CS 315: Algorithms and Data Structures

Gordon S. Novak Jr.

Department of Computer Sciences
University of Texas at Austin
`novak@cs.utexas.edu`
`http://www.cs.utexas.edu/users/novak`

"When you tell women you're really good at algorithms, it doesn't do much." – Aaron Patzer[1]

[1] "whose skill at algorithms helped him create Mint.com, the online personal finance tool, which he recently sold to Quicken for $170 million." – New York Times, Dec. 6, 2009.

[2] Acknowledgment: Many figures in these notes are taken from Wikipedia.
MapReduce slides are from `code.google.com/edu/parallel/index.html` .
All are licensed under `creativecommons.org/licenses/by/2.5/`.

Course Topics

- Introduction: why we *need* clever algorithms and data structures.
- Performance and Big O: how much time and storage is needed as data sizes become large
- Some Lisp: `(+ x 3)`
- Lists and Recursion; Stacks and Queues
- Trees and Tree Recursion
- Using Library Packages
- Balanced Trees and Maps
- Hashing and randomization; `XOR`
- Priority Queues and Heaps
- Sorting, Merge
- Graphs
- Map, Reduce, and MapReduce / Hadoop
- Data-intensive and parallel computation

Introduction

In this course, we will be interested in *performance*, especially as the size of the problem increases.

> "An engineer can do for a dime
> what any fool can do for a dollar."

Thus, $engineer/fool \cong 10$.

eine kleine LispMusik

We will learn a bit of Lisp in this course:

- Parentheses go outside a function call: `(sqrt 2.0)`
- Everything is a function call: `(+ x 3)`
- `defun` **de**fines a **fun**ction:
  `(defun twice (x) (* 2 x))`
- Every function returns a value:

```
(defun abs (x)
  (if (< x 0)
      (- x)
      x ) )
```

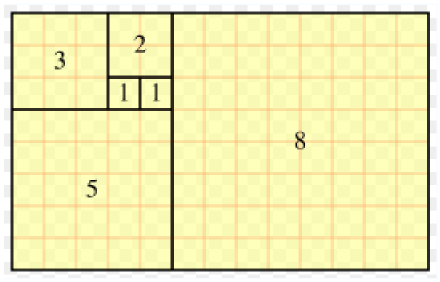
- `setq` is like `=` : `(setq i 3)`

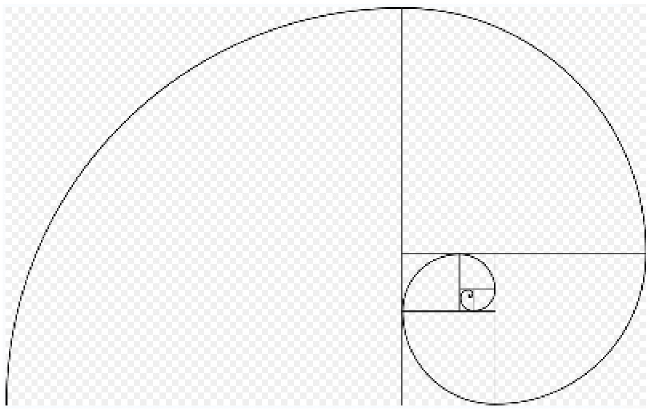
Fibonacci Numbers

Leonardo of Pisa, known as Fibonacci, introduced this series to Western mathematics in 1202:

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n-2) + F(n-1) \text{ , where } n > 1 \end{aligned}$$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610





The ratio between successive Fibonacci numbers approaches the *golden ratio* $\varphi = (1+\sqrt{5})/2 = 1.618034\ldots$.

