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 Univ. 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 artefacts 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

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

        * s.size()

    – can access element at an arbitrary index of List s

        * s.get(index)

    – can insert new elements

        * s.add(index, item)

    – can remove elements

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

    – special check for empty List

        * s.isEmpty() $\equiv$ (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: $0 \leq i < a.length$

- 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: $0 \le i < a.length$
// 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* 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 boolean isEmpty();
    // post: Returns true iff this is empty

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

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

    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 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().

}
```

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 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

  – <u>problem</u>: every use of **new** exposes the representation

  – <u>solution</u>: 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

program that uses ADT		implementation 1 of ADT
module 1		
module 2	ADT method calls	implementation 2 of ADT
module 3		
data factory		implementation 3 of ADT
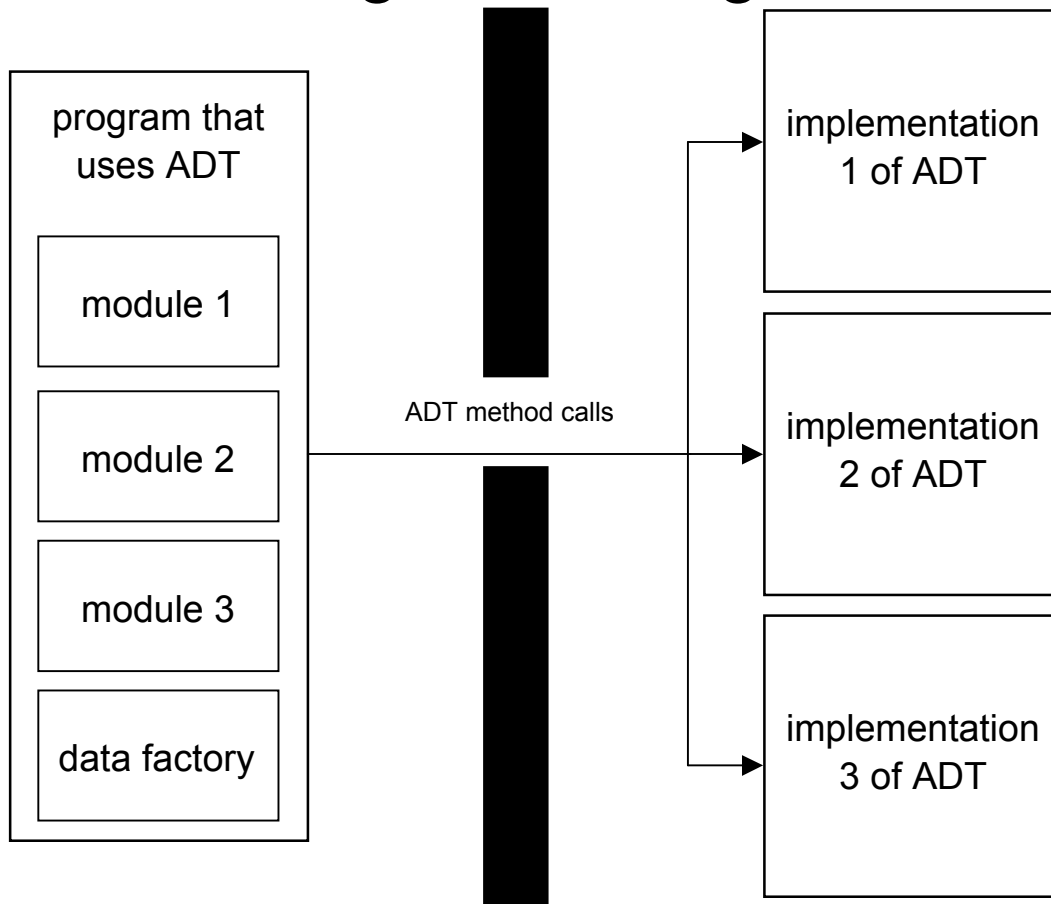
- Choice of which implementation to use is isolated within data factory.

- 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

- 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

- *Carrano & Prichard* only discuss this briefly on pp. *160-161*

- this is precisely what a Vector or ArrayList is (java.util)

- *N.B.* A Vector or ArrayList is not really an ADT because of `trimToSize()` and `capacity()`

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

space for "local" method variables

space for explicitly allocated storage

count   45

grades   → | 73 | 99 | 65 |

student   →

93123456

→ **Jane Doe**

→ **Honours CS**

where reference or primitive value is stored

where object or array is stored
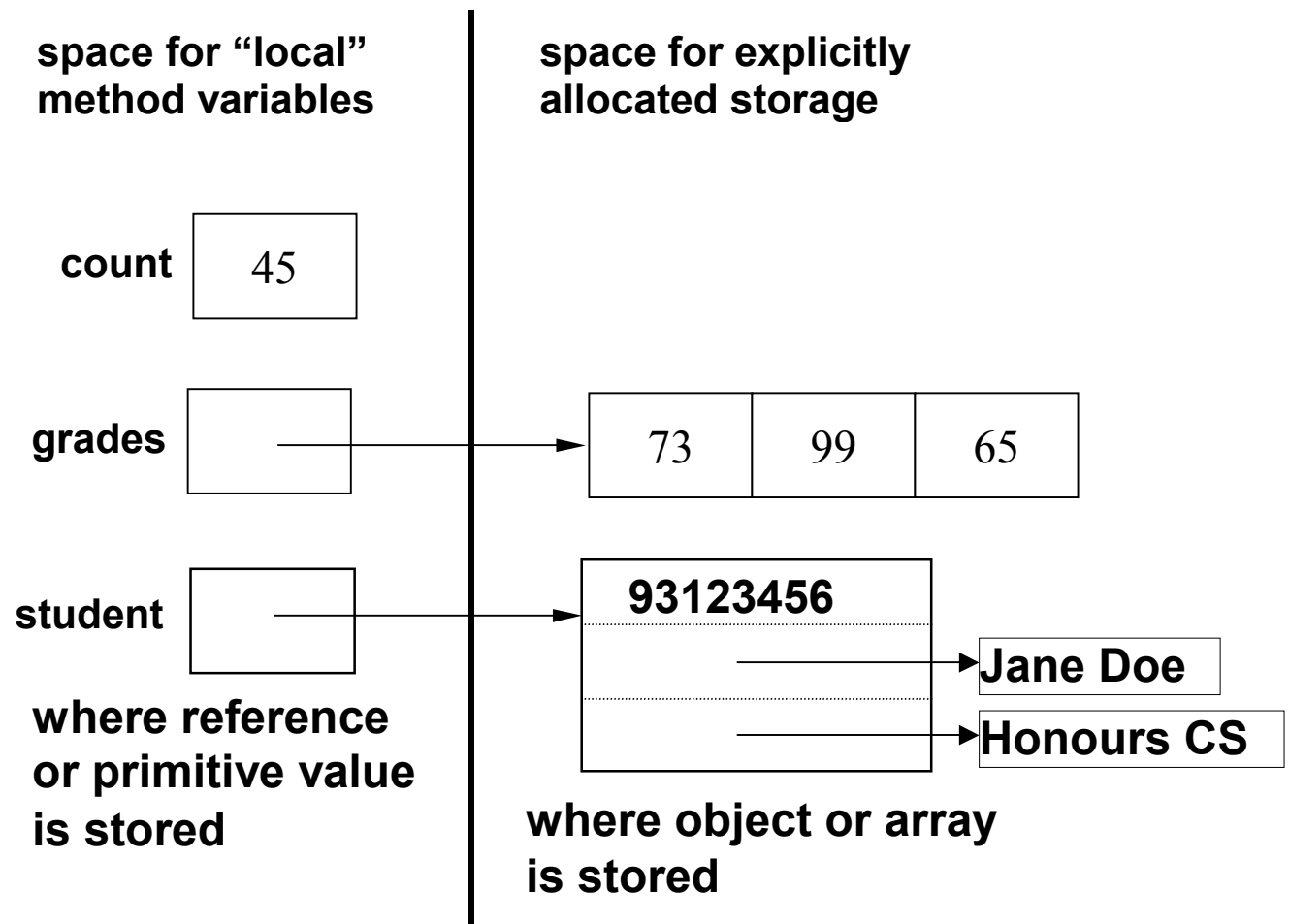
Using references in Java

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);

i   ☐

j   ☐

a   ☐

b   ☐

p   ☐

q   ☐

r   ☐

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
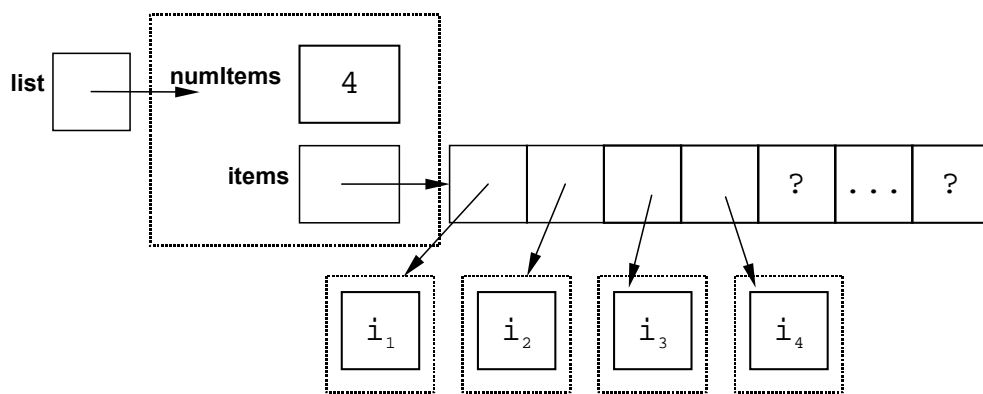
or more simply diagrammed as
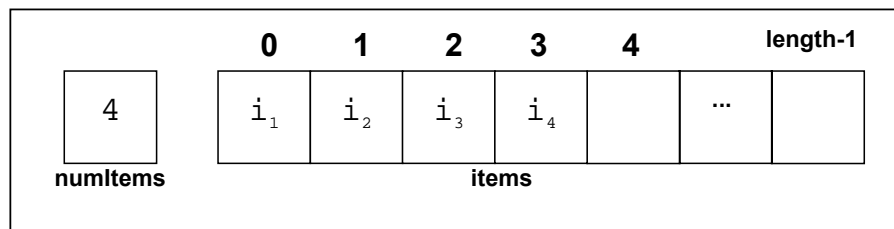
- Straightforward implementations for size(), isEmpty() and get(index) using

