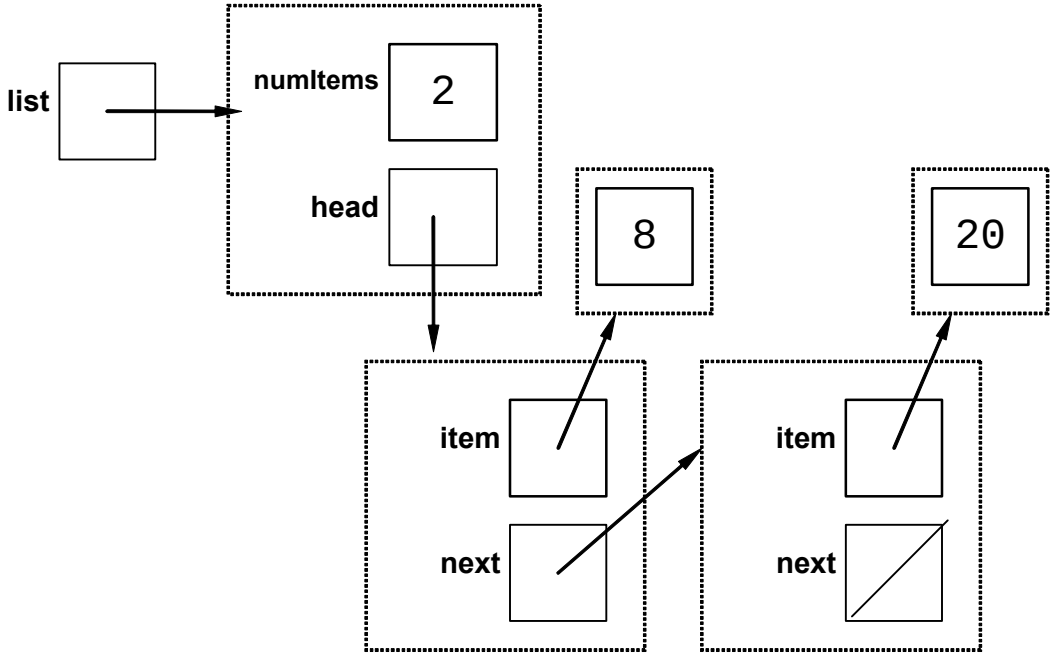
Adding elements to an array-based List

- add(index, item) might require array to be enlarged
- replacing the array:
  - allocate a larger array (but how much larger?)
  - copy all the elements (object references) to the new array
  - replace (the reference to) the old array by (a reference to) the new one

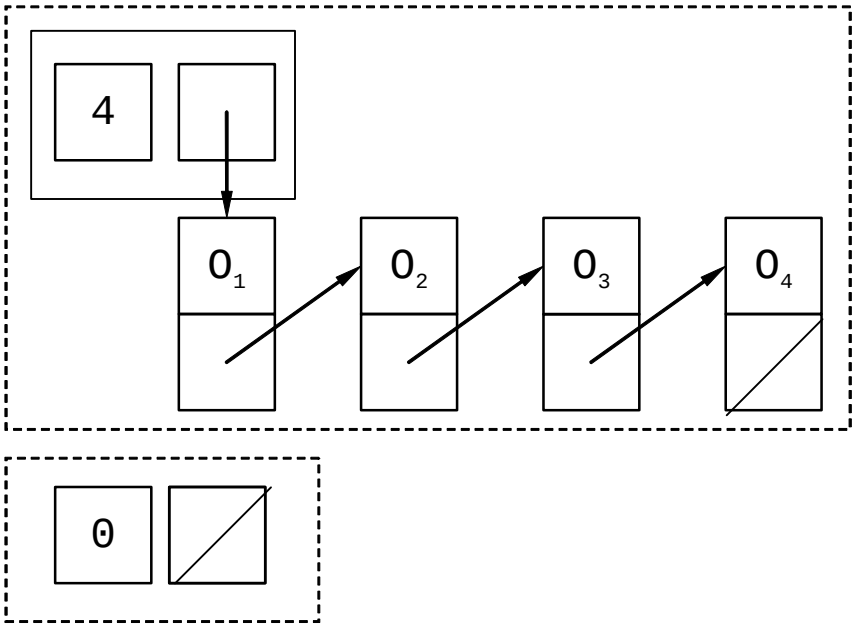




Alternative implementation of List: Singly-Linked List



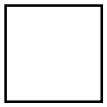
- Expands and contracts as needed
- Diagrammatic representation



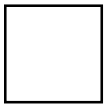
Manipulating linked list elements

Node methods: see Carrano & Prichard pp. 231-232

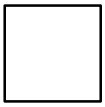
```
Node myNode = new Node(new Integer(5));
myNode.setItem(new Integer(7));
myNode.setNext(new Node(new Integer(9)));
myNode.getNext().setItem(new Integer(2));
Node temp = myNode.getNext();
Node newNode = new Node(myNode.getItem());
temp.setNext(newNode);
```



myNode



temp



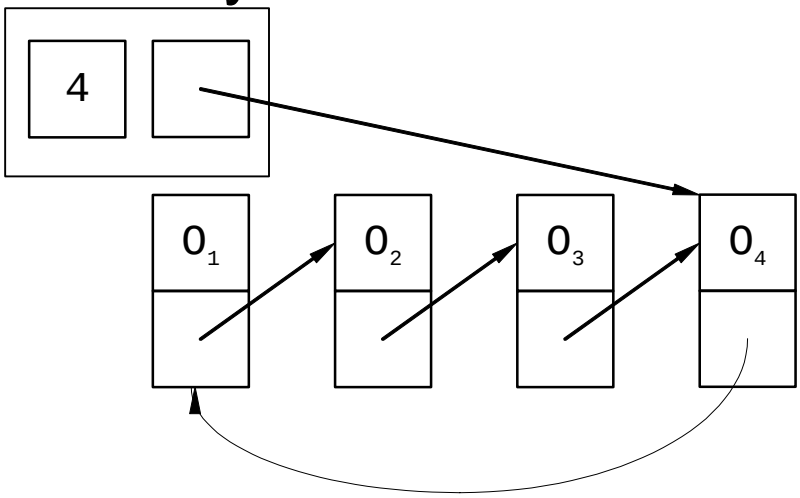
newNode

Implementation of List as a Singly-Linked List

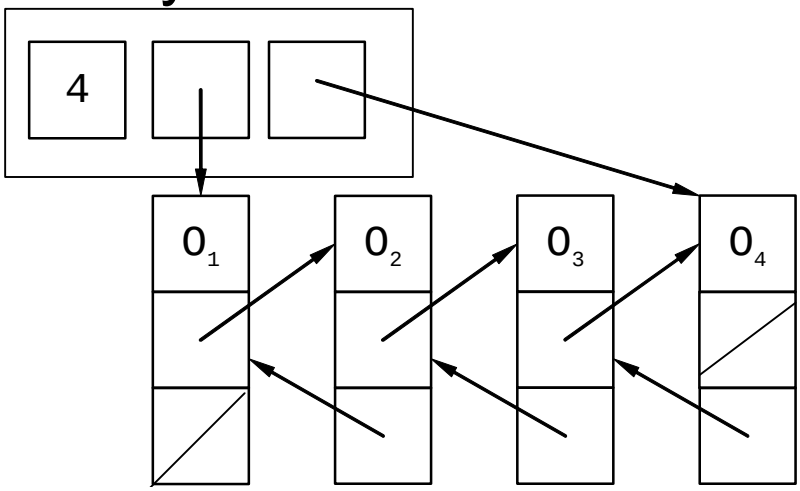
```
public class ListReferenceBased implements
                                ListInterface {
   private Node head;
   private int numItems;

   private Node find(int index) {
   // pre: 1 ≤ index ≤ numItems
   // post: Returns a reference to the node at position index
      Node curr = head;
      for (int skip = 1; skip < index; skip++) {
         curr = curr.getNext();
      }
      return curr;
   }

   public Object remove(int index) {
   // pre: 1 <= index <= size()
   // post: removes and returns the item at position index in the
   //       list and other items are renumbered accordingly
      Object toRemove;
      if (index == 1) {
         toRemove = head;
         head = head.getNext(); }
      else {
         Node prev = find(index - 1);
         Node curr = prev.getNext();
         toRemove = curr;
         prev.setNext(curr.getNext());
      }
      numItems--;
      return toRemove;
   }
   …
}
```

Variations on linked lists

- circularly linked lists



- doubly-linked lists



- other variants: no counters; more links; “sentinel” element at end

ADT Deque

- alternative linear collection
- functionality:
  - add only at either of the two ends
  - read at ends
  - test for membership via **`contains`**
  - remove at ends
- implementation alternatives:
  - as linked list (singly-linked, doubly-linked, circular)
  - as **`Vector`** or **`ArrayList`** (partially-filled array)
  - as ADT List
- efficiency trade-offs?

Measuring efficiency

- some possible measures:
  - the amount of time it takes to code
  - the amount of memory the *code* uses
  - the amount of time it takes to run
  - the amount of memory the *data* uses
- costs to prepare and to maintain software critical aspects of software engineering
- principal measures for this course: efficiency of software execution
- How can we compare several possible choices of algorithm or several possible implementations of an interface?
  - run the code (stopwatch approach)
    - influenced by hardware
    - influenced by compiler and system software
    - influenced by simultaneous users' activity
    - influenced by selection of data
  - analyze the code (or pseudo-code)
    - still influenced by choice of data

Comparing implementations

- first approximation: abstraction that ignores low level details
  - calculate number of method invocations for critical methods used
  - typically interested in methods that read object's values or change object's values
  - for collections, often interested in
    - number of data values encountered (e.g., how many calls to `getNext()`)
    - number of data values moved or modified (e.g., how many calls to `setNext()` or `setItem()`)
- concerned with how algorithm behaves across all possible inputs
  - worst-case analysis (worst input)
  - best-case analysis (best input)
  - average-case analysis (depends on distributions)
- usually analysis done with respect to data of a given, but unknown, size
  - *e.g.*, Considering all collections having N elements (for any fixed value of N), how many times is operation P executed in the worst case?

Code Analysis: Example

Consider possible implementation of a remove by value method for array-based representation of List

```
public Object removeByValue(Object obj){
// pre: obj is non-null
// post: removes and returns the first object equal to obj from the array;
//       returns null if no objects are equal to obj
    int index;
    for (index = 0; index < numItems; index++) {
        if (obj.equals(items[index])) break;
    }
    if (index == numItems) return null;
    Object val = items[index];
    numItems--;
    while (index < numItems) {
        items[index] = items[index+1];
        index++;
    }
    items[numItems] = null; // free reference
    return val;
}
```

- How many calls will be made to equals (comparing two data values)?
  - best case? worst case?
  - what would influence the average case?
- How many elements are moved to a new location in the array?
  - best case? worst case?