Fibonacci Functions

Let's look at two versions of this function:

```
(defun fib2 (n)           public int fib2(int n) {
  (if (< n 2)                if (n < 2)
      n                         return n;
                                else return
      (+ (fib2 (- n 2))            fib2(n - 2) +
         (fib2 (- n 1)) ) ) )      fib2(n - 1);  }
```

```
(defun fib1 (n) (fib1b 0 1 n))

(defun fib1b (lo hi steps)
  (if (<= steps 0)
      lo
      (fib1b hi (+ lo hi) (- steps 1)) ) )
```

```
public int fib1(int n) { return fib1b(0, 1, n); }

public int fib1b(int lo, int hi, int steps) {
  if (steps <= 0)
     return lo;
     else return fib1b(hi, (lo + hi), steps - 1); }
```

Testing Fibonacci

```
>(fib1 8)
21

>(fib2 8)
21

>(fib1 20)
6765

>(fib2 20)
6765
```

`fib1` and `fib2` appear to compute the correct result.

```
>(fib1 200)

280571172992510140037611932413038677189525

>(fib2 200)          . . .
```

While $engineer/fool \cong 10$,
we see that $cs\ major/fool \rightarrow \infty$

Rates of Growth

In Greek mythology, the *Hydra* was a many-headed monster with an unfortunate property: every time you cut off a head, the Hydra grew two more in its place.

One of our Fibonacci functions is like the Hydra: each call to `fib2` may generate two more calls to `fib2`.

Like Hercules, our task is to slay the monster by avoiding excessive rates of growth.

Exponential Growth

"Know your enemy." – Sun Tzu, *The Art of War*

We want to be able to recognize problems or computations that will be *intractable*, impossible to solve in a reasonable amount of time.

Exponential growth of computation has several names in computer science:

- combinatoric explosion
- NP-complete
- $O(2^n)$

In real life, things that involve exponential growth usually end badly:

- Atomic bombs
- Cancer
- Population explosion

Goals of the Course

- Understand the tools of the trade:
  - Data Structures that allow certain operations on the data to be done efficiently.
  - Algorithms that perform operations and maintain the data structure.
- Gain experience in using library packages
  - the buy-versus-build decision
- Understand performance:
  - Big O: mathematical shorthand for rate of growth.
  - How to analyze a problem, design a solution, and estimate the performance.
  - How to analyze an existing design or program, spot the performance problems, and fix them.
- Gain experience in other programming paradigms:
  - Lisp: a notation, as well as a language
  - Patterns or Rewrite Rules
  - MapReduce, Hadoop and parallel data-intensive programming

Big O and Performance

We are very interested in *performance* of algorithms and how the cost of an algorithm increases as the size of the problem increases. Cost can include:

- Time required
- Space (memory) required
- Network bandwidth
- Other costs

The terms *efficiency* and *complexity* are also used.

Big O

Big O (often pronounced *order*) is an abstract function that describes how fast a function grows as the size of the problem becomes large. The order given by Big O is a least upper bound on the rate of growth.

We say that a function $T(n)$ has order $O(f(n))$ if there exist positive constants c and n_0 such that:
$T(n) \leq c * f(n)$ when $n \geq n_0$.

For example, the function $T(n) = 2 * n^2 + 5 * n + 100$ has order $O(n^2)$ because $2 * n^2 + 5 * n + 100 \leq 3 * n^2$ for $n \geq 13$.

We don't care about performance for small values of n, since small problems are usually easy to solve. In fact, we can make that a rule:

Rule: Any algorithm is okay if we know that the size of the input is small. In such cases, the simplest (easiest to code) algorithm is often best.

Rules for Big O

There are several rules that make it easy to find the order of a function:

- Any constant multipliers are dropped.
- The order of a sum is the maximum of the orders of its summands.
- A higher power of n beats a lower power of n.
- n beats $log(n)$.
- 2^n beats any power of n.

For example, in analyzing $T(n) = 2 * n^2 + 5 * n + 100$, we first drop all the constant multipliers to get $n^2 + n + 1$ and then drop the lower-order terms to get $O(n^2)$.

Classes of Algorithms

$f(n)$	Name	Example
1	Constant	$+$
$log(n)$	Logarithmic	binary search
$log^2(n)$	Log-squared	
n	Linear	max of array
$n * log(n)$	Linearithmic	quicksort
n^2	Quadratic	selection sort
n^3	Cubic	matrix multiply
n^k	Polynomial	
2^n	Exponential	knapsack problem

When we use $log(n)$, we often mean $log_2(n)$; however, the base of the $log(n)$ does not matter:

$$log_k(n) = log(n)/log(k)$$

Thus, a $log(n)$ is converted to another base by multiplying by a constant; but we eliminate constants when determining Big O.