```
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 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().

  if (index >= 1 && index <= numItems) {
    for (int pos = index+1; pos <= size(); pos++)
    {
      items[translate(pos-1)] =
            items[translate(pos)];
    }
    items[translate(numItems)] = null;
    numItems--;
  }
  else {
      throw new ListIndexOutOfBoundsException(
      "ListIndexOutOfBoundsException on remove");
    }
}
```

- 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
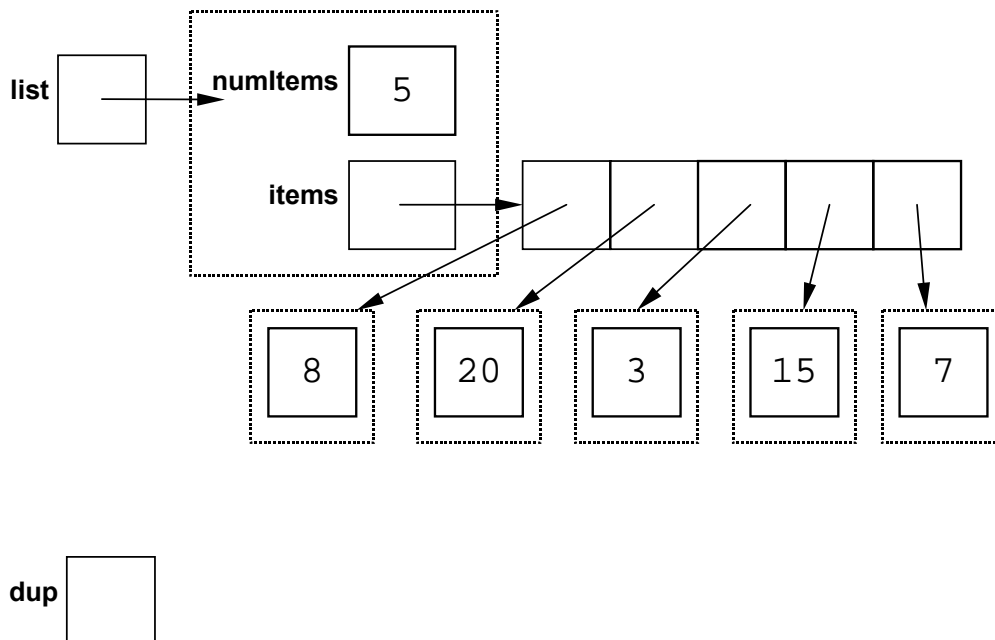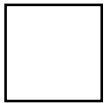
Alternative implementation of List: singly-linked list
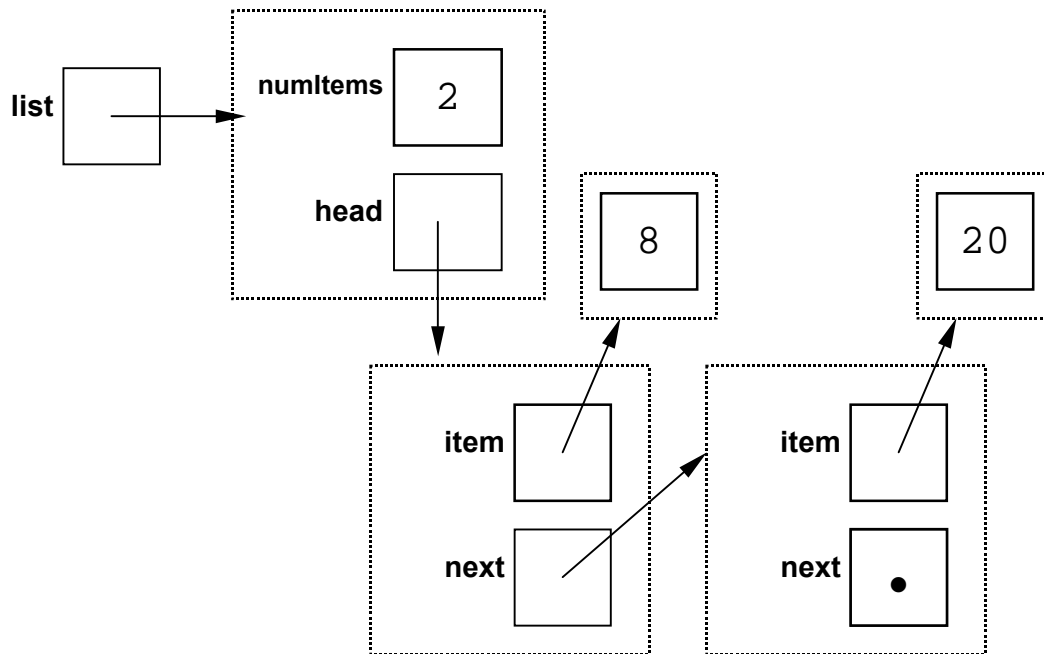
- Expands and contracts as needed
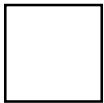
- Diagrammatic representation
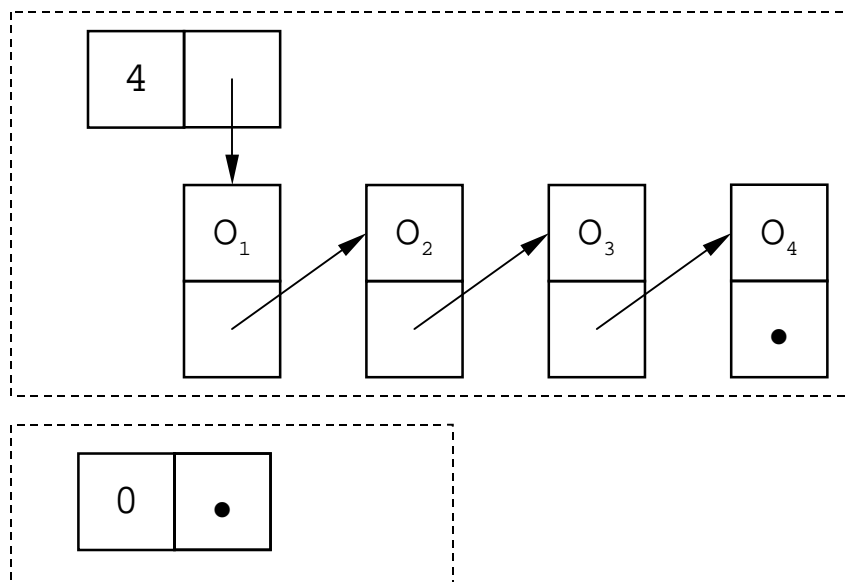
Manipulating linked list elements
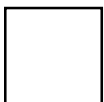
Node methods: see Carrano & Prichard p 163

```
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 node = new Node(myNode.getItem());
temp.setNext(node);
```

myNode

temp

node

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 void remove(int index)
   throws   ListIndexOutOfBoundsException {
   // post: see previous slide for precondition and postcondition
      if (index >= 1 && index <= numItems) {
         if (index==1){ head=head.getNext();
         } else {
            Node prev = find(index-1);
            Node curr = prev.getNext();
            prev.setNext(curr.getNext());
         }
         numItems--;
      } else {
         throw new
            ListIndexOutOfBoundsException();
      }
   }
   ...
}
```
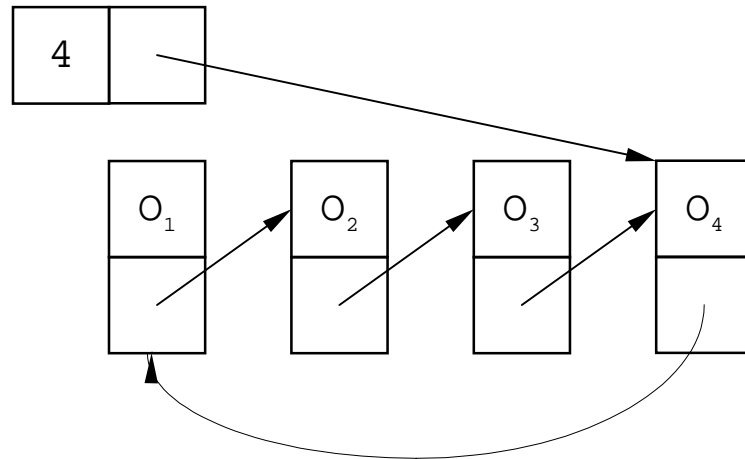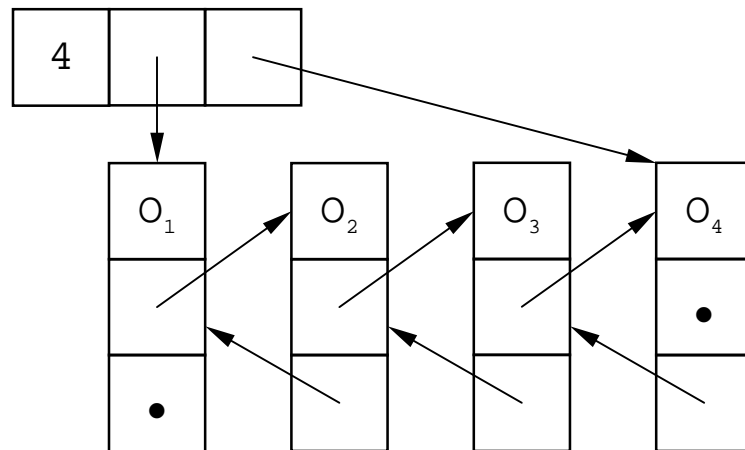
Variations on linked lists

- 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
  - 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

```java
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());
  }
}
```
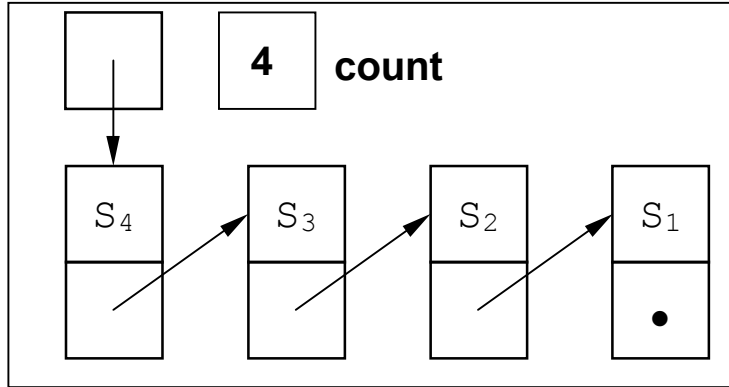
- Why does this algorithm work?