ADT Stack

- A linear collection, (s_1, s_2, ..., s_n), of elements accessed from one end only
  - top: s_n
- Sometimes called a LIFO structure (last-in first-out)
- Operations:
  - boolean isEmpty()
  - void popAll()
  - void push(Object newItem)
  - Object pop()
  - Object peek()
  - plus an operation to create an initially empty stack (e.g., `DataFactory.makeStack()`)
- Applications:
  - processing nested elements (e.g., subroutine flow control)
  - reversing element ordering

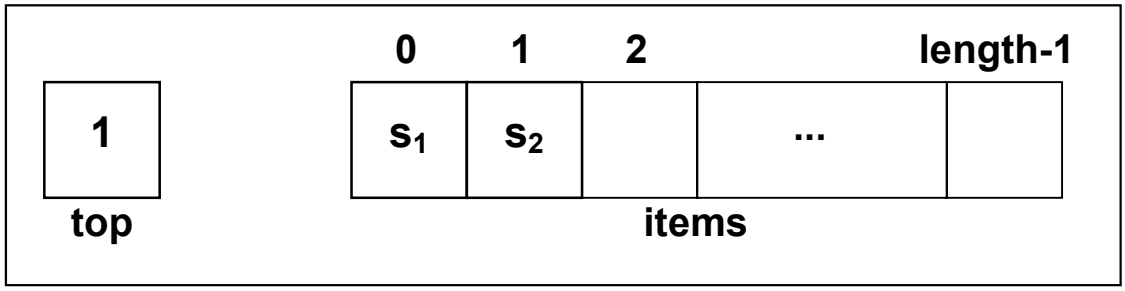
Using a Stack to Check that Parentheses are Balanced

- a(b()cd(e))(f)g is OK, but not (h(i)k or l(m)) or n)(

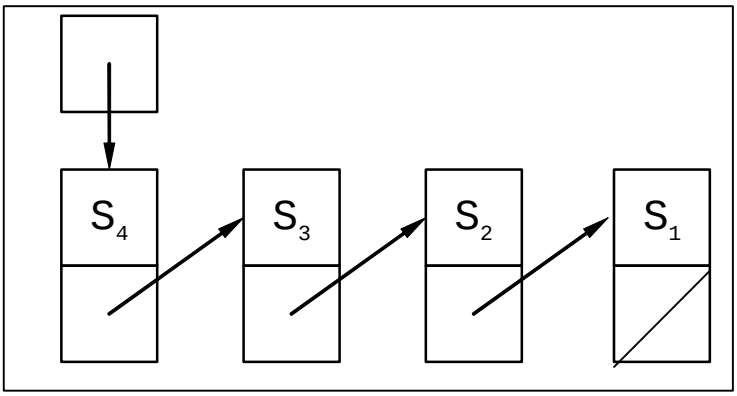
```
class BalanceChecker {
   StackInterface s =
                    DataFactory.makeStack();
   ...
   public boolean check(String in) {
      s.popAll();
      for (int i=0; i < in.length(); i++) {
         if (in.charAt(i) == '(')
            s.push(new Integer(i));
         else if (in.charAt(i) == ')') {
            if (s.isEmpty())
                return false;
            Integer open =(Integer)s.pop();
            // open.intValue() position has matching '('
         }
      }
      return (s.isEmpty());
   }
}
```

- Why does this algorithm work?

Array-Based Implementation of Stack



Linked-List-Based Implementation of Stack



Implementing Method Calls Using a Stack

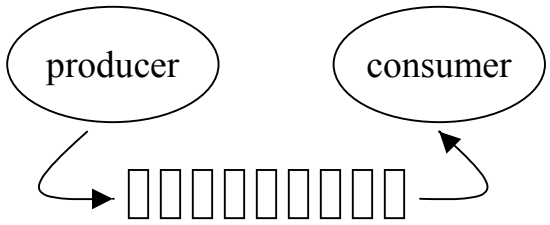
- An array-based stack is normally used within generated code
- All information needed to support a particular method call is kept in an *activation record*
  - space for parameter values
  - space for local variables
  - space for location to which control is returned
- During execution, maintain the stack of activation records
  - When method called: activation record created, initialized, and pushed onto the stack
  - When a method finishes, its activation record is popped

ADT Queue

- A linear collection, (q_1, q_2, ..., q_n), of elements accessed in sequence
  - front: q_1; rear: q_n
- Sometimes called a FIFO structure (first-in first-out)
- Operations:
  - createQueue()
  - boolean isEmpty()
  - void dequeueAll()
  - void enqueue(Object newItem)
  - Object dequeue()
  - Object peek()
- Applications:
  - communications channel
  - items waiting to be serviced

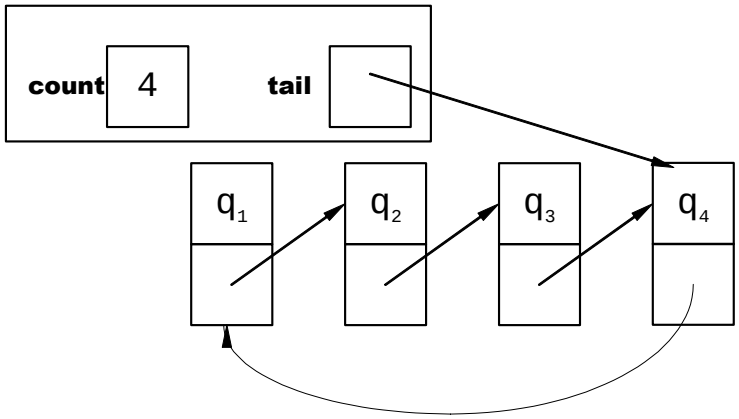
The Use of Queues in Producer-Consumer Situations

- General situation: one software package is producing data and another is consuming the same data.



- Rate of consumption or production often data-dependent.
- Producer can use *enqueue* to insert and Consumer can use *dequeue* to take data in insertion order.

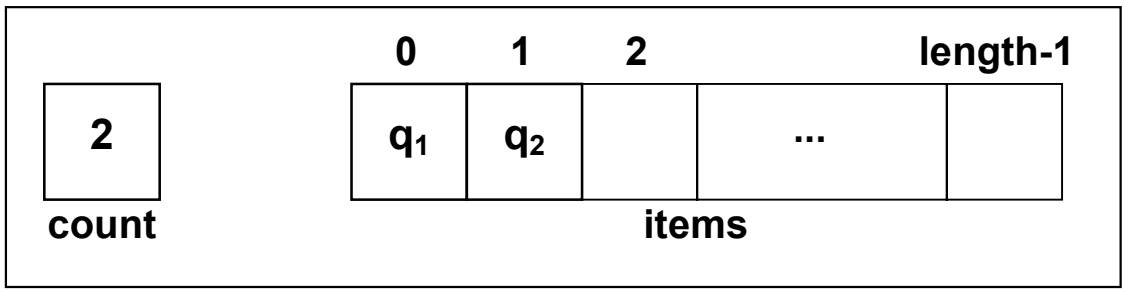
Queue Using a Circular Linked List




```
public Object dequeue() {
// pre: queue is not empty
// post: the head of the queue is removed and returned
   Object val = tail.getNext().getItem();
   count--;
   if (count == 0) tail = null;
   else tail.setNext(
            tail.getNext().getNext());
   return val;
}
public void enqueue(Object val) {
// post: val is added to the tail of the queue
   Node temp = new Node(val);
   if (count == 0) {
      tail = temp;
      tail.setNext(tail);
   }
   else {
      temp.setNext(tail.getNext());
      tail.setNext(temp);
      tail = temp;
   }
   count++;
}
```

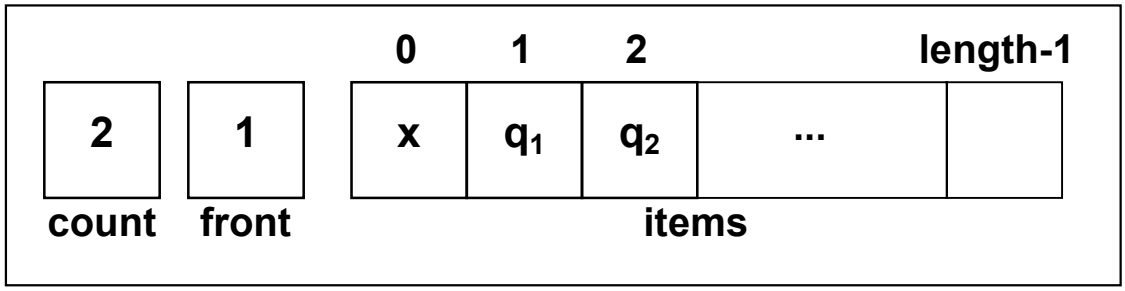
Queue Using a "Circular" Array

- Naïve use of array

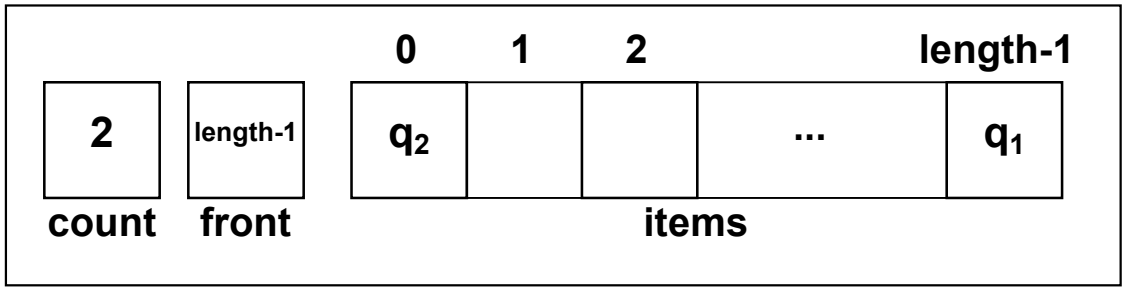


– dequeue ⇒ shuffling values to start of array

- Alternative: let queue drift down array



– when reaching end of array, wrap around to start



- Implementation details...

see Carrano & Prichard, Chapter 8

Ariane 5

On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded.

[*Ariane 5 Flight 501 Failure*, Report by the Inquiry Board, July 1996]

What happened?

- "At 36.7 seconds after H0 (approx. 30 seconds after lift-off) the computer within the back-up inertial reference system, which was working on stand-by for guidance and attitude control, became inoperative. This was caused by an internal variable related to the horizontal velocity of the launcher exceeding a limit which existed in the software of this computer.
- "Approx. 0.05 seconds later the active inertial reference system, identical to the back-up system in hardware and software, failed for the same reason. Since the back-up inertial system was already inoperative, correct guidance and attitude information could no longer be obtained and loss of the mission was inevitable.
- "As a result of its failure, the active inertial reference system transmitted essentially diagnostic information to the launcher's main computer, where it was interpreted as flight data and used for flight control calculations.
- "On the basis of those calculations the main computer commanded the booster nozzles, and somewhat later the main engine nozzle also, to make a large correction for an attitude deviation that had not occurred."

Ariane 5 (cont'd)

The inertial reference system of Ariane 5 is essentially common to a system which is presently flying on Ariane 4.

So, how did the failure happen?

- "The part of the software which caused the interruption in the inertial system computers is used before launch to align the inertial reference system and, in Ariane 4, also to enable a rapid realignment of the system in case of a late hold in the countdown. This realignment function, *which does not serve any purpose on Ariane 5*, was nevertheless retained for commonality reasons and allowed, as in Ariane 4, to operate for approx. 40 seconds after lift-off.
- "In Ariane 4 flights using the same type of inertial reference system there has been no such failure because the trajectory during the first 40 seconds of flight is such that the particular variable related to horizontal velocity cannot reach, with an adequate operational margin, a value beyond the limit present in the software.
- "Ariane 5 has a high initial acceleration and a trajectory which leads to a build-up of horizontal velocity which is five times more rapid than for Ariane 4. The higher horizontal velocity of Ariane 5 generated, within the 40-second timeframe, the excessive value which caused the inertial system computers to cease operation."