Log n Grows Slowly

The function $log(n)$ grows so slowly that it can almost be considered to be the same as 1.

A good rule: $log_2(1000) \cong 10$, since $2^{10} = 1024$.
10 bits $\cong$ 3 decimal digits.

	n	$log_2(n)$
students in CS 315	120	7
students at UT	50,000	16
people in US	300,000,000	28
people on earth	7,000,000,000	33
national debt	11,700,000,000,000	44
Library of Congress, bytes	$20 * 10^{12}$	45
earth surface area, mm^2	$5 * 10^{20}$	69
atoms in universe	10^{80}	266

Thus, we can say that $log_2(n) < 300$ for any problem that we are likely to see. If we simply wrote 300 instead of $log_2(n)$, our rule tells us to eliminate the constant 300.

Powers of 10: SI Prefixes

Prefix	**Symbol**	10^n	**Prefix**	**Symbol**	2^k	**Word**
Yotta	Y	10^{24}	Yobi	Yi	2^{80}	
Zetta	Z	10^{21}	Zebi	Zi	2^{70}	
Exa	E	10^{18}	Exbi	Ei	2^{60}	
Peta	P	10^{15}	Pebi	Pi	2^{50}	
Tera	T	10^{12}	Tebi	Ti	2^{40}	trillion
Giga	G	10^{9}	Gibi	Gi	2^{30}	billion
Mega	M	10^{6}	Mebi	Mi	2^{20}	million
Kilo	K	10^{3}	Kibi	Ki	2^{10}	thousand
milli	m	10^{-3}				
micro	μ	10^{-6}				
nano	n	10^{-9}				
pico	p	10^{-12}				
femto	f	10^{-15}				
atto	a	10^{-18}				
zepto	z	10^{-21}				
yocto	y	10^{-24}				

A Scientist Should Be Careful, Skeptical

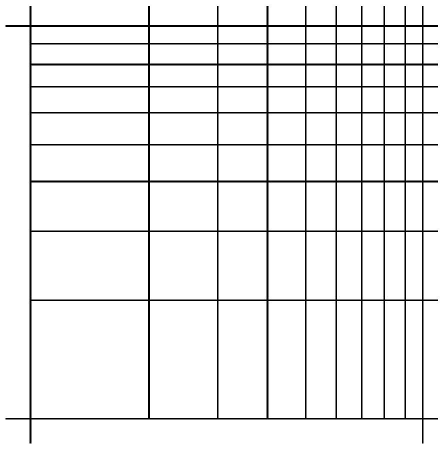
- Is the input data good? *Garbage in, garbage out!*
  - Is the data source authoritative? (*Don't trust it!*)
  - Was the data measured accurately?
  - How much noise does the data have?
  - Was the data recorded accurately?
  - Does the data mean what it is alleged to mean?
- Is the algorithm correct?
- Is the algorithm an accurate model of reality?
- Are the answers numerically accurate?
- Is the data represented accurately?
  - 32-bit `int`: 9 decimal digits
  - 64-bit `long`: 19 decimal digits
  - 32-bit `float`: 7 decimal digits (*avoid!*)
    Patriot missile: inaccurate float conversion caused range gate error: 28 killed, 98 injured.
  - 64-bit `double`: 15+ decimal digits
  - Ariane 5: 64-bit `double` converted to 16-bit int: overflowed, rocket exploded, $500 million loss.

Finding Big O: Log-log Graph

There are two ways to find the order of a computation:

- **Analytically** before developing software.
- **Experimentally** with existing software.
  - Log-log graph
  - Time ratios

A plot of time versus n on a log-log graph allows Big O to be found directly. Polynomial algorithms show up as straight lines on log-log graphs, and the slope of the line gives the order of the polynomial.