Array-based implementation of Stack

0	**1**	**2**		**length-1**

1
top

s_1	s_2		...	

items
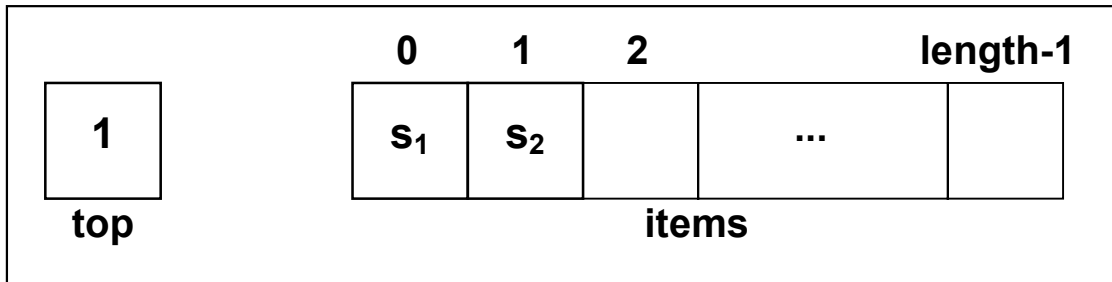
Linked-list-based implementation of Stack

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.
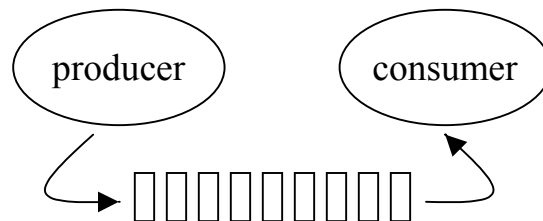
- Rate of consumption or production often data-dependent.

- Producer can use *enqueue* to insert and Consumer can use *dequeue* to take data in insertion order.

Queue using a circular linked list
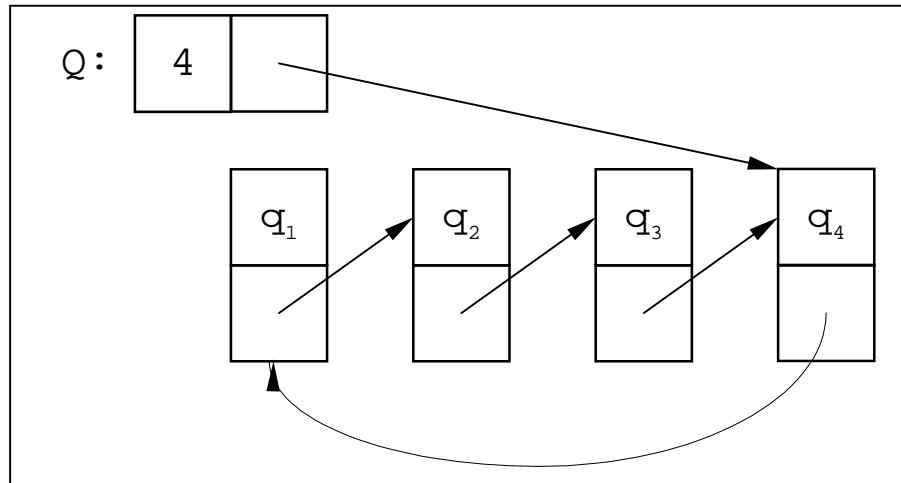


```
public Object dequeue ( ) {
// pre: queue is not empty
// post: the head of the queue is removed and returned
    Object val = tail.getNext().getItem();
    count--;
    if (count == 0) tail = null;
    else tail.setNext(
            tail.getNext().getNext());
    return val;
}
public void enqueue(Object 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++;
}
```
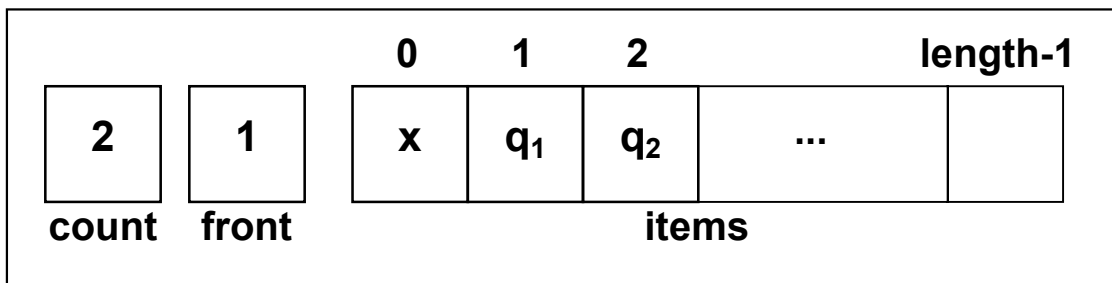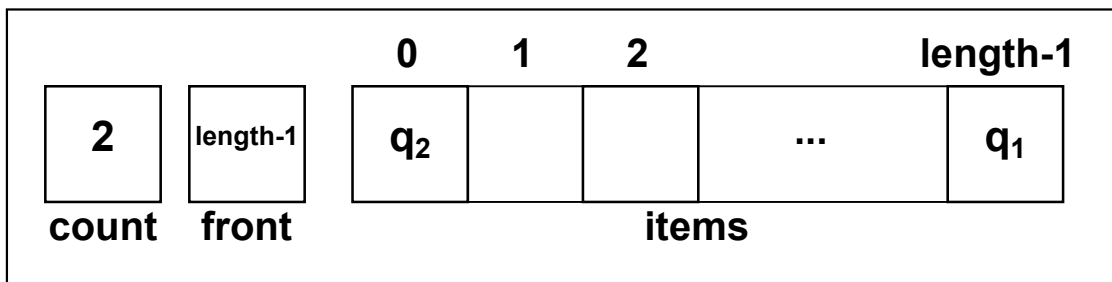
Queue using a "circular" array

- Naïve use of array

		0	1	2		length-1
2		q_1	q_2		...	
count				**items**		
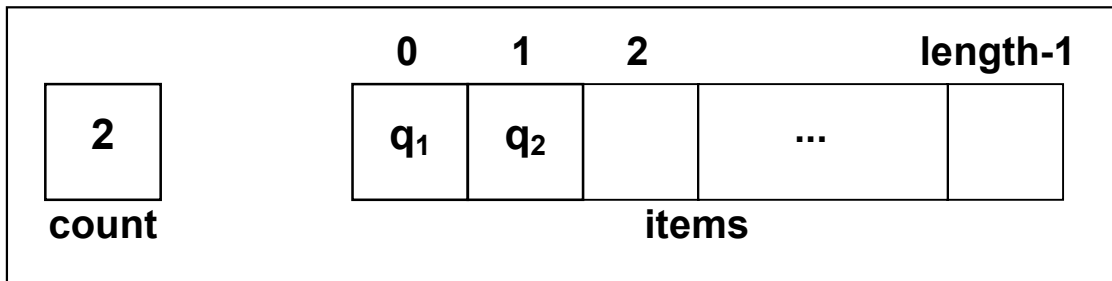
   – dequeue $\Rightarrow$ shuffling values to start of array

- Alternative: let queue drift down array

		0	1	2		length-1
2	**1**	x	q_1	q_2	...	
count	**front**			**items**		

   – when reaching end of array, wrap around to start

		0	1	2		length-1
2	length-1	q_2			...	q_1
count	**front**			**items**		

- Implementation details . . .

   see Carrano & Prichard, Chapter 7

The use of queues in simulation

- Example: a store's cashier

- Long periods of waiting punctuated by occurrence of events

- Possible events:
  - arrival of a new customer
  - completion of service to a customer checking out

- Event-driven simulation
  - concentrates on the events
  - ignores the time in between when nothing changes

Event-driven simulation

- pseudocode for event-driven simulation:

```
time = 0
while (simulation not over)
    time = time of next event
    update state according to event
```

- for the example: arrivals and departures

  – find which event is next by "polling" possible sources (e.g., check time for next arrival vs. time for next departure)

- For an arrival, check for a free cashier:

  – if there is one, the customer goes to that cashier

 – otherwise, the customer joins the line of previously waiting customers
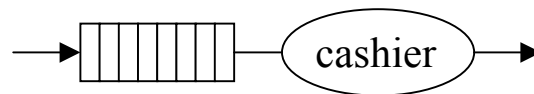
- Determining times for various events:

  – need to know arrival times and how long it takes to service each customer

    ∗ use data from a log obtained from tracing existing system (trace-driven simulation)

    ∗ use data generated from a model based on probability distributions

Collecting statistics

- in the store example: average customer wait, average length of the line, amount of busy time of cashiers

- only need to update statistics when events occur

- Where do queues come in?
  - For a bank, there is usually one queue representing the waiting customers; for supermarkets there are several.
  - There could be several classes of queues for various services.
  - Queue item: object representing customer (e.g. recording arrival time)