Lessons learned:

R5 *… Identify all implicit assumptions made by the code and its justification documents on the values of quantities provided by the equipment. Check these assumptions against the restrictions on use of the equipment….*

R11 *Review the test coverage of existing equipment and extend it where it is deemed necessary.*

R12 *Give the justification documents the same attention as code. Improve the technique for keeping code and its justifications consistent….*

Software in the real world

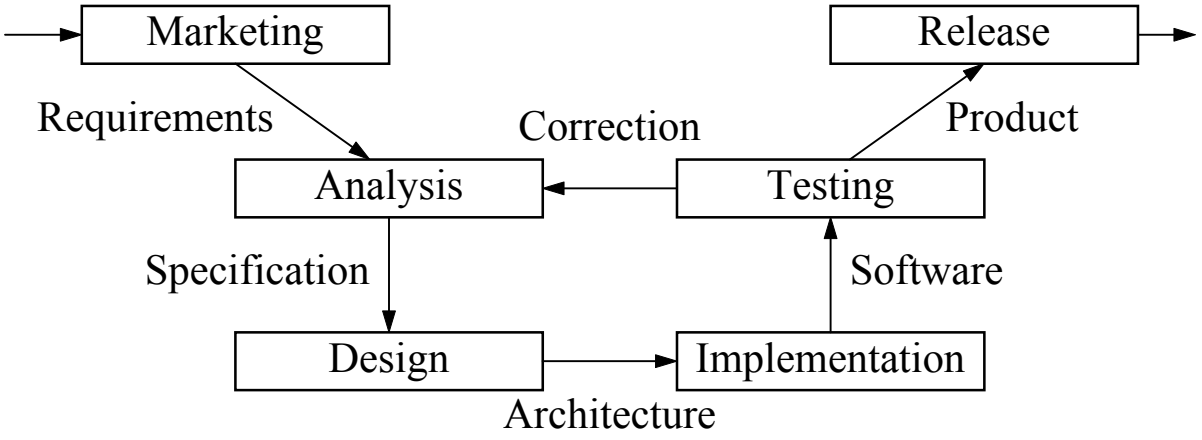
- Specifications change
- People change
- Support systems change
- Intended applications change

Programs must survive these changes.

- well-designed programs are *adaptable*
- well-designed components can be *reused*

Guideline: Design and document software components as you would have others design and document them for you.

The “Software Life Cycle”



Iterative model of software evolution

[Hume, West, Holt, and Barnard]

- Note the cycle!
  - more accurate than including a maintenance box as part of the traditional “waterfall model”
  - for correcting software
  - for responding to new marketing information
- Throughout the whole life cycle, documentation is critical: to capture rationale and communicate intent.

Recursive Definitions

- common in mathematics

  Recursive definition defines an object in terms of smaller objects of the same type.

- includes base (degenerate) cases and recursive cases

- Example 1: factorial function

```
n! = 1        if n=0 {base case}
n! = n(n-1)!  if n>0 {recursive case}
```

- Example 2: Fibonacci numbers

$f_0 = f_1 = 1$ {base cases}

$f_n = f_{n-1} + f_{n-2}$ if $n \geq 2$ {recursive case}

- Example 3: balanced strings

– base case:

  A string containing no parentheses is balanced.

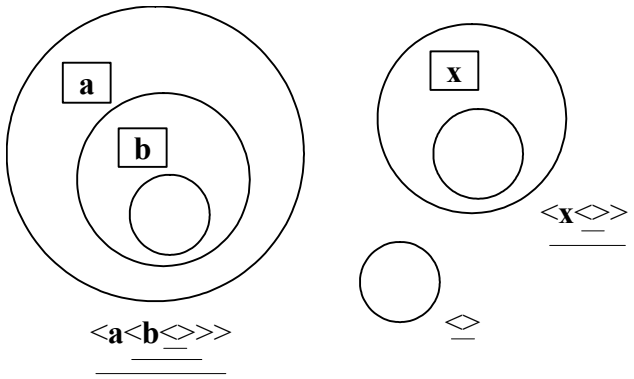
– recursive cases:

  `(x)` is balanced if `x` is a balanced string.

  `xy` is balanced if `x` and `y` are balanced strings.

Recursive Structures: Sublist ADT

- As an alternative to List ADT (based on nodes) define recursive linear structure:



- A *sublist* is a finite collection of nodes that is
  - empty, or
  - partitioned into 2 sub-collections: a designated node, called the *root*, together with a sublist, designated as the *remainder*
- Additional recursive definitions

  sublist L *contains* object x if L is not empty *and*
  - the head element of L is x or
  - the remainder of L contains x

  sublist L1 *encompasses* sublist L2 if
  - L1 *is* L2 (i.e., they both name the same thing), or
  - L1 is the pair <E, L1′> *and* L1′ encompasses L2.
- Essentially the basis of LISP (1958)

Defining Sublist ADT

```
public interface SubListInterface {

  public SubListInterface detachRemainder();
  // pre:  this is not empty.
  // post: returns the remainder of this and sets the remainder to be the empty
  //             sublist.

  public Object getRootItem();
  // pre:  this is not empty.
  // post: returns value associated with the root.

  public boolean isEmpty();
  // post: returns true iff this is empty.

  public void makeEmpty();
  // post: this is empty.

  public SubListInterface remainder();
  // pre:  this is not empty
  // post: returns the remainder of this

  public void setRemainder(SubListInterface newRem);
  // pre:  this is not empty, the remainder of this is empty, and newRem is
  //             not null.
  // post: attaches value of newRem as the remainder of this

  public void setRootItem(Object newItem);
  // post: Sets the value associated with the root to be newItem
  //              if this is not empty; otherwise sets this to consist of a
  //              root node only, with root item set to newItem.

}
```

Dealing with Sublist Remainders

- `attachRemainder()` provides concatenation of two sublists
- `detachRemainder()` allows tail parts of lists to be split off to become independent sublist
- `remainder()` must be used with care!

– useful for accessing components of SubList without deconstruction and subsequent reconstruction

– returned object shares its value with *part* of some other object

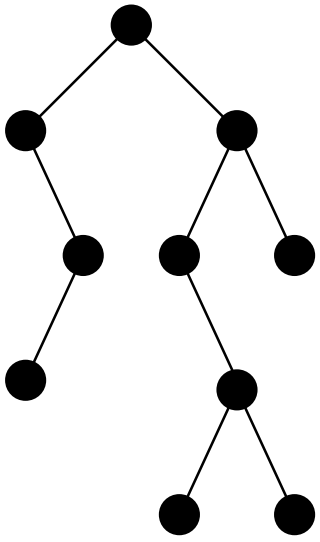
```
SubList s1 = ....
SubList s2 = s1.remainder();
s2.setRootItem(x); // s1 has also been changed!!
```

A Non-Linear Recursive Structure

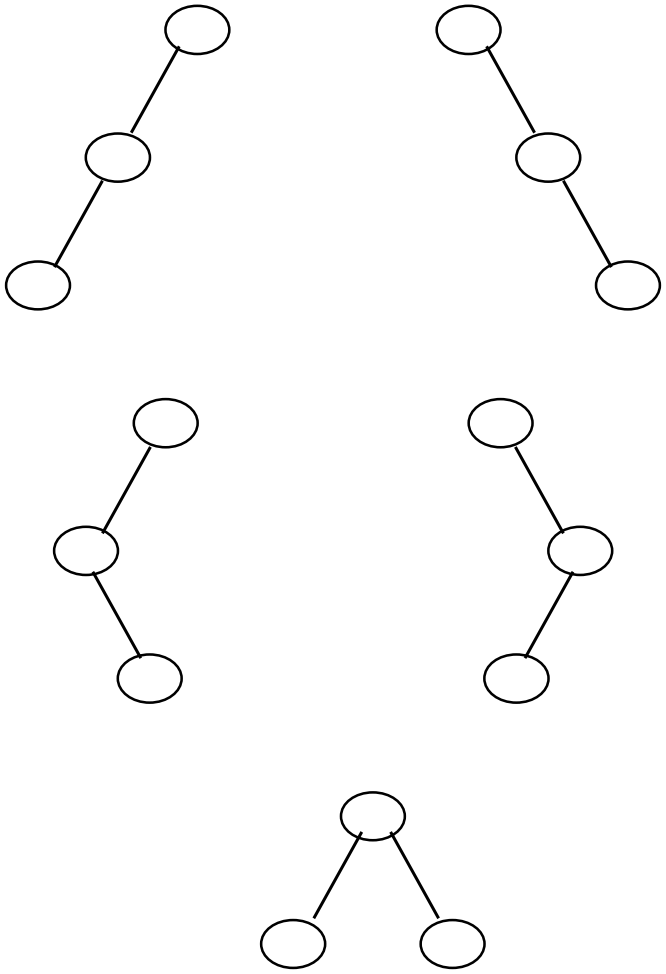
- A *binary tree* is a finite collection of nodes that is
  - empty, or
  - partitioned into 3 sub-collections: a designated node, called the *root*, together with two binary trees, designated as *left* and *right* subtrees

base case, empty tree *recursive case*

- Tree terms for nodes: root and leaf
- Familial terms for nodes: parent, child, sibling, ancestor, descendant



Binary trees on three nodes

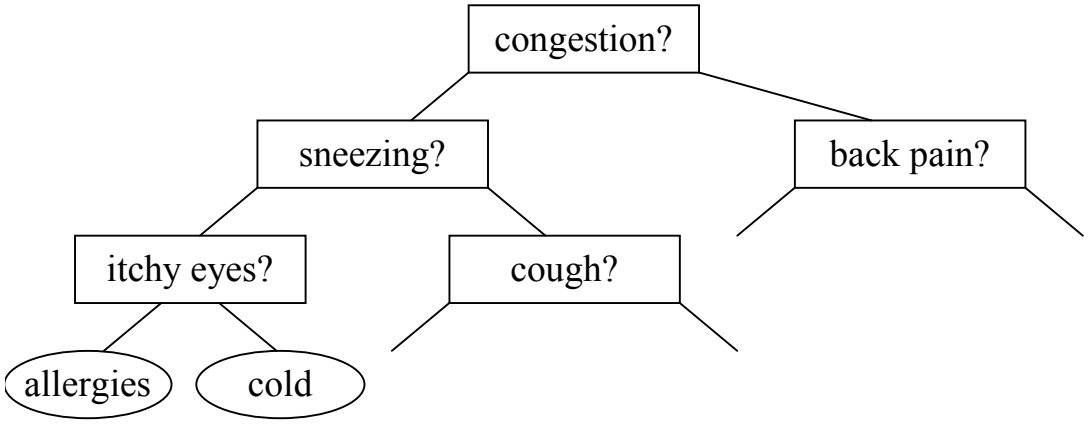


- Note: one-to-one correspondence between nodes in a binary tree and non-empty binary subtrees encompassed by that tree