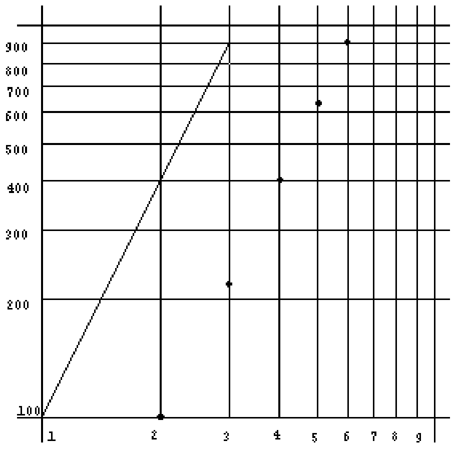


Log-log Example

Suppose that $f(n) = 25 * n^2$.

n	$f(n)$
2	100
3	225
4	400
5	625
6	900

The log-log graph makes it obvious that $f(n)$ is $O(n^2)$.



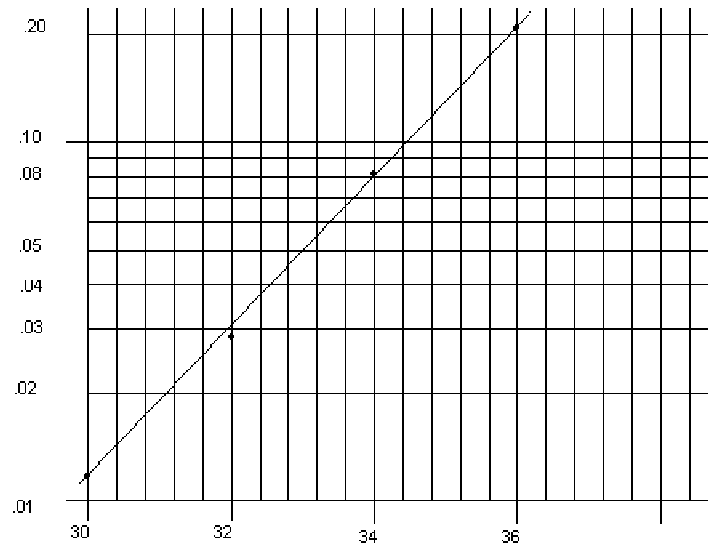
The points that are plotted from the function lie on a straight line that is parallel to a known $O(n^2)$ line.

Finding Big O: Semi-Log Graph

Functions whose Big O is exponential, $O(2^n)$, grow so quickly that they can hardly be plotted on a log-log graph.

If you find that your function cannot be plotted on a log-log graph, try plotting it on a semi-log graph, where the x axis is linear and the y axis is logarithmic.

An exponential function will produce a linear plot on a semi-log graph. Such functions are considered to be *intractable*, i.e. thay can only be computed when n is relatively small.



Big O from Timing Ratio

Another method that can often be used to find Big O is to look at the *ratio* of times as the size of input is doubled. Ratios that are approximately powers of 2 are easily recognized:

Ratio	**Big O**
2	n
2+	$n * log(n)$?
4	n^2
8	n^3

Example:

n	**Time**	**Ratio**
4000	0.01779	
8000	0.06901	3.8785
16000	0.27598	3.9991
32000	1.10892	4.0180
64000	4.44222	4.0059

Since the ratio is about 4, this function is $O(n^2)$.

Computation Model

We will assume that basic operations take constant time.

Later, we will consider some exceptions to this assumption:

- Some library functions may take more than O(1) time. We will want to be conscious of the Big O of library functions.
- Memory access may be affected by paging and cache behavior.
- There may be hidden costs such as garbage collection.

Big O from Code

It is often easy to determine Big O directly from code:

- Elementary operations such as `+` or `=` are $O(1)$.
- The Big O for a sequence of statements is the *max* of the Big O of the statements.
- The Big O for an `if` statement is the *max* of the Big O of the test, then statement, and else statement.
- The Big O for a loop is the loop count times the Big O of the contents.
- Big O for a recursive call that cuts the problem size in half is:
  - discard one half: $log(n)$ (binary search).
  - process both halves: $n \cdot log(n)$ (quicksort).