- What data to keep for simulating the following system?
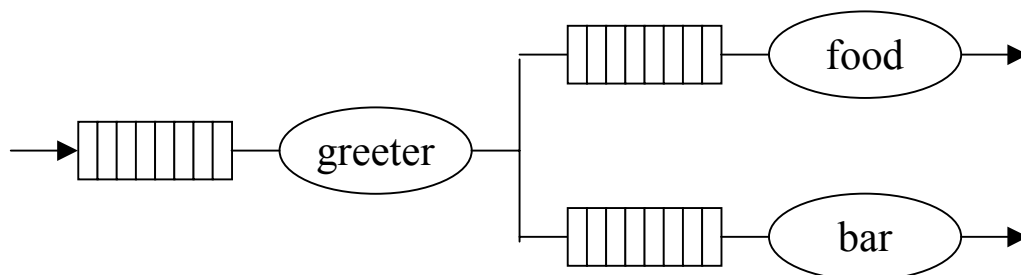
Ariane 5

> *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.*
>
> [*Ariana 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….*
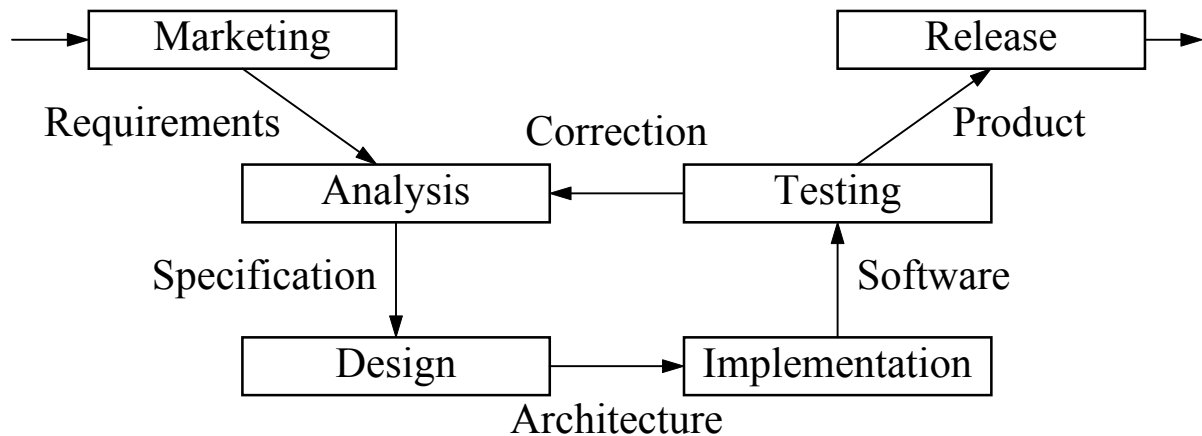
Software in the real world

- 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"

Marketing		Release

Requirements   Correction   Product

| Analysis | ← | Testing |

Specification   Software

| Design | → | Implementation |

Architecture

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

Professional Ethics

> *References:*
>
> Michael J. Quinn, *Ethics for the Information Age*, Addison-Wesley, 2005.
>
> Deborah G. Johnson, *Computer Ethics, Third Edition*, Prentice Hall, 2001.

- **Computing affects society at large**
  - Safety (e.g., Ariane 5, Therac-25)
  - Security (and hence privacy)
  - Reliability (i.e., systems and applications fulfill their purpose)
  - Protection of intellectual and other property

- **Computer scientists need to address variety of responsibilities:**
  - *Role responsibility* results from assigned duties
  - *Causal responsibility* results from previous actions (or inactions)
  - *Legal responsibility* results from laws
  - *Moral responsibility* results from human society

    Note: Unlike the others, moral responsibility is not exclusive; it cannot be avoided by passing it on to others.

    Responsibilities arise in a variety of relationships: with employers, support staff, clients, society at large, and other computing specialists.

Dealing with Responsibilities

Unfortunately, responsibilities sometimes conflict (e.g., dilemma of whistle-blowers).

Example scenarios:

- [Johnson, p.55] You are in charge of designing a database management system for the personnel office of a medium-sized company. After reviewing the options for providing security, the client insists that the cheapest security system be implemented, in spite of your detailed explanation of the security risks. You know that highly confidential information will be stored in the database. What should you do?

Further scenarios

- [Quinn, pp. 386-7] While installing a software package on a colleague's computer, you accidentally come across directories with suspicious looking names. In spite of company policy against reading other peoples email, web logs, and personal files, you look further and discover that several files contain child pornography. What should you do?

- [Quinn, pp. 387-88] While battling a new computer worm, you realize that one way to close the responsible software loophole is to write an anti-worm program that spreads like a worm, patching the loophole on each machine that it reaches. You implement the anti-worm, prove that it will not cause any other damage to any target machines, and release it, taking precautions that the anti-worm cannot be traced back to you. Is this ethical?

Computing Profession

- ## Characteristics of a profession

  Typified by doctors, lawyers, engineers, architects, accountants, clergy

  – Mastery of an esoteric body of knowledge.

  – Autonomy.

  – Formal organization.

  – Code of ethics.

  – Fulfillment of an important social function.

  Members of a profession are held to higher standards of responsibility.

- ## Infrastructure to support a profession

  – Initial professional education

  – Accreditation

  – Skills development

  – Certification

  – Licensing (in case of exclusive right to practice)

  – Ongoing professional development

  – Code of ethics

  – Professional society

Fundamental ethical principles

- Be impartial.

- Disclose information that others ought to know.

- Respect the rights of others.

- Treat others justly.

- Take responsibility for your actions and inactions.

- Take responsibility for the actions of those you supervise.

- Maintain your integrity.

- Continually improve your abilities.

- Share your knowledge, expertise, and values.

Code of Ethics and Standards of Conduct (CIPS)

P) To the public: I will endeavour at all times to protect the public interest. I will strive to promote understanding of information systems and their application. I will not represent myself as an authority on topics in which I lack competence.

M) To myself and my profession: I will guard my competence and effectiveness as a valuable possession. I will work to maintain them despite changing circumstances and requirements. I will demonstrate the highest personal standards of moral responsibilities, character, and integrity when acting in my professional capacity.

C) To my colleagues: I will treat my colleagues with integrity and respect their right to success. I will contribute to the information systems profession to the best of my ability

E) To my employer and/or clients: I will give conscientious service to further my employer's and/or client's legitimate best interests through management's direction.

R) To my employees and contracted staff: I will observe their obligation to uphold the Code of Ethics of the professional societies to which they belong.

S) To my students: I will provide a scholarly education to my students in a supportive and helpful manner.

These are refined in the code though a list of specific, non-exhaustive obligations.

Principles of the Software Engineering Code of Ethics (ACM/IEEE)

(with cross-references to CIPS categories)

1. Software engineers shall act consistently with the public interest. (P)

2. Software engineers shall act in a manner that is in the best interests of their clients and employer, consistent with the public interest. (E)

3. Software engineers shall ensure that their products and related modifications meet the highest professional standards possible. (C,E)

4. Software engineers shall maintain integrity and independence in their professional judgment. (P,M)

5. Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance. (E,R)

6. Software engineers shall advance the integrity and reputation of the profession consistent with the public interest. (P,C)

7. Software engineers shall be fair and supportive of their colleagues. (C,R)

8. Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession. (M)

These are refined in the code though a list of clauses giving specific, but non-exhaustive examples.

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

- ## <u>Example 1</u>: factorial function

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

- ## <u>Example 2</u>: Fibonacci numbers

  $$f_0 = f_1 = 1 \qquad \text{\{base cases\}}$$
  $$f_n = f_{n-1} + f_{n-2} \text{ if } n \geq 2 \text{ \{recursive case\}}$$

- ## <u>Example 3</u>: balanced strings

  - base case:

    A string containing no parentheses is balanced.

  - recursive cases:

    (x) is balanced if x is a balanced string.
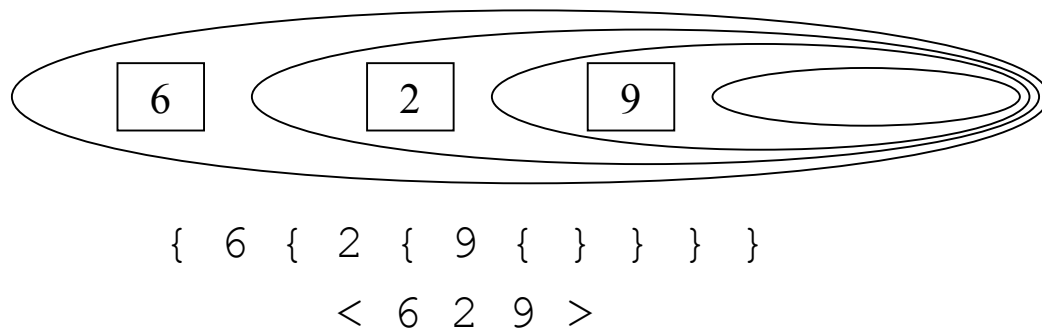
    xy is balanced if x and y are balanced strings.

Recursive definitions (cont'd)

- ## Example 4: sublist
  - ➤ an empty collection   `{base case}`
  - ➤ the pair <H,L> where H is a   `{recursive case}`
    head element and L, the
    remainder, is a sublist

  e.g., <6,<2,<9,ϕ>>> where ϕ is the empty collection

  – alternative "pictures"

```
{ 6 { 2 { 9 { } } } }
     < 6 2 9 >
```

- ## Example 5: contains
  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

- ## Example 6: encompasses
  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.

Recursive structures

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

   public Object getRootItem() throws
                         NestingException;
   // post:  Returns value associated with the root if this is not
   //        empty; otherwise throws NestingException.