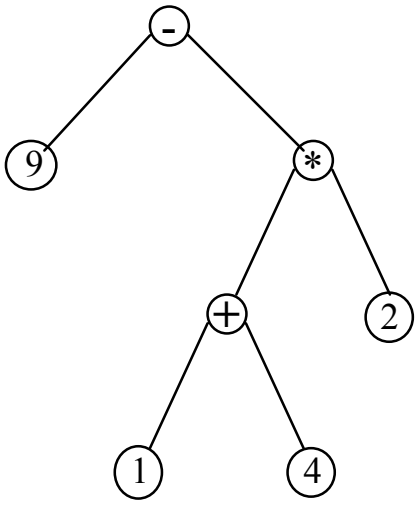
Labelled Binary Trees

- Typically nodes are labelled.
- Examples:

– yes-no decision tree



– expression tree (for binary operators)



- Semantics captured by nesting

Binary Tree ADT

- based on recursive definition of tree

```
public interface BinaryTreeInterface {

    /* methods dealing with empty trees */

    public void makeEmpty();
    // post:   this is empty.

    public boolean isEmpty();
    // post:   Returns true iff this is empty.

    /* methods dealing with items */

    public Object getRootItem();
    // pre:    this is not empty.
    // post:   returns value associated with the root.

    public void setRootItem(Object newItem);
    // post:   Sets the value associated with the root to be newItem
    //         if this is not empty; otherwise sets this to consist of a
    //         root node only, with root item set to newItem.

    public void attachLeft(Object newItem);
    // pre:    this is not empty.
    // post:   No change if the left subtree is non-empty; otherwise, sets
    //         the left subtree to be a leaf node with associated value set
    //         to newItem.

    public void attachRight(Object newItem);
    // pre:    this is not empty.
    // post:   No change if the right subtree is non-empty; otherwise, sets
    //         the right subtree to be a leaf node with associated value set
    //         to newItem.
```

Binary Tree ADT (cont'd)

```
    /* methods dealing with subtrees */

    public BinaryTreeInterface leftSubtree();
    // pre:    this is not empty
    // post:   returns the left subtree of this

    public void setLeftSubtree(
                        BinaryTreeInterface leftTree);
    // pre:    this is not empty, the left subtree of this is empty, and leftTree is
    //         not null.
    // post:   attaches leftTree as the left subtree of this.

    public BinaryTreeInterface detachLeftSubtree();
    // pre:    this is not empty
    // post:   returns the left subtree of this and sets the left subtree to be the
    //         empty tree.

    public BinaryTreeInterface rightSubtree();
    // pre:    this is not empty
    // post:   returns the right subtree of this

    public void setRightSubtree(
                        BinaryTreeInterface rightTree);
    // pre:    this is not empty, the right subtree of this is empty, and rightTree
    //         is not null.
    // post:   attaches rightTree as the right subtree of this

    public BinaryTreeInterface detachRightSubtree();
    // pre:    this is not empty
    // post:   returns the right subtree of this and sets the right subtree to be the
    //         empty tree.
}
```

Recursive Programs

- Solution defined in terms of solutions for smaller problems of the same type

```
int solve (int n) {...
   value = solve(n-1) + solve(n/2);
...}
```

- One or more base cases defined

```
... if (n < 10) value = 1; ...
```

- Some base case is always reached eventually.

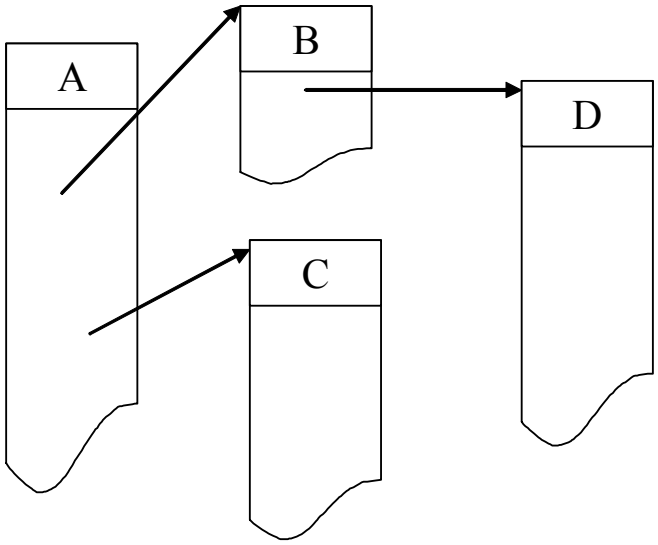
Example:

```
static public int fib(int n) {
// pre: n ≥ 0
// post: returns the nth Fibonacci number

   if (n < 2) return 1;
   else return fib(n-1) + fib(n-2);
}
```

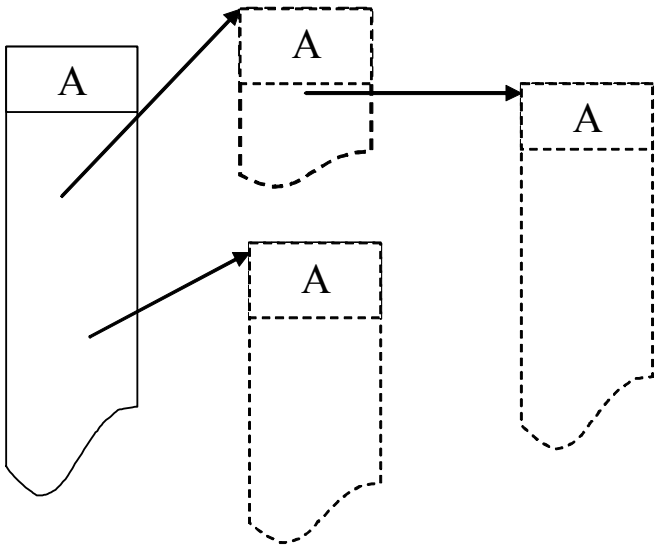
- N.B., structure of code typically parallels structure of definition

Tracing Recursive Programs

- recall: stack of activation records
  - When method called: activation record created, initialized, and pushed onto the stack
  - When a method finishes, its activation record is popped



- same mechanism for recursive programs



Recursive Sublist Programs

- Some additional methods that might be included in the **`SubList`** class

1. calculating the size of a sublist

```
public int size() {
// post:  Returns number of elements in this.
   if (isEmpty()) return 0;
   return 1 + remainder().size();
}
```

2. searching for an item in a sublist

```
public boolean contains(Object key) {
// pre:   key is not null.
// post:  Returns true iff a sublist has a head value matching key.
   if (isEmpty()) return false;
   if (key.equals(getRootItem())) return true;
   return remainder().contains(key);
}
```

3. printing the values of a sublist in reverse order

```
public void printReverse() {
// post:  Prints the elements of this in  reverse order.
   if (!isEmpty()){
      remainder().printReverse();
      System.out.println(getRootItem());
   }
}
```

Recursive Binary Tree Programs

- Similarly, possible additional methods for the **`BinaryTree`** class

1. calculating the size of a binary tree

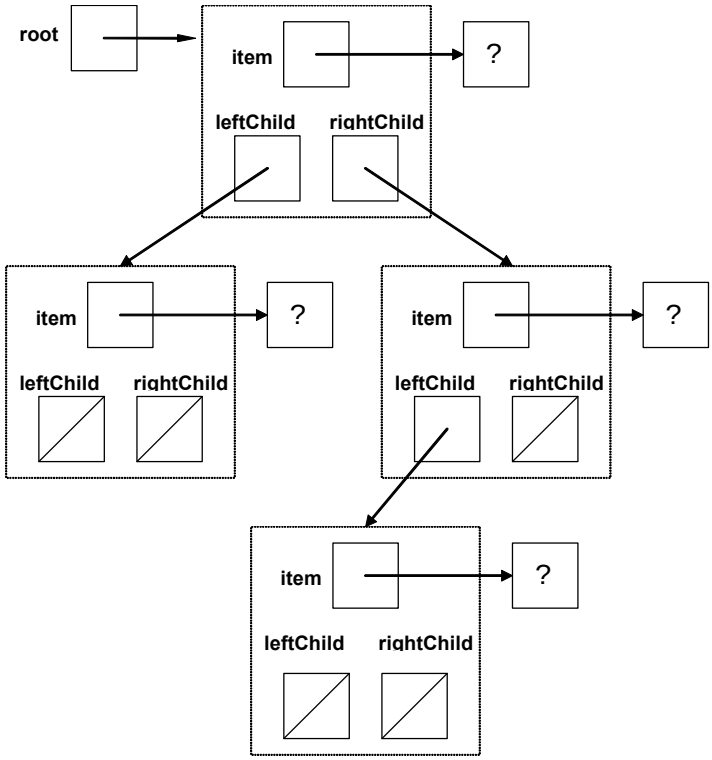
```
public int size() {
// post:  Returns number of elements in this.
   if (isEmpty()) return 0;
   else return 1 + leftSubtree().size()
                 + rightSubtree().size();
}
```

2. following a path to find a subtree

```
public Object follow(Queue path) {
// pre:   path is not null and path objects are all Boolean.
// post:  Returns item at root of subtree identified by path of booleans s.t.
//        true ⇒ follow left branch, false ⇒ right
//        and path is set to empty queue; returns null if
//        path invalid, and valid prefix of path is dequeued.
   if (isEmpty()) return null;
   if (path.isEmpty()) return getRootItem();
   if (((Boolean) path.dequeue()).booleanValue())
      return leftSubtree().follow(path);
   else
      return rightSubtree().follow(path);
}
```

Linked Implementation

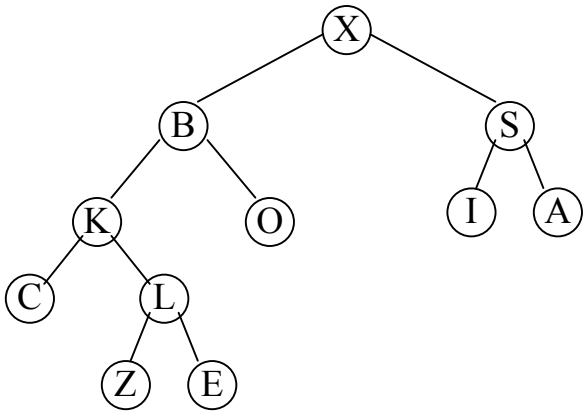
- *Carrano & Prichard* Chapter 11
  - sublist can be implemented using the **`Node`** class, similarly, linked implementation for binary trees can use a **`TreeNode`** class:
    - three data fields (**`item`**, **`leftChild`**, **`rightChild`**)



- **our approach**: use recursive **`BinaryTree`** references in place of **`TreeNode`** references and store parent references
- representation of an empty binary subtree?