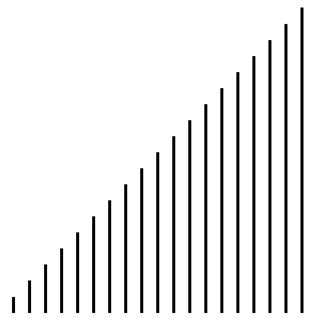
```
for ( i = 0; i < n; i++ )
  for ( j = 0; j < n; j++ )
    sum += a[i][j];
```

We all know this is $O(n^2)$; but what about:

```
for ( i = 0; i < n; i++ )
  for ( j = 0; j <= i; j++ )
    sum += a[i][j];
```

Beware the Bermuda Triangle

Some gym teachers punish misbehaving students by making them run 50 yards. However, the way in which they must do it is to run 1 yard, then come back, then run 2 yards, then come back, ...



How many total yards does a student run?

$$\sum_{i=1}^{n} i = n * (n+1)/2 = O(n^2)$$

```
for ( i = 0; i < n; i++ )
  for ( j = 0; j <= i; j++ )
    sum += a[i][j];
```

Rule: If a computation of $O(i)$ is inside a loop where i is *growing* up to n, the total computation is $O(n^2)$.

Big O for Arrays

Arrays are *random access*, which means that access to any element has the same cost.

- `a[i]` is $O(1)$.
- Allocating a new array (not initializing the contents) could be essentially $O(1)$. However, since Java is type-safe, the new array must be initialized for its contents type, so creation of a **`new`** size n array costs $O(n)$.

  If the $O(n)$ cost of array creation is *amortized* over the n elements, the creation cost per element is $O(1)$.
- Arrays are rigid: they cannot be expanded. One must either:
  - Get a new larger array and copy the old contents into it, $O(n)$
  - Make the array over-sized, wasting some space.
- If an array is larger than main memory, the cost to access an element can be much larger due to disk paging.

Average Case vs. Worst Case

Sometimes we will be interested in several performance values:

- Average Case: a random or average input
- Typical Case: it might typically be the case that the input is “nearly sorted”.
- Worst Case: to guarantee performance even in the worst case of input data.

The worst-case figure gives the strongest guarantee, and it may be easier to analyze.

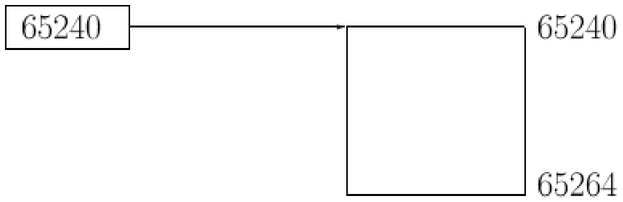
Familiarity with Big O

We want to be familiar with Big O for typical algorithms:

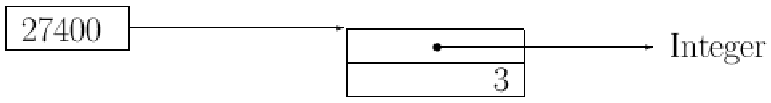
- Given a description of a task, guess its Big O. “First you sort the input, which is $n * log(n)$, then ...”
- Predict how well an algorithm will scale up. “Now that my new web service is the hottest thing in the CS Department, will it scale up to every college in the country so I can take it public and make my first billion dollars?”
- Given a program that is too slow, figure out what is wrong. “The sorting step is taking $O(n^2)$ time; that’s pathetic.”
- Select algorithms based on data sizes and Big O. “This set could be large, so we need $O(log(n))$ lookup ...”

Pointer / Reference

A *pointer* or *reference* is the memory address of the beginning of a *record* or block of storage, which extends through higher memory addresses. A pointer has the same size as `int`, 32 bits, which can address 4 Giga-bytes of storage.

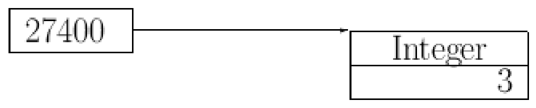


An *object* or *reference type* in a language such as Java is also a record, which has as its first item a pointer or other code that indicates its *class*.



As in this example, a record often contains other pointers. This is sometimes called a *boxed integer* .