   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 makeEmpty();
   // post:  this is empty.
}
```
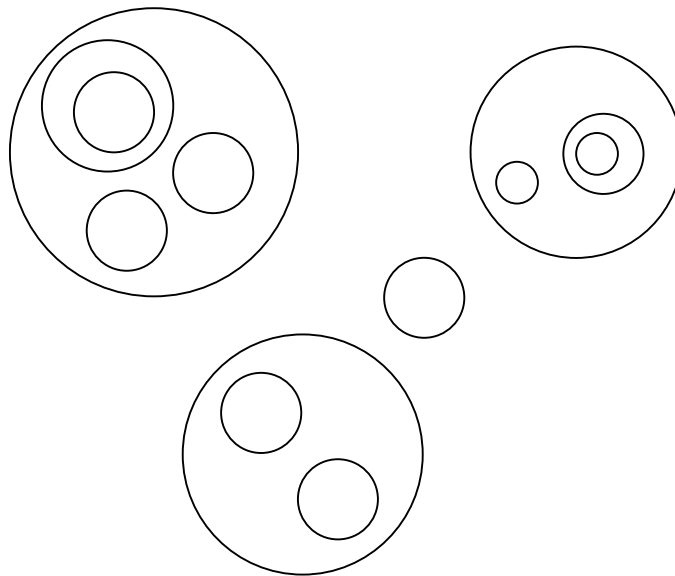
Sublist ADT

- linear nestings

- need nesting + access to nested sublists

```
class SubListException extends NestingException{}

public interface SubListInterface extends
                              NestingInterface {

    public void attachRemainder
                    (SubListInterface newRem)
              throws SubListException;
```
// pre: newRem is non-null.
// post: Throws SubListException if this is empty or the
//   remainder of this is non-empty; otherwise, attaches
//   the sublist referenced by newRem as the remainder
//   of this and sets newRem to be the empty sublist.

```
    public SubListInterface detachRemainder()
                throws SubListException;
```
// post: Throws SubListException if this is empty; otherwise,
//   returns the sublist that is the remainder of this and sets
//   the remainder to be the empty sublist.

```
}
```
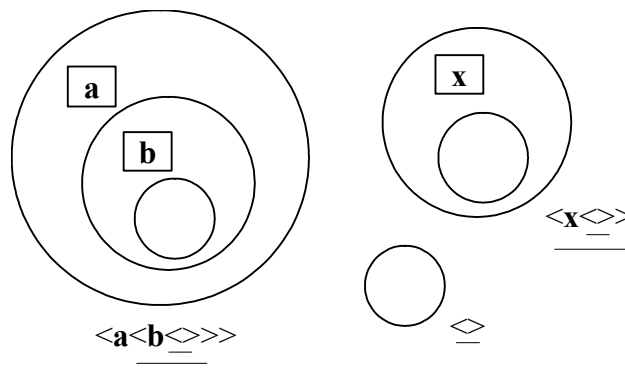
Implementing SubListInterface

- Assume the class SubList implements the SubListInterface.

```
public class SubList implements
                      SubListInterface {

    ... code for each method in the interface ...

    protected SubList remainder()
                 throws SubListException {
    // post:  throws SubListException if this is empty; otherwise,
    //         returns the remainder.

        ...  code similar to code for detachRemainder, but
             without changing the sublist denoted by this  ...

    }
}
```

- 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!!
    ```
  - available to any class in the same package and to any class extending SubList, but not to other users of SubList
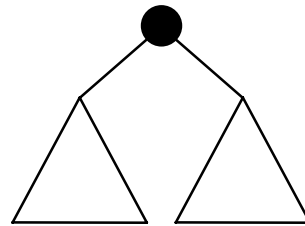
A non-linear recursive structure

- 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
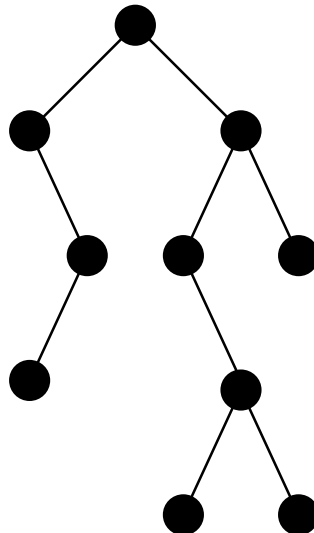
is
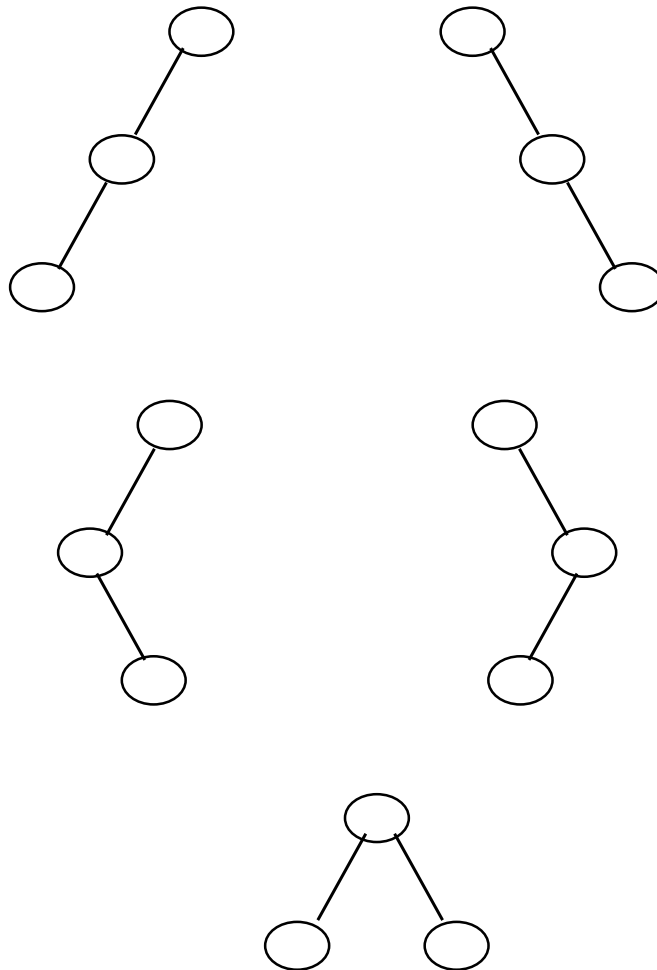
or
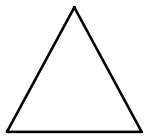
base case, empty tree   *recursive case*

- Tree terms for nodes: root and leaf

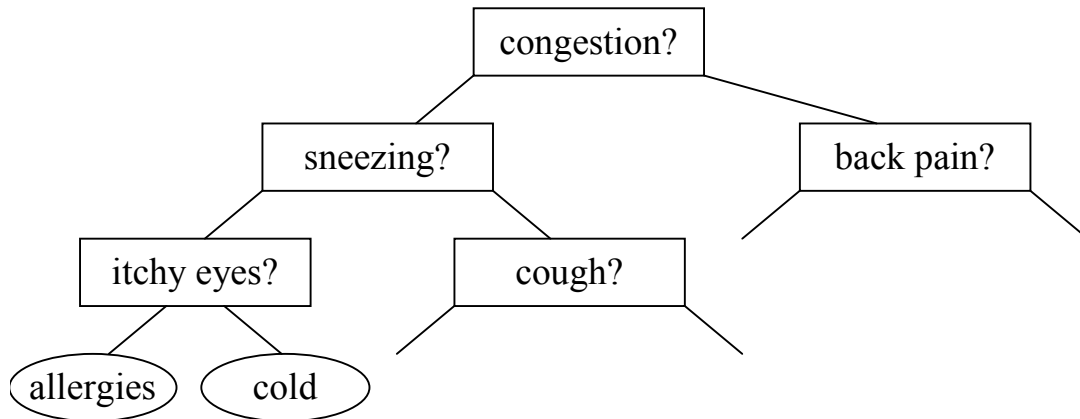- Familial terms for nodes: parent, child, sibling, ancestor, descendant
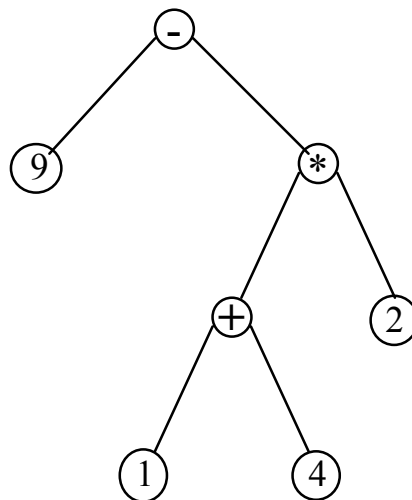
Binary trees on three nodes

- 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

```
class TreeException extends NestingException{}

public interface BinaryTreeInterface extends
                                    NestingInterface {

    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 attachLeftSubtree (BinaryTreeInterface
                    leftTree) throws TreeException;
    // pre:    leftTree is non-null.
    // post:   Throws TreeException if this is empty or the left subtree
    //         is non-empty; otherwise, attaches value of leftTree as the
    //         left subtree of this and sets leftTree to be the empty tree.

    public BinaryTreeInterface detachLeftSubtree() throws
                                            TreeException;
    // post:   Throws TreeException if this is empty; otherwise,
    //         returns the left subtree of this and sets the left subtree
    //         to be the empty tree.