Tree Traversals

- want to traverse a tree in some orderly manner
- we visit each node exactly once (e.g., print its contents or determine if it meets certain criteria)
- one option is to visit the nodes level by level:
  - for each level of the tree
    visit each node at that level



  - traversal: X B S K O I A C L Z E
- known as a *breadth-first* traversal.
- can be implemented using a queue

Iterators

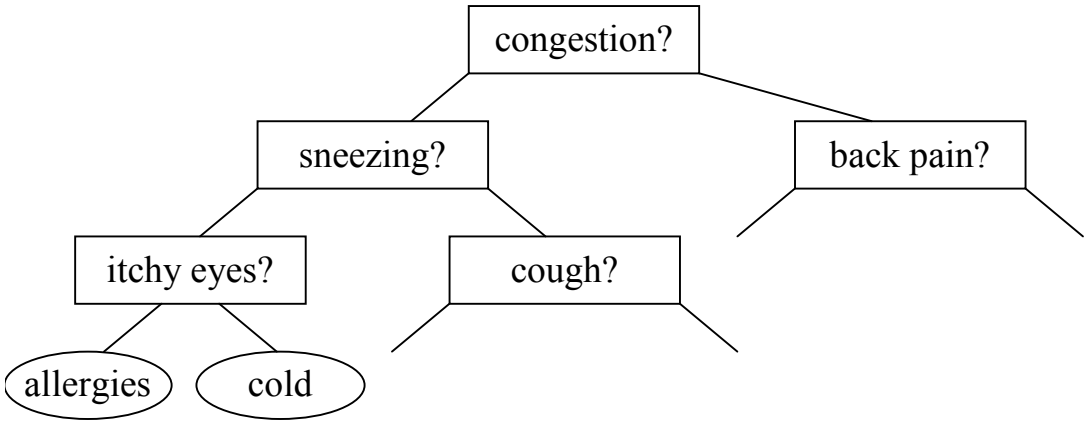
- Auxiliary types that provide access to the elements of a collection
  - each element is “visited” once, and only once
- **`Iterator`** interface from **`java.util`**

```
Iterator i =
someTree.getLevelOrderIterator();
while (i.hasNext()) {
   … i.next(); …
}
```

- May promise a particular order for visiting the elements
- Subclasses may add two more operators:
  - `void reset()`
  - `Object value()`
- `remove()` will delete the last element returned by an iterator
- however, in general, behaviour usually undefined if a collection changes during iteration
- *Note*: several iterators can visit a single structure simultaneously

Depth-First Traversals

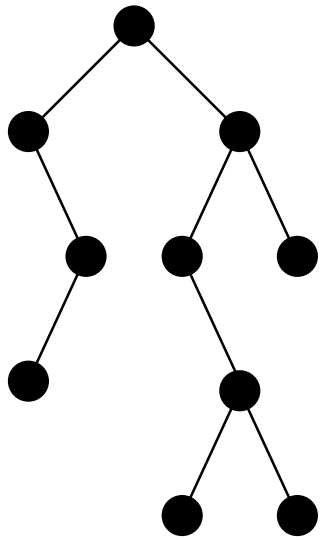
- visit tree's components (root, left subtree, right subtree) in some order
- preorder traversal:
  - visit the root
  - visit the left subtree recursively
  - visit the right subtree recursively



- Preorder traversal yields parents before children, but does not completely characterize a tree's structure.

Implementing Tree Iterators

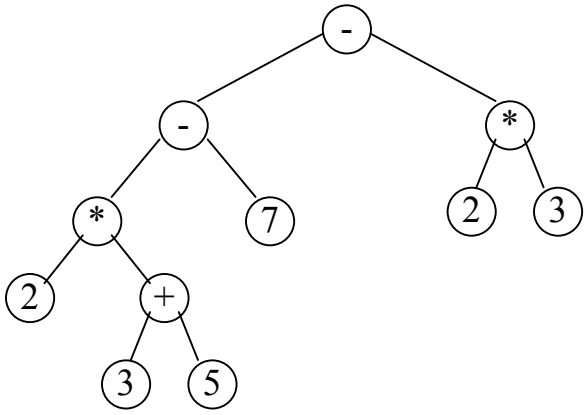
- In the absence of parent pointers, iterator supporting depth-first traversal requires a stack.



- For preorder, stack maintains list of non-empty subtrees remaining to be visited
  - when visiting a node, push right subtree (if non-empty) *and then* left subtree (if non-empty) onto stack
  - peek the top of the stack to find current node
  - pop the stack to find next node to visit
  - iterator completes when stack is empty
- Alternatively, all the work can be done during construction of the iterator using recursion

Inorder Traversal

- ordering:
  - visit the left subtree recursively
  - visit the root
  - visit the right subtree recursively
- inorder traversal of expression tree gives an infix expression



  - traversal: `2 * 3 + 5 - 7 - 2 * 3`
  - but need to insert `(` before visiting any subtree and `)` after visiting any subtree

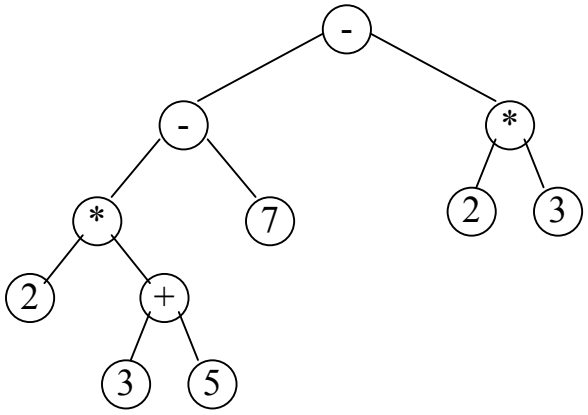
    `(((2)*((3)+(5)))-(7))-((2)*(3))`

    or perhaps `(((2*(3+5))-7)-(2*3))`

- does not characterize which value labels the root

Postorder traversal

- ordering
  - visit the left subtree recursively
  - visit the right subtree recursively
  - visit the root
- Postorder traversal yields children before parents.
- Postorder traversal of an expression tree gives a postfix expression (used for some calculators)



2 3 5 + * 7 – 2 3 * –

  - unambiguous without parentheses !
  - “reverse Polish notation” created in 1920s by logician Jan Lukasiewicz

Properties of Binary Trees

- often expressed recursively (following definition of binary tree)
- depth (or level) of a node:
  - root has level 1
  - otherwise 1+ level of parent
- height of a tree:
  - if the tree is empty, its height is 0
  - otherwise, its height is

    1 + max{height T_L, height T_R }, where T_L and T_R designate left and right subtrees

```
public int height() {
// post:  Returns height of subtree.
  if (isEmpty()) return 0;
  else {
        int leftHt = leftSubtree().height();
        int rightHt = rightSubtree().height();
        return 1 + Math.max(leftHt,rightHt);
  }
}
```

Properties of binary trees (cont'd)

- The height of a non-empty binary tree, T, the maximum depth taken over all of its nodes.

  i.e. max {depth(x) | x is a node of T}

- A binary tree of height *h* has at most 2^h -1 nodes.
- A binary tree with n nodes has height at least $\log_2(n+1)$.
- Define a *full node* to be a node with exactly two children.

  N.B. A tree of height *h* having the maximum number of nodes (2^h-1) is called a *full tree*.

- In any non-empty binary tree, the number of leaves is one more than the number of full nodes.
- In any binary tree, the number of empty subtrees is one more than the number of non-empty subtrees.
- Most of these properties can be proved by mathematical induction.

ADT Table

- Components: associations from keys (from some domain space) to values
  - simple (partial) functions: values are atomic
  - databases: values are records of field-value pairs (often including the key-value pair too)
  - sets: values are empty; ~ *characteristic function*
- Examples:
  - mapping from student ID to name
  - mapping from student ID to student record
  - set of student IDs for students in CS 126 (mapping from ID to “taking CS 126”)
- Intuitive operations:
  - look up given key ≡ *tableRetrieve*
  - insert a new association ≡ *tableInsert*
  - delete association for a given key ≡ *tableDelete*
- Keys are unique; values need not be.
  - at most one value per key (although that value can be a collection for some tables)
  - does not support inverse mapping except through enumeration
- Iterators do not necessarily encounter keys in order.

Simple Representations

- Keep table as a sequence of associations in no particular order, with no keys repeated.
  - using a partially-filled array
  - using a linked list
- efficiency of operations:

  Consider a partially-filled array representation.

  Look at number of comparisons of two keys and number of elements whose locations in the partially-filled array change:
  - e.g., how many comparisons are executed during a call to “retrieve” in the worst case? or in the best case? how many moves are needed during a call to “delete”?

  Similar analyses can be applied to reason about linked list representations.
- implementation convenience: use a single private (non-ADT) method “position” to find the location of association matching a given key, if it exists

Bounding Efficiency

- Running time of a program is a function of the "size" of the problem being solved

– for collections: size = number of elements

Consider solution to a problem of size n

- Running time using one compiler and one machine might be

$$0.33365n^2 - 0.43n + 3.4\ \mu\text{sec}$$

- Another compiler and another machine might take $4.5n^2 + 17n$ msec
- In either case, doubling the size of a large problem means that the solution takes about four times as long to run
- simplify both to "order n-squared", written $O(n^2)$