Boxed Number



A *boxed number*, such as this instance of `Integer`, is nice because it can be treated as an object and has useful methods. However:

- `Integer` has the same numeric accuracy as an `int`, 32 bits.
- `Integer` takes 4 to 6 times as much storage as an `int`.
- `Integer` is *immutable*: since it is possible that multiple variables point to it, we cannot change its value.
- Doing arithmetic (even `++`) may require creation of a new box (allocation of new storage), often making the old box *garbage*. (Java has special techniques to make small Integer values more efficient.)
- Boxes are somewhat expensive because they must be *garbage collected*. The garbage collector is a highly paid worker.

References and ==

```
Integer i = 3;
Integer j = i;                      (i == j) = ?

Integer i = 127;
Integer j = 127;

Integer i = 130;
Integer j = 130;

Integer i = new Integer(100);
Integer j = new Integer(100);

Integer i = new Integer(100);
Integer j = 100;

Integer i = Integer.valueOf(100);
Integer j = 100;

Integer i = new Integer(99);
i++;
Integer j = 100;

Integer i = 127; i++;
Integer j = 127; j++;
```

`==` vs. `.equals()`

For reference types, including `Integer` etc., `==` means *equality of pointer values*, i.e. same data address in memory.

Rule: To test equality of the contents of a reference type, use `.equals()`.

Rule: To make an `Integer`, either use casting or `Integer.valueOf()`

- `new Integer()` always makes a new box, costing 16 bytes.
- `Integer.valueOf()` will reuse a value in the `Integer` cache if possible, costing 0 bytes.

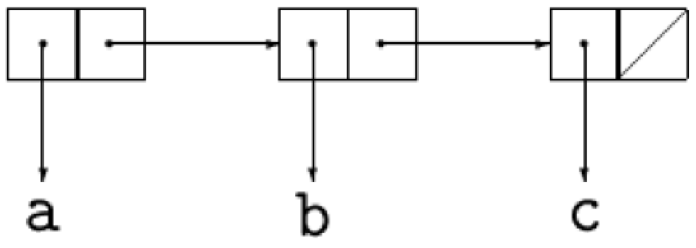
Linked List

A *linked list* is one of the simplest data structures. A linked list element is a record with two fields:

- a *link* or pointer to the next element (**`null`** in Java or **`nil`** in Lisp if there is no next element).
- some *contents*, as needed for the application. The contents could be a simple item such as a number, or it could be a pointer to another linked list, or multiple data items.

In Lisp, a list is written inside parentheses:

```
(list 'a 'b 'c)          ->  (a b c)
list("a", "b", "c")
```



```
(list (+ 2 3) (* 2 3))  ->  (5 6)
```

We will use parentheses and spaces as a *notation* for linked lists, in Java as well as in Lisp.

Constructing a Linked List

A linked list is usually made by linking new elements onto the front of the list. In Lisp, the system function `(cons` *item list*`)` makes a new list element containing *item* and adds it to the front of *list*:

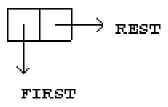
```
(cons 'a nil)                 ->  (a)
cons("a", null)

(cons 'a '())                 ->  (a)

(cons 'a '(b c))              ->  (a b c)
cons("a", list("b", "c"))
```

We can easily do the same thing in Java:

```
public class Cons  {
    private Object car;
    private Cons cdr;

    private Cons(Object first, Cons rest)
       { car = first;
         cdr = rest; }
    public static Cons
                  cons(Object first, Cons rest)
      { return new Cons(first, rest); }
```

Access to Parts of a List

The two fields of a `cons` cell are traditionally called `car` and `cdr`, but perhaps better names are `first`, the first item in the list, and `rest`, the link to the rest of the list.



```
(first '(a b c))                ->  a
first(list("a", "b", "c"))

(rest '(a b c))                 ->  (b c)
rest(list("a", "b", "c"))

public static Object first(Cons lst) {
    return ( (lst == null) ? null : lst.car  ); }

public static Cons rest(Cons lst) {
    return ( (lst == null) ? null : lst.cdr  ); }
```

Note that `first` and `rest` in general have different types; `first` can be any Object type, while `rest` is a `Cons` or `null`.

The functions `second` and `third` are also defined.