    ...   and analagous methods for right subtrees
}
```

- leftSubtree and rightSubtree useful protected methods in classes implementing BinaryTreeInterface

- *Carrano & Prichard* extend BinaryTreeBasis instead of Nesting and do not include leftSubtree() and rightSubtree() methods

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
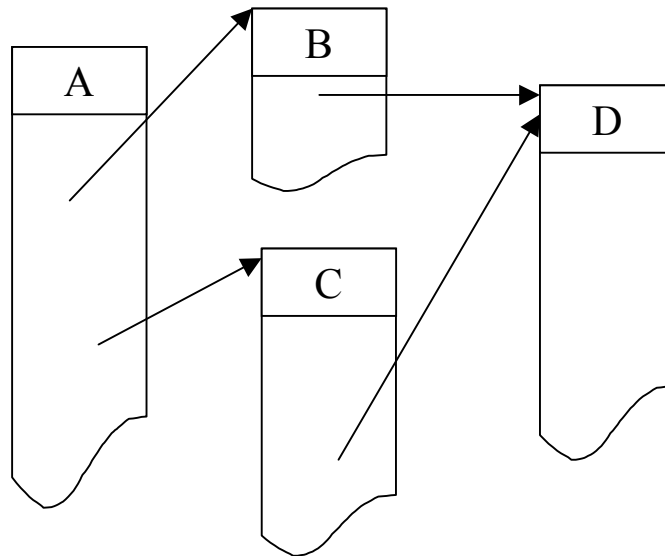
• same mechanism for recursive programs

Recursive sublist programs

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

1. calculating the size of a sublist

```java
public int size() {
// post:  Returns number of elements in this.
    if (isEmpty()) return 0;
    return 1 + remainder().size();
}
```
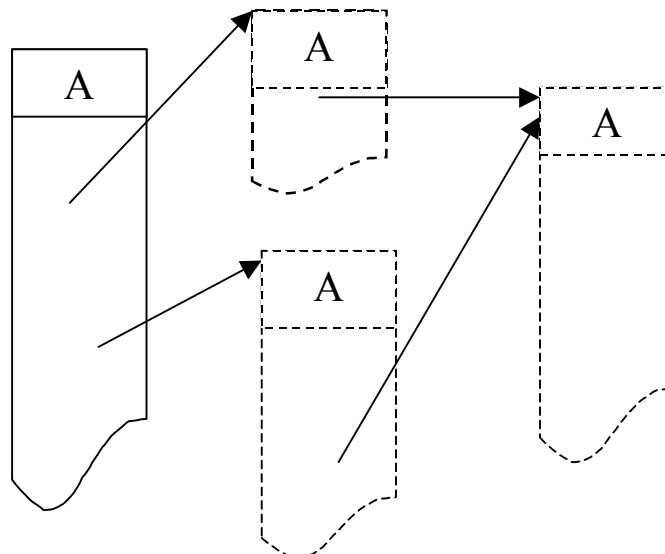
2. searching for an item in a sublist

```java
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

```java
public void printReverse() {
// pre:   this is not empty.
// 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 10

    - sublist can be implemented using the Node class, similarly, linked implementation for binary trees can use a TreeNode class:

    - 3 data fields (item, leftChild, rightChild)
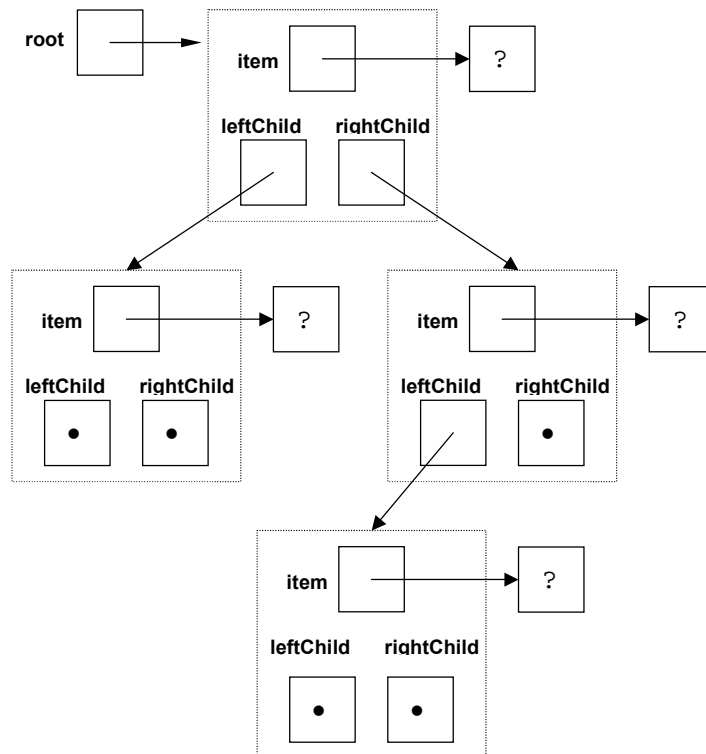
- **our approach:** use recursive BinaryTree references in place of TreeNode references

- could also store count or 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
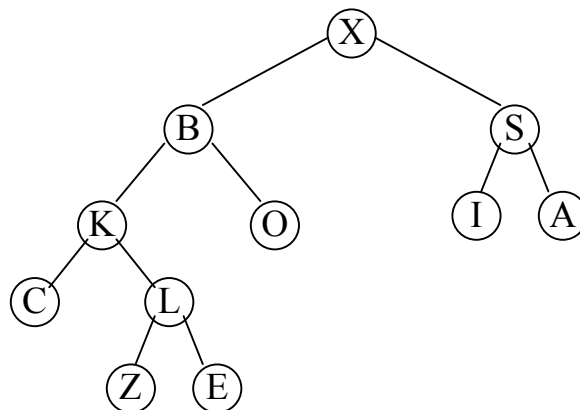
    - traversal: X B S K O I A C L Z E

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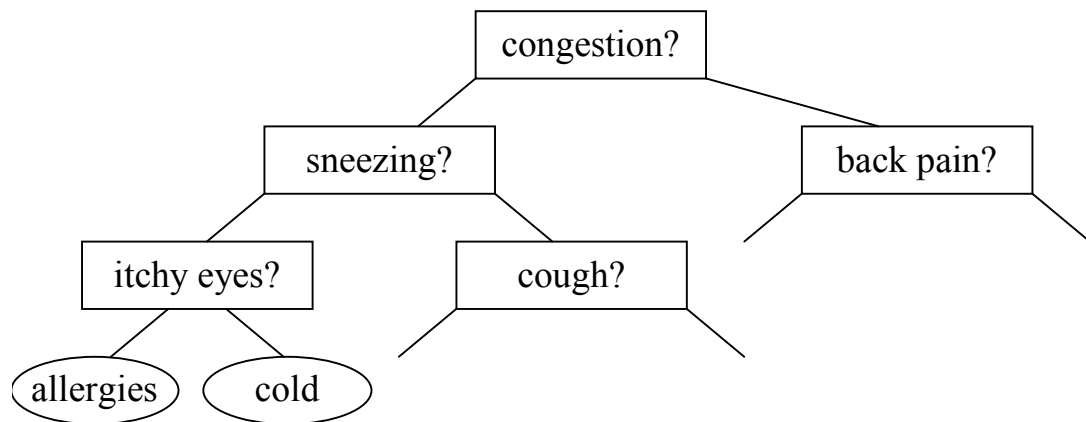
- Preorder traversal yields parents before children, but does not completely characterize a tree's structure.

Implementing tree iterators

- In the absence of parent pointers, iterator supporting depth-first traversal requires a stack.
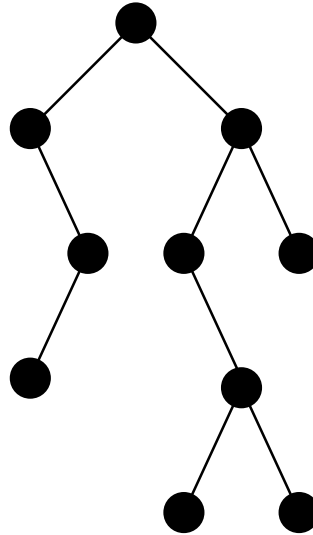
- For preorder, stack maintains list of non-empty subtrees remaining to be visited
  - when visiting a node, push right subtree (if non-empty) *and then* left subtree (if non-empty) onto stack
  - peek the top of the stack to find current node
  - pop the stack to find next node to visit
  - iterator 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
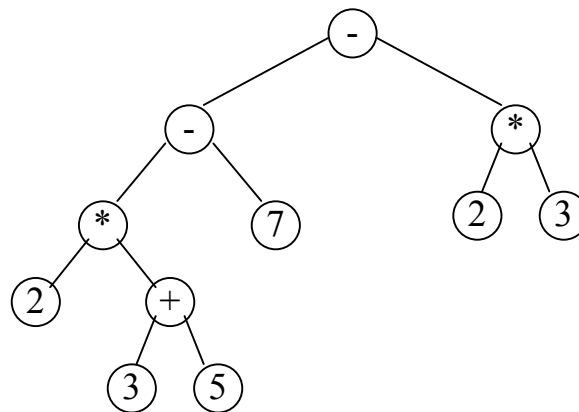
  - traversal: `2 * 3 + 5 - 7 - 2 * 3`
  - but need to insert `(` before visiting any subtree and `)` after visiting any subtree

        `( ( ( (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)
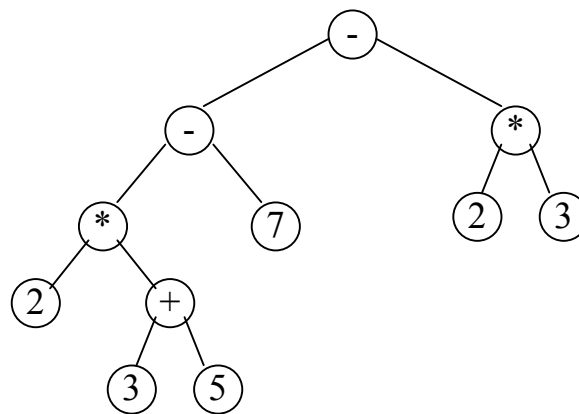

```
2 3 5 + * 7 - 2 3 * -
```

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

> Assume that height is defined in BinaryTree class