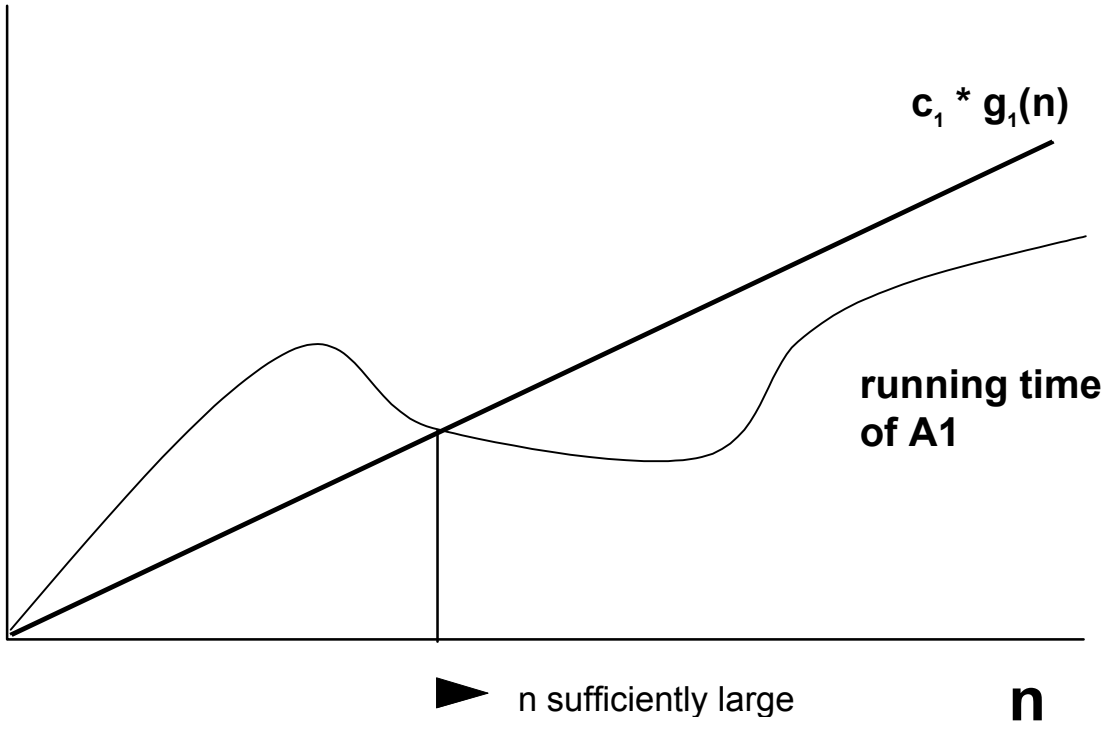
Big-O Notation

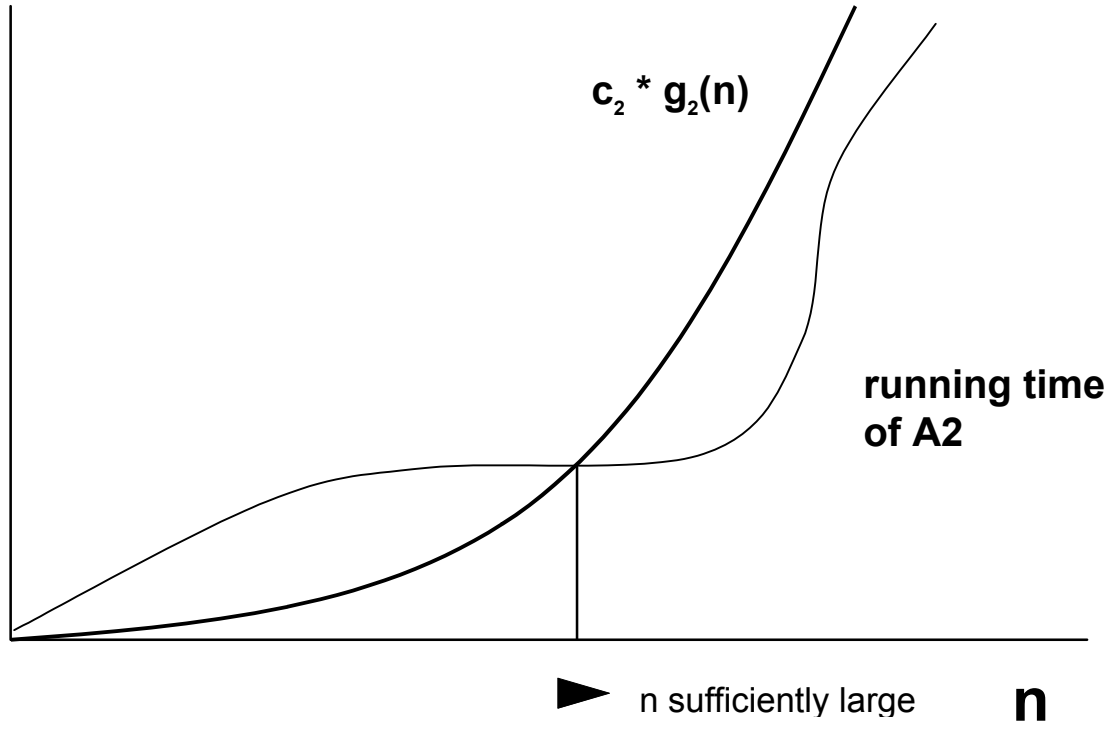
- Intuitively:
  - keep dominant term,
  - remove leading constant,
  - put O(..) around it
- Informally: *f(n) is O(g(n))* if *f(n)* and *g(n)* differ by at most a fixed constant for sufficiently large **n**.
- Formally: *f(n) is O(g(n))* if there exist two positive constants, c and n_0, such that $f(n) \leq c*g(n)$ for all $n \geq n_0$
- Algorithm A is O(**g(n)**) if for any reasonable implementation of the algorithm on any reasonable computer, the time required by A to solve a problem of size **n** is **O(g(n))**.

$(1/2\ n^2 + 1/2\ n)$ is $O(n^2)$

$(13.3\ n + 4\ n^3 + 3/4\ n^2)$ is $O(n^3)$

Big-O Notation (Intuition)





Algorithm A_i has running time $O(g_i(n))$

Use of Big-O Notation

- Common classes of functions
  - constant: O(1)
  - logarithmic: O(log n)
  - linear: O(n)
  - quadratic: $O(n^2)$
  - cubic: $O(n^3)$
  - exponential: $O(2^n)$
- We don't need an exact analysis of every operation; constants can be accumulated
- Examples:
  - popping an element from a stack:
  - removing an element from a list:
  - calculating the size of a binary tree:

Comparing Algorithms

[Jon Bentley, "Programming Pearls: Algorithm Design Techniques," *Comm. ACM 27*, 9 (Sept. 1984) pp. 865-871]

- problem: given an integer array A, find the values i and j which maximize

$$\sum_{k=i}^{j} A[k]$$

	0	1	2	3	4	5	6	7	8	9	10	11
A	25	-5	-12	-9	14	12	-13	5	8	-2	18	-8

- $O(n^3)$ algorithm: try all possible values of i and j
  - How many choices for i?
  - How many choices for j, given i?
  - Cost of figuring out the value for a given i and j?

Alternative Approach

- O(n) algorithm: possible through more clever analysis
  - in single pass over array, keep track of best range so far as well as best starting point for a range ending at current index

- Bentley's implementations:
  - $O(n^3)$ algorithm in finely-tuned FORTRAN on a Cray-1 supercomputer
  - O(n) algorithm in interpreted BASIC on a Radio Shack TRS-80 Model III microcomputer

- estimated running times:
  - $3.0n^3$ nanoseconds on Cray computer
  - 19.5n milliseconds (19500000n nanoseconds) on Radio Shack computer

Bentley's Results

	Cray	TRS-80
n	$3.0n^3$ ns	19.5n ms
10	3 μs	.195 s
100	.003 s	1.95 s
1000		
2500		
10^4		
10^5		
10^6		

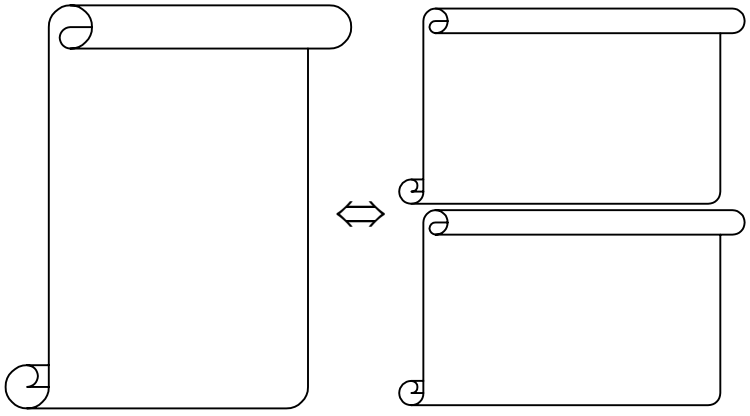
. . .

Faster hardware isn't good enough!

Improvement: Ordered Tables

- Why does a dictionary keep words in alphabetical order?
  - Convenient to find associated information
  - Convenient to determine that some key is not included
- To find a word, we could start at the beginning and scan every word in the dictionary, but we can do better since the dictionary is sorted by key.
- Idea: open a dictionary near the middle, and then determine whether to search in the first or second half

  a sorted dictionary is either empty or the concatenation of two sorted sub-dictionaries of (approx.) half the size for which every word in the first is smaller than every word in the second



– requires that keys be “comparable”

Binary Search

Given a partially filled array, `data`, with counter `numItems`, of comparable objects in ascending order, find the index matching a target key if it is present, otherwise return the index of the slot *where it would be inserted*.

```
int position(Comparable target){
// pre: target is non-null and data values ascending
// post: retuns index s.t. 0 <= index <= numItems, and
//       data[0..index-1] < target, and
//       data[index..numItems-1] >= target

   return search(0, numItems, target);
}

int search(int lo, int hi, Comparable key) {
// pre: 0 ≤ lo ≤ hi ≤ numItems; key not null
// post: returns ideal position of key in data[lo..hi]
   Comparable m;
   int mid = (lo + hi)/2;
   if (lo == hi) return lo;
   else {
      m = (Comparable)data[mid];
      if (m.compareTo(key) < 0)   // m < key
            return search(mid+1, hi, key);
      else  return search(lo, mid, key);
   }
}
```

Efficiency of Binary Search

- *asymptotic analysis*: interested in behaviour for large partially-filled arrays

```
int search(int lo, int hi, Comparable key) {
   int mid = (lo + hi)/2;
   if (lo == hi) return lo;
   else ... // compare key to value of data[mid]
         return search(mid+1, hi, key);
      or return search(lo, mid, key);
}
```

- Each recursive call halves the partially-filled array:

  n $n/2$ $n/4$ $n/8$ $n/16$ …

  after i comparisons, hi-lo = $n/2^i$

  but search ends when hi-lo < 1

  and there is O(1) work between calls

  $\Rightarrow$ time for binary search is O(log n)

- Doubling the size of the partially-filled array requires only one more call to search!

Efficiency of Implementing `Table` Using a Resizable Array

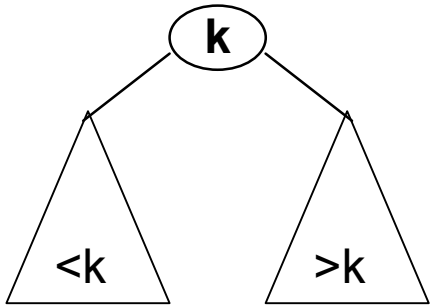
Re-examine worst case:

method	unordered PFA		ordered PFA	
	comps	moves	comps	moves
tableRetrieve				
tableDelete				
tableInsert (with array large enough)				
tableInsert (with array fully occupied)				

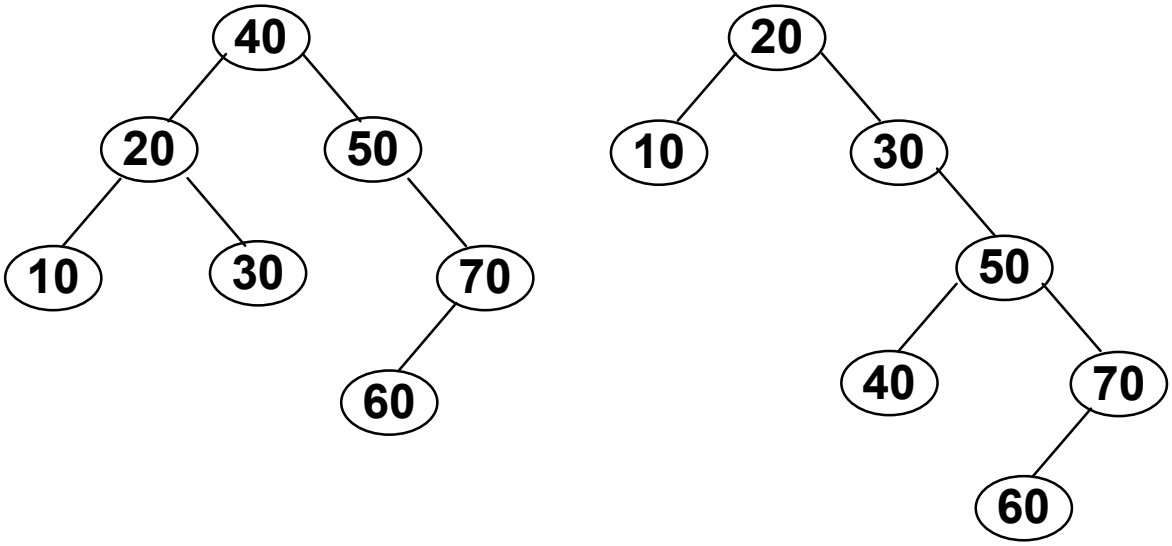
Exercise: Fill in the corresponding chart for implementing `Table` using a linked list in place of a resizable array.