```
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);
  }
}
```

Proving properties about trees

Theorem: A binary tree of height h has at most $2^h - 1$ nodes.

Proof: (by induction on tree height)

base case: $h = 0$

By definition, $h = 0$ implies the tree is empty. Thus, there are 0 nodes. Since $2^0 - 1 = 0$, theorem is true for this case.

inductive case: Assume the theorem is true for all $h < k$, for some $k > 0$, and prove it for trees of height $h = k$.
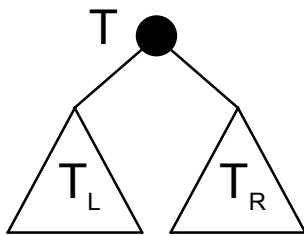
Assume T has height k for some value of $k > 0$. Then both subtrees are of height $< k$. By the inductive hypothesis, T_L has at most $2^{k-1} - 1$ nodes and T_R has at most $2^{k-1} - 1$ nodes. Thus T has at most $1 + 2(2^{k-1} - 1) = 2^k - 1$ nodes. $\square$

"Strong" induction

- To prove P(n) true for all n≥0:
  - Base case: Show that P(0) is true
  - Inductive hypothesis: Assume property P(i) is true for i=0,1,...,k-1.
  - Inductive conclusion: Show, using the inductive hypothesis, that P(k) is true.

- Variations
  - base case something other than 0
  - several base cases
  - assume P(0), P(1),...,P(k) true and use them to prove P(k+1) is true
  - "weak" induction: assume P(k-1) true and use it alone to prove P(k) is true

- The form of the inductive proof usually matches the form of the recursion.

Number of leaves in a binary tree

- 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*.

- *Theorem*: In any non-empty binary tree, the number of leaves is one more than the number of full nodes.

    - *Proof:*

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 134 (mapping from ID to "taking CS 134")

- 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 vector
    - using a linked list

- efficiency of operations:

    Consider a vector representation.

    Look at number of comparisons of two keys and number of elements whose locations in the vector 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

Ordering collections

Why does a vocabulary dictionary keep words in alphabetical order?

- To find a word, we could start at the beginning and scan every word in the dictionary.

- We can do better if 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
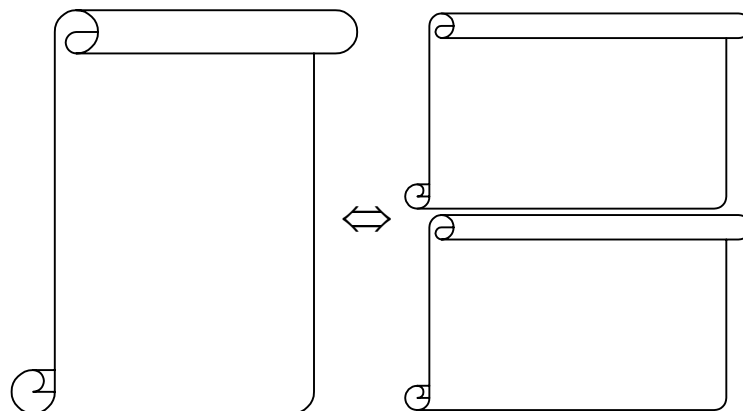
- requires that keys be "comparable"

Binary search

Given a Vector of comparable objects in ascending order, find the index matching a target key if it is present, or otherwise return the index of the slot *where it would be inserted*.

returns index such that $0 \le$ index $\le$ data.size(),
data[0..index-1] < target, *and*
data[index..data.size()-1] $\ge$ target

```
int position(Comparable target){
// pre: target is non-null and data values ascending
// post: returns ideal position of target in data vector
    return search(0,data.size(),target);
}


int search(int lo, int hi, Comparable key) {
// pre: 0 ≤ lo ≤ hi ≤ data.size( ); 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.elementAt(mid);
       if (m.compareTo(key) < 0)    // m < key
            return search(mid+1, hi, key);
       else  return search(lo, mid, key);
    }
}
```

● How efficient is binary search?

Bounding efficiency

- Running time of a program is a function of the "size" of the problem being solved
    - for collections: size = number of elements
    - for binary search: size = hi – lo

Consider solution to a problem of size n

- Running time using one compiler and one machine might be

$$.33365n^2 - .43n + 3.4 \ \mu sec$$

- 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 4 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)

$c_1 * g_1(n)$

running time of A1

n sufficiently large   **n**

$c_2 * g_2(n)$

running time of A2

n sufficiently large   **n**
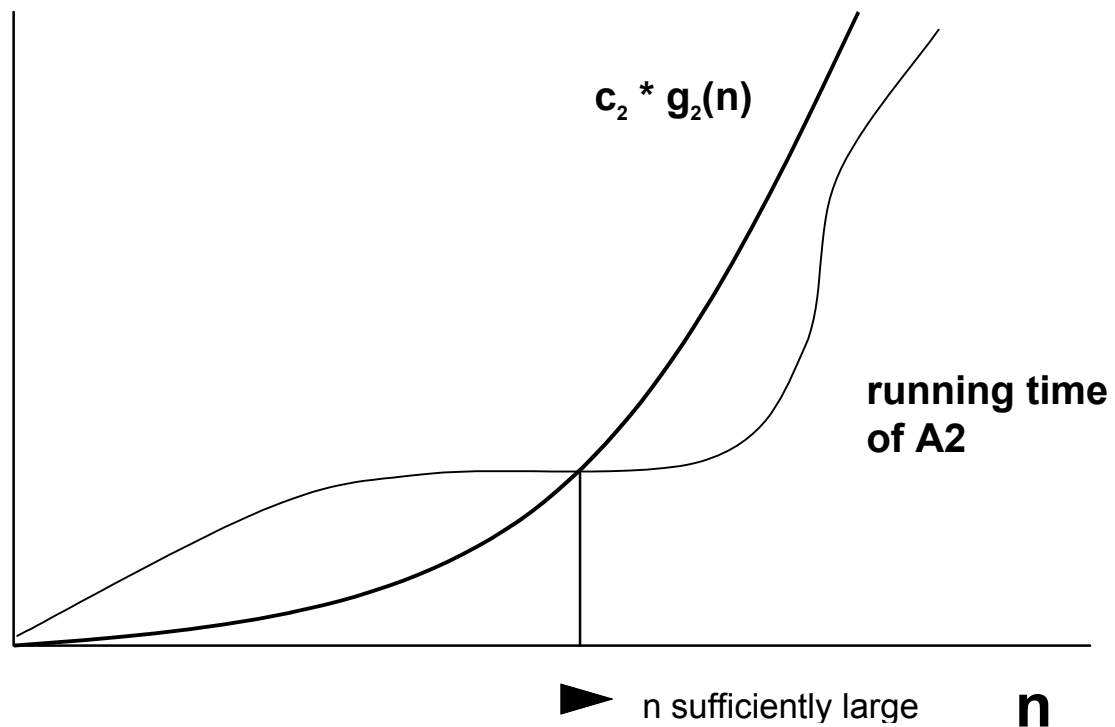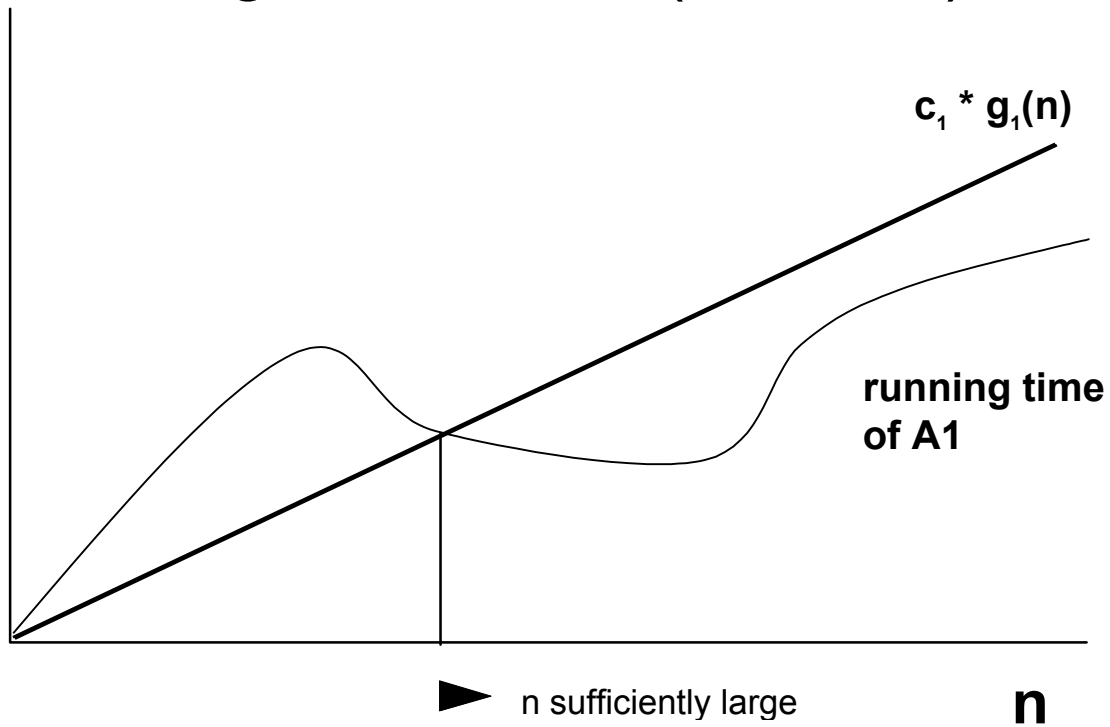
Algorithm A_i has running time $O(g_i(n))$

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**

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

	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	$3\ \mu s$	.195 s
100	.003 s	1.95 s
1000		
2500		
10^4		
10^5		
10^6		

. . .

Faster hardware isn't good enough!

Efficiency of binary search

- *asymptotic analysis*: interested in behaviour for large vectors