Binary Search Trees

- A binary search tree is an empty binary tree or a labelled binary tree such that:
  - The labels can be compared.
  - The label of the root of a binary search tree is greater than all labels in its left subtree.
  - The label of the root of a binary search tree is less than all labels in its right subtree.
  - The left and right subtrees are also binary search trees.



- *Note*: Binary search tree for a given set not unique



Binary Search Trees as Implementations of Tables

- Labels represent associations
  (or just keys if no associated values)
- Simple code for retrieving an item:

```
public TableBSTBased implements TableInterface{
    BinaryTreeInterface table;

    ...

public KeyedItem tableRetrieve(Comparable SearchKey) {
    TableBSTBased subtree = locate(searchKey,table);
    if (subtree.isEmpty()) return null; // not in tree
    else return (KeyedItem) subtree.getRootItem();
}

private static BinaryTreeInterface locate
    (Comparable searchKey, BinaryTreeInterface tree) {
// post:  returns subtree where the root node contains the sought key,
//        or empty tree if not found
    if (tree.isEmpty()) return tree; // not in tree
    KeyedItem treeItem = (KeyedItem)tree.getRootItem();
    if (searchKey.compareTo(treeItem.getKey()) == 0)
        return tree;   // found it
    else if (searchKey.compareTo(treeItem.getKey()) < 0)
        // if it's there it must be in the left subtree
        return locate(searchKey, tree.leftSubtree());
    // otherwise, if it's there it must be in the right subtree
    else return locate(searchKey, tree.rightSubtree());
}
...
}
```

- Efficiency?

Maintaining a Binary Search Tree

- Inorder traversal encounters values in increasing order.
- Insertion

  Postcondition can be expressed recursively.

  – Empty tree: replaced by a leaf node containing the new value

  – Otherwise: if the new value is less than the root's value, inserted in the left subtree; else inserted in the right subtree

- Deletion

  Postcondition can be expressed by cases.

  1) value not present: no change to the tree
  2) value found in a leaf: that leaf is removed
  3) value found in a node having one empty subtree: that node is removed and that subtree is attached to the parent
  4) value found in a node having two non-empty subtrees: that node's value is changed to the value of it's inorder predecessor (or successor) and that inorder predecessor (or successor) is deleted recursively

- **`tableRetrieve`**, **`tableInsert`**, and **`tableDelete`** use O(*h*) time, where *h* is the height of the tree—in the best case and on average, *h* is O(log n)

Sorting Introduction

- Remember our dictionary that was sorted in alphabetical order: how did the information become sorted?
- The remaining lectures will be devoted to different sorting algorithms and their efficiencies.
- Here are some assumptions we'll use when sorting:

– All the data can fit in memory.

– Data is stored in an array of size n.

– Data is all "comparable":

```
public interface Comparable {
    public int compareTo(Object other);
    // pre:  other is not null
    // post: returns value less than 0 if this is less than other;
    //       returns 0 if this equals other; otherwise returns
    //       value greater than zero
}
```

– Contents of the array are to be modified so that the data is rearranged into non-decreasing order.

Selection Sort

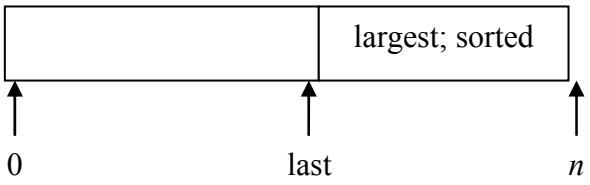
- Idea: repeatedly extract maximal element from among those still unsorted

```
int indexOfLargest(Comparable[] theArray, int size) {
// pre:  0 ≤ size ≤ theArray.length, theArray is non-null and
//       elements of array between 0 and size –1 are non-null
// post: returns index result such that theArray[i] <= theArray[result]
//       for all i in {0,...,size-1}
    int indexSoFar = 0;
    for (int currIndex=1; currIndex<size; currIndex++) {
       // theArray[indexSoFar] >= theArray[0..currIndex-1]

        if (theArray[currIndex].
              compareTo(theArray[indexSoFar]) > 0) {
           indexSoFar = currIndex;
           }
        }
    return indexSoFar;
}

void selectionSort(Comparable[] theArray, int n) {
// pre:  0 ≤ n ≤ theArray.length, theArray is non-null and every
//       element of the array is non-null
// post: values in theArray[0..n-1] are a permutation of the
//       original values and in non-descending order
    for (int last = n-1; last >= 1; last--) {
       // theArray[last+1..n-1] is sorted and each element is
       //    >= any element in theArray[0..last]

        int largest = indexOfLargest(theArray, last+1);
        Comparable temp = theArray[largest];
        theArray[largest] = theArray[last];
        theArray[last] = temp;
    }
}
```

Understanding Selection Sort

- Convince ourselves and others that precondition + execute(method body) $\Rightarrow$ postcondition
- Examine preconditions, postconditions, and the loop

– what is true every pass through the loop?



- For `indexOfLargest`, loop picks out the largest element not yet included in sorted part
- For main loop, largest elements have been identified and they are in sorted order

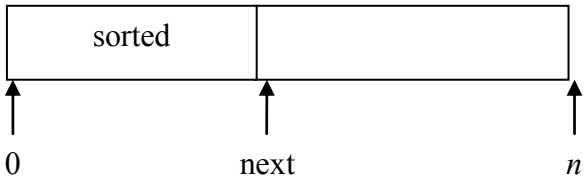
Understanding Linear Insertion Sort

Idea: repeatedly insert the element that happens to be next into the proper place among those elements already sorted.

see Carrano & Prichard, pp 483-485

- Examine preconditions, postconditions, and the loop

– what is true this time with every pass through the loop?



Efficiency of Sorting

- How much space is needed?
  - Always need space to hold the data itself; the critical question is how much *auxiliary* space is needed.
  - How much space is needed for Selection Sort beyond that used for the input arguments?

- What is the running time?
  - two important operations affecting time:
    - *data* comparisons: comparing values of two items
    - *data* movements: moving or copying a data item
  - Ignore operations on index values, etc.
  - *Rationale*:
    - number of other operations executed between comparisons and data movements is bounded by a constant

Efficiency of Selection Sort

- How much space is needed (using big-O)?

- What is the running time (using big-O) in the worst case? best case?

– Reminder: look at the structure of the code:

```
for (int last=n-1; last >= 1; last--) {
    ... indexOfLargest(...) ...
}
```

where indexOfLargest looks like:

```
for (int currIndex=1; currIndex < size;
                                currIndex++) {
    ...
}
```

and size = last+1

- How many data movements?

Efficiency of Linear Insertion Sort

- How much auxiliary space is needed?

- What is the running time in the worst case? best case?

- How many data movements?

A Recursive Sorting Algorithm: Mergesort

- Idea: merge results of applying mergesort to both halves of the data
- Uses “divide and conquer”

– divide a large problem into smaller problems
– solve the smaller problems
– put the solutions together to form an overall answer

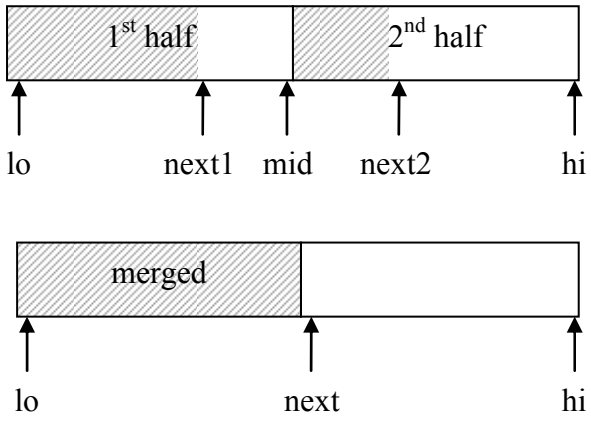
- In mergesort,

– smaller problems: sorting two half arrays
  * to be solved recursively
– merges the solutions of those two problems:
– start with a non-recursive wrapper method:
  `mergesort(Comparable[] theArray, int n)`
  calls `mergesort(theArray, 0, n-1);`

```
public static void mergesort(Comparable[] tA,
        int first, int last) {
// pre: (0 ≤ first ≤ last < tA.length) or (last < first)
// post: values in tA[first..last] are permuted into ascending order
   if (last > first) {
      mid = (first+last)/2
      mergesort(tA, first, mid);
      mergesort(tA, mid+1, last);
      merge(tA, first, mid, last);
   }
}
```

Merge

- Important subroutine: merge
  - Input is two sorted ranges in an array.
  - Identify candidate at start of each input range.
  - Repeatedly copy the smaller of the two candidates to the temporary array.
  - When one input range is exhausted, simply copy the rest of the other one to the temporary array.
  - Copy the temporary array back to the input array.

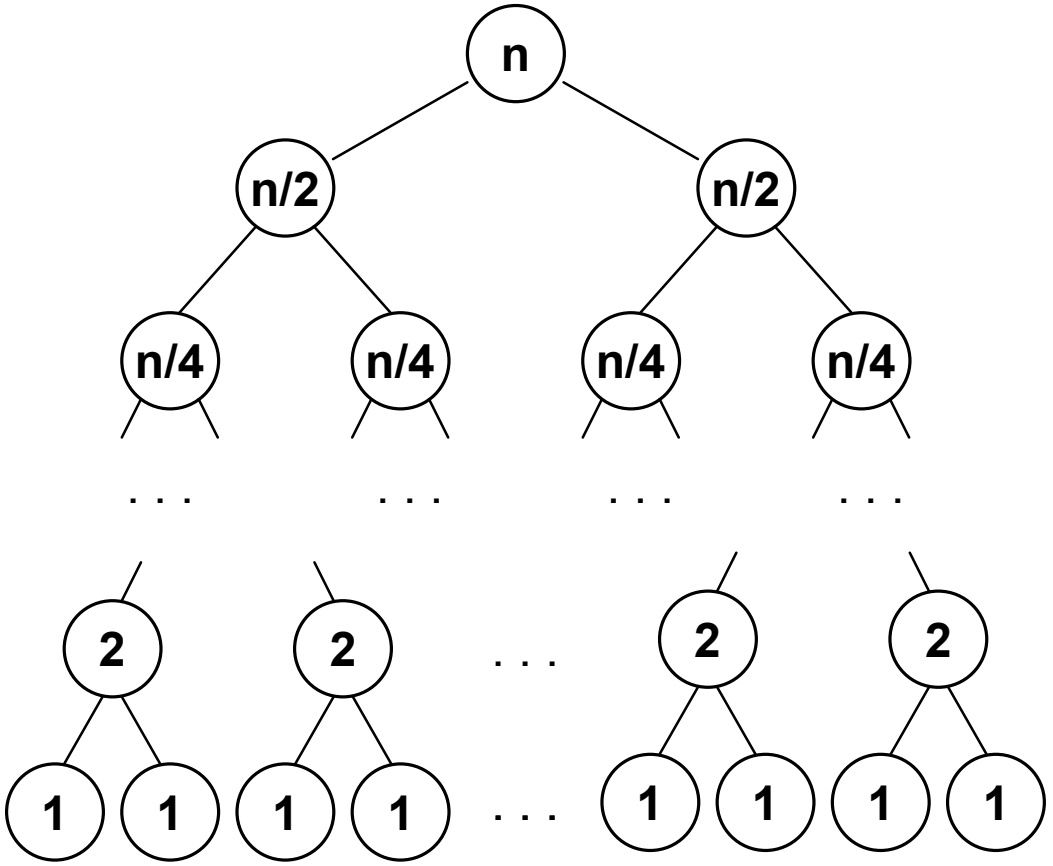


- time = space =

see Carrano & Prichard, pp 485-491

Analysis of Mergesort

- If n is a power of 2, the “tree of problems to solve” looks like:



where labels represent sizes of the problems to solve

- Runtime =

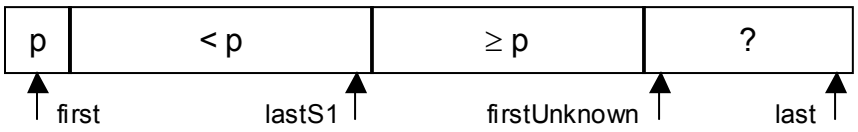
- Auxiliary space =

Quicksort

- Idea: pick some *pivot* element and place it where it belongs;
  - sort all elements less than the pivot;
  - sort all elements greater than the pivot
- Quicksort also uses divide and conquer, but the sizes of the two problems depend on the input.

  see Carrano & Prichard, pp 491-503
- Important subroutine: partition

```
int partition(Comparable[] theArray,
                    int first, int last);
// pre: 0 ≤ first ≤ last < theArray.length
// post: returns split s.t. first ≤ split ≤ last
// and permutes theArray s.t
// theArray [i] ≤ theArray [split] for first ≤ i < split
// theArray [i] ≥ theArray [split] for split < i ≤ last
```



Quicksort Itself

- Like mergesort, a recursive method with a non-recursive wrapper
- Contrast the order of the calls

```
quickSort(Comparable[] theArray, int n)
```

calls

```
quickSortRec(theArray, 0, n-1);
```

```
void quickSortRec(Comparable[] tA,
                  int first, int last) {
// pre: (0 ≤ first ≤ last < tA.length) or (last < first)
// post: values in tA[first..last] are permuted into ascending order
   int pivot;
   if (first < last) {
      pivot = partition(tA, first, last);
      quickSortRec(tA, first, pivot-1);
      quickSortRec(tA, pivot+1, last);
   }
}
```

Analysis of Quicksort (Time)

- Like mergesort, quicksort also uses divide and conquer, but the sizes of the two problems depend on the input.
- Best case
  - each segment split exactly in two $\Rightarrow$ O(n log n) [similar “tree of problems” to that of mergesort]
  - but splits having n/2 elements in each part not likely
- Worst case
  - each segment split on its first element $\Rightarrow$ $O(n^2)$
  - but how likely are splits into 0 and n-1 elements?
- So what can we expect on average?
  - assume all elements distinct and each one could be chosen as split element with equal probability
  - splits having n/4 elements in one of the parts still yield “logarithmic height tree” (but base of logarithm is smaller, so value is larger by constant factor) and probability of split at least this good is 0.5
  - similarly for splits having n/k elements in one of the parts, for any constant k
  - Intuitively: average case is like best case but with larger constant factor
  - O(n log n) average case can be proven

Analysis of Quicksort (Space)

- O(1) space for partition
- But space needed on program stack to manage the recursion
  - Stack will have an activation record for each segment of A that still remains to be sorted
  - Worst case occurs when lots of small segments remain to be sorted; could be O(n)
- Auxiliary space = O(log n) with clever re-programming

Speeding Up Quicksort

- Try to avoid bad splits (e.g., do not just choose first elements as pivots).
  - randomization works well
  - worst case still $O(n^2)$, but less likely in practice to have “bad” inputs
- Speed up the algorithm in practice by using linear insertion sort on small segments (e.g., < 10 elements).
  - increases range of base cases
  - also appropriate for speeding up mergesort
- Even better: Stop recursion without sorting small segments (e.g., < 10 elements) at all:
  - every element within 10 of its final position
  - final single call to linear insertion sort finishes overall sort quickly

Sorting Summary

	select	**insert**	**merge**	**quick**
best time	$O(n^2)$	$O(n)$	$O(n \log n)$	$O(n \log n)$
average time	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
worst time	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$
time for sorted input	$O(n^2)$	$O(n)$	$O(n \log n)$	$O(n \log n)$ *
aux. space	$O(1)$	$O(1)$	$O(n)$ †	$O(\log n)$ ‡

– Other properties: stability

---

* But $O(n^2)$ if pivots not properly selected.

† But some implementations use $O(n^2)$ space.

‡ But many implementations use $O(n)$ in the worst case.

History of some main ideas in Computer Science

- Calculating machines: 17th Century
  - 1614 - John Napier
    - logarithms: Napier bones
  - 1642 - Blaise Pascal
    - digital adding machine
  - 1671 - Gottfried Wilhelm Leibniz
    - multiplication, division, and square roots
- Punched card control: 19th Century
  - 1801 - Joseph-Marie Jacquard
    - Jacquard loom wove complicated patterns described by holes in perforated cards
  - 1835 - Charles Babbage
    - Analytical engine: operations and input values on perforated cards

    plus conditional execution and overwriting intermediate data (as well as instructions)
    - Lady Ada Lovelace: algorithm as program
  - 1886 - Herman Hollerith
    - Electrically read punched cards for tabulating
    - Sorting and punching peripherals
    - 1911: Computing Tabulating Recording Co. (evolved to become IBM)

What is computing?

- Late 19th Century
  - Formal approaches to set theory and algebra
- What can we compute?
  - 1900 - David Hilbert (Hilbert's problems)
    - Presented 23 problems for the 20th Century
    - How to formulate axioms for all of arithmetic and show them to be consistent?
  - 1910-1913 - B. Russell and A. N. Whitehead
    - *Principia Mathematica*: axiomatic logic
  - 1931 - Kurt Gödel (Incompleteness)
    - In any consistent formulation of arithmetic, some formulae are not provably true or false
    - *Gödel numbering* of all formulae
  - 1936 - Alan Turing, Alonzo Church, Emil Post
    - Turing machine: model of computation having a finite automaton controller read and write symbols on an unbounded tape
    - Computable functions
    - Universal Turing machine: takes the *Gödel number* of the Turing machine to emulate as a parameter
    - Undecidability: the halting problem

Early computers

- Motivating applications in 1940s
  - perform military computations
  - break codes
  - census
  - later, business applications
- Automatic digital computers, 1939-46
  - John Atanasoff & Clifford Berry (Iowa) – ABC
  - Konrad Zuse (Berlin) – Z1, Z2, Z3
    - (=> … Siemens)
    - Plankalkül programming language
  - Howard Aiken (Harvard) – Mark I
    - electromechanical computer
  - Presper Eckert & John Mauchly (Penn) – ENIAC
    - electronic computer
    - later development of Univac (=> … Unisys)
  - John von Neumann (Princeton) – EDVAC
    - *von Neumann machine*: single processor, stored program, stored return address for procedure call
    - von Neumann, Arthur Burks, & Herman Goldstine: design for parallel processors
  - Alan Turing (Nat'l Physical Lab, London) – ACE
    - "reversion storage" provided a hardware stack

Programming languages

- Algorithmic languages
  - Grace Murray Hopper's A-0 compiler (1951)
  - Fortran (1957), Cobol (1959)
  - Algol (1960)
  - PL/I (1965)
  - BCPL (1966), B (1972), C (1975)
  - Pascal (1970), Modula (1975)
  - Ada (1979)
- Array, list and string languages
  - IPL (1957), LISP (1958), Scheme(1975)
  - APL (1962)
  - COMIT (1962), Snobol (1964)
  - sed (1978), AWK (1978), PERL (1991)
- Object-oriented languages
  - Simula (1967)
  - Smalltalk (1972)
  - Alphard (1976), Clu (1979)
  - C++ (1983)
  - Java (1995)

Computer Science as a discipline

- Arose from mathematics, science, and engineering

  key words: algorithm, information (“informatics”)

  – term coined by George Forsythe, a numerical analyst and founding head (1965-1972) of Stanford Univ. CS Department

  – CS at Waterloo: formally founded in 1967 as the Department of Applied Analysis and Computer Science

CS subareas

[*Report of the ACM Task Force on the Core of Computer Science*, Denning, et al., 1989]

– Algorithms and data structures

– Programming languages

– Architecture

– Numerical and symbolic computation

– Operating systems

– Software methodology and engineering

– Database and information retrieval systems

– Artificial intelligence and robotics

– Human-computer communications

Some CS Notables

- Hardware
  - J. Cocke, I. Sutherland, D. Englebart
  - Supercomputers, personal computers
  - storage devices, peripherals, graphics
  - communications & distributed computing
- Operating systems
  - F. Brooks, E. W. Dijkstra
  - Multics (MIT), Unix (Bell Labs)
  - DOS, Mac-OS, Windows
- Computability and complexity
  - N. Chomsky
  - What can be feasibly computed?
    - NP-Completeness (S. A. Cook, R. M. Karp)
    - Public Key Cryptosystems
  - models of parallelism
- Correctness
  - R. W. Floyd, E. W. Dijkstra, C. A. R. Hoare
  - structured programming, software engineering

Motivating applications in the 1990s

- Business
  - database management
  - process planning
  - telecommunications
  - electronic commerce
- Science and engineering
  - scientific computation
  - symbolic computation
  - embedded systems
  - robotics
  - simulation
  - bioinformatics
- Human-computer interaction and entertainment
  - graphics
  - vision
  - natural language processing
  - information retrieval

A. M. Turing Award Recipients

"given to an individual … for contributions … of lasting and major technical importance to the computer field"

1966 A.J. Perlis
1967 Maurice V. Wilkes
1968 Richard Hamming
1969 Marvin Minsky
1970 J.H. Wilkinson
1971 John McCarthy
1972 E.W. Dijkstra
1973 C.W. Bachman
1974 Donald E. Knuth
1975 Allen Newell & Herbert A. Simon
1976 Michael O. Rabin & Dana S. Scott
1977 John Backus
1978 Robert W. Floyd
1979 Kenneth E. Iverson
1980 C. Antony R. Hoare
1981 Edgar F. Codd
1982 Stephen A. Cook
1983 Ken Thompson & Dennis M. Ritchie
1984 Niklaus Wirth
1985 Richard M. Karp
1986 John Hopcroft & Robert Tarjan
1987 John Cocke
1988 Ivan Sutherland
1989 William (Velvel) Kahan
1990 Fernando J. Corbató
1991 Robin Milner
1992 Butler W. Lampson
1993 Juris Hartmanis & Richard E. Stearns
1994 Edward Feigenbaum & Raj Reddy
1995 Manuel Blum
1996 Amir Pnueli
1997 Douglas Engelbart
1998 James Gray
1999 Frederick P. Brooks, Jr.
2000 Andrew Chi-Chih Yao
2001 Ole-Johan Dahl & Kristen Nygaard
2002 Ronald L. Rivest, Adi Shamir, & Leonard M. Adelman
2003 Alan Kay
2004 Vincent G. Cerf & Robert E. Kahn