```
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.elementAt(mid)
          return search(mid+1, hi, key);
       or return search(lo, mid, key);
}
```

- Each recursive call halves vector:

  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($\log_2 n$)

- Doubling the size of the vector requires only one more call to search!

O(log n)

We usually write O(log n) with no subscript.

Theorem: f(n) is $O(\log_2 n)$ iff f(n) is $O(\log_k n)$ for any constant $k \geq 2$.

Proof:

Efficiency of implementing Table using a resizable array

Re-examine worst case:

method	unordered vector		ordered vector	
	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.
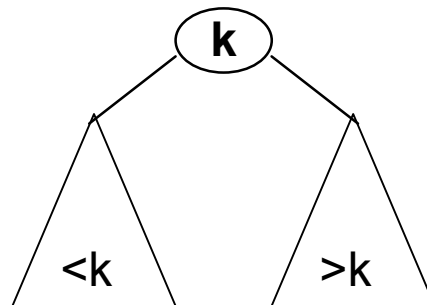
- *Note*: Binary search tree for a given set not unique
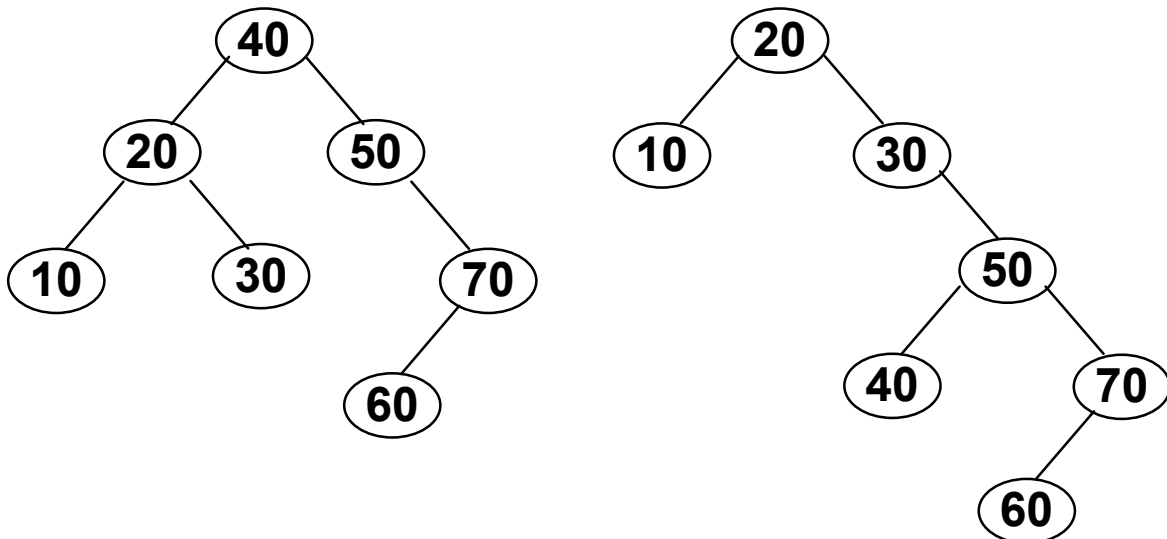
Binary search trees as implementations of tables

- labels represent associations (or just keys if no associated values)

- simple code for retrieving an item:

```
public TableBSTBased implements TableInterface{
    BinaryTree table;

    ...

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

protected static BinaryTree locate
                (Comparable searchKey, BinaryTree 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

  What should be done for duplicate keys?

- Deletion

  Postcondition can be expressed by cases.

  1) Not present: no change to the tree

  2) Else value to delete found in leaf: that leaf deleted

  3) Else value to delete found in node having one empty subtree: that node deleted and other subtree attached to the parent

  4) Else value to delete found in node having two non-empty subtrees: that node contains the value previously found in its predecessor (or successor) node and that other node deleted (using either case 2 or 3 as appropriate)

Sorting

- goals for studying sorting:
  - "common knowledge" in computer science
  - wide variety of possible approaches
  - practice in the design and analysis of algorithms

- assumptions:
  - All the data can fit in memory.
  - Data is all comparable and stored in array of size n.
  - Sort methods are `public static void`

  `sort(Comparable[] theArray, int n)` or
  `sort(Comparable[] theArray, int start, int end)`

- 2 important operations affecting time:
  - *comparisons*: comparing values of two data items
  - *data movements*: moving or copying a data item
  Ignore operations on index values, etc.
  Rationale:
    * data access and manipulation may be expensive for large objects
    * number of other operations executed between comparisons and data movements is bounded by a constant

- 1 important factor affecting space:
  - *amount of auxiliary storage*: Always need O(n) space to hold the data itself, but how much *other* space is needed?

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++) {
```
// inv: 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--) {
```
// inv: 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;
    }
}
```

Correctness of selection sort

- Convince ourselves and others that precondition + execute(method body) $\Rightarrow$ postcondition

- Examine preconditions, postconditions, and loop invariants
  - *loop invariant*: a major form of assertion
  - at the top of a loop: true on *every* iteration of the loop
  - constraints on loop variables and progress towards goal

- For indexOfLargest, loop picks out the largest element not yet included in sorted part

<p	p	unsorted

- For main loop, largest elements have been identified and they are in sorted order

Efficiency of selection sort

- How much auxiliary space is needed?

- What is the running time?

  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

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 392-395

• What is the running time and space?

• What is an appropriate loop invariant?

A recursive sorting algorithm: mergesort

- **Idea:** merge results of applying mergesort to both halves of the data

  see Carrano & Prichard, pp 395-400

- **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 =**

Mergesort itself

- ## Mergesort uses "divide and conquer"
  - – divide a large problem into smaller problems
  - – solve the smaller problems
  - – put the solutions together to form an answer to the larger problem

- ## In mergesort,
  - – smaller problems: sorting two half arrays
    - ∗ to be solved recursively
  - – put the solutions of those two problems together using merge

- ## Code outline:

```
void mergesort(d[0..n-1]) {
  if (n>1) {
    mergesort(d[0..n/2-1]);
    // elements in first half of d now in increasing order
    mergesort(d[n/2..n-1]);
    // elements in second half of d now in increasing order
    merge(d[0..n/2-1], d[n/2..n-1]);
    // all elements of d now in increasing order
  }
}
```

Analysis of mergesort

- If n is a power of 2, the "tree of problems to solve" looks like:
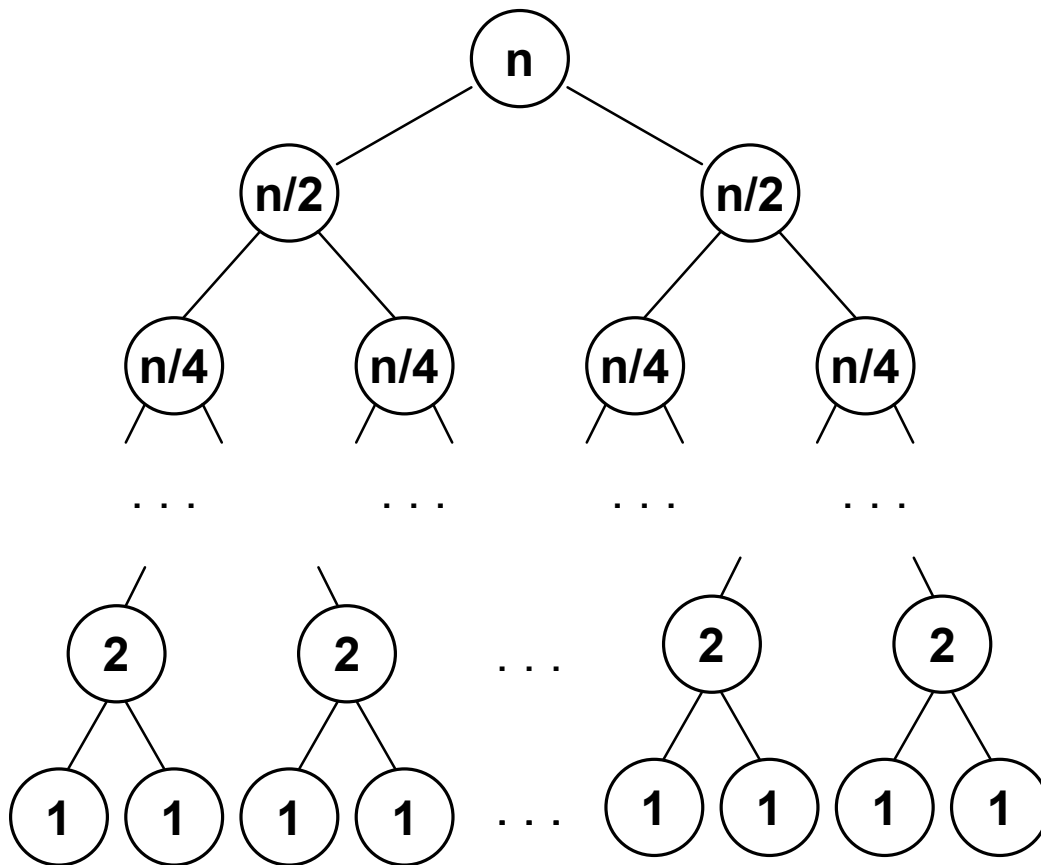
where labels represent sizes of the problems to solve

- Runtime =

- Auxiliary space =

Quicksort

- Idea:

1. *partition*: pick some *pivot* element and place it where it belongs;
2. sort all elements less than the pivot;
3. sort all elements greater than the pivot

see Carrano & Prichard, pp 400-412

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

- How could you code partition so it needs only O(1) auxiliary storage?

  1. We need to select a partitioning value.

  2. We need to move larger elements to the end of theArray and smaller elements to the beginning of theArray.

  3. We'll need a loop to compare elements in theArray to the partitioning value.

     – What should we use as a loop invariant?

p	< p	$\geq$ p	?

first   lastS1   firstUnknown   last
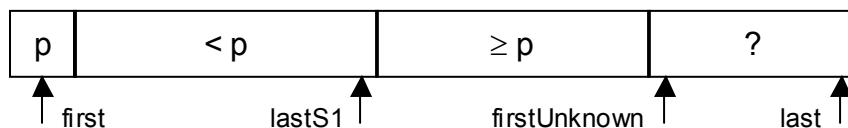
Analysis of quicksort (time)

- 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 using material from Stats 230.

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

- Encompasses hardware, software, theory, methods, applications
  - George Forsyth, Alan Perlis

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