Iterative Processing of List

The basic operation on a list is to walk along it, processing one element at a time. We will illustrate the *design patterns* for doing this with the `length` function.

```
(length '(a b c))       ->  3

(defun length (lst)
  (let (n)                ; let declares variables
    (setq n 0)            ; setq is like =
    (while (consp lst)  ; is it a cons
      (setq n (+ n 1))
      (setq lst (rest lst)) )
    n ))

public static int length (Cons lst) {
  int n = 0;
  while ( lst != null ) {
    n++;
    lst = rest(lst); }
  return n; }

public static int length (Cons arg) {
  int n = 0;
  for (Cons lst=arg; lst != null; lst = rest(lst) )
      n++;
  return n; }
```

Iterative List Design Pattern

```
(defun fn (lst)
  (let (answer)
    initialize answer
    (while (consp lst)
      (setq answer
            (some-combination-of
              answer
              (something-about (first lst)) ) )
      (setq lst (rest lst)) )
    answer ))

public int fn (Cons lst) {
  initialize answer;
  for (Cons ptr = lst;
            ptr != null;
            ptr = rest(ptr) ) {
    answer = someCombinationOf(
               answer,
               somethingAbout( first(ptr) )); }
  return answer; }
```

Recursion

A *recursive* program calls itself as a subroutine. Recursion allows one to write programs that are powerful, yet simple and elegant. Often, a large problem can be handled by a small program which:

1. Tests for a *base case* and computes the value for this case directly.
2. Otherwise,
   (a) calls itself recursively to do *smaller* parts of the job,
   (b) computes the answer in terms of the answers to the smaller parts.

```
(defun factorial (n)
  (if (<= n 0)
      1
      (* n (factorial (- n 1))) ) )
```

Rule: Make sure that each recursive call involves an argument that is *strictly smaller* than the original; otherwise, the program can get into an *infinite loop*.

A good method is to use a counter or data whose size decreases with each call, and to stop at `0`; this is an example of a *well-founded ordering*.

Designing Recursive Functions

Some guidelines for designing recursive functions:

1. Write a clear definition of what your function should do. Write this definition as a comment above the function code.
2. Identify one or more *base cases*: simple inputs for which the answer is obvious and can be determined immediately.
3. Identify the *recursive case*: an input other than the base case. How can the answer be expressed in terms of the present input and the answer provided by this function (assuming it works as desired) for a smaller input?

   There are two common ways of making the input smaller:

   - Remove a small part of the input, e.g. remove the first element from a linked list.
   - Cut the input in half, e.g. follow one branch of a tree.

Design Pattern for Recursive Functions

A *design pattern* is an abstracted way of writing programs of a certain kind. By learning useful design patterns, you can write programs faster and with fewer errors.

A design pattern for recursive functions is:

```
(defun myfun  (arg)
  (if (basecase? arg)
      (baseanswer arg)
      (combine arg (myfun (smaller arg)))))
```

In this pattern,

- (*basecase*? *arg*) is a test to determine whether *arg* is a base case for which the answer is known at once.
- (*baseanswer arg*) is the known answer for the base case.
- (*combine arg* (*myfun* (*smaller arg*))) computes the answer in terms of the current argument *arg* and the result of calling the function recursively on (*smaller arg*), a reduced version of the argument.

Exercise: Show how the `factorial` function corresponds to this design pattern.

Recursive Processing of List

Recursive processing of a list is based on a *base case* (often an empty list), which usually has a simple answer, and a *recursive case*, whose answer is based on a recursive call to the same function on the rest of the list.

```
(defun length (lst)
  (if (null lst)       ; test for base case
      0                ; answer for base case
      (+ 1
         (length (rest lst))) ) )  ; recursive call

public static int length (Cons lst) {
  if ( lst == null )
     return 0;
   else return (1 +
                 length( rest(lst) )); }
```

Recursive List Design Pattern

```
(defun fn (lst)
  (if (null lst)       ; test for base case
      baseanswer       ; answer for base case
      (some-combination-of
         (something-about (first lst))
         (fn (rest lst))) ) )  ; recursive call

public int fn (Cons lst) {
  if ( lst == null )
     return baseanswer;
   else return someCombinationOf(
                  somethingAbout(first(lst)),
                  fn(rest(lst))); }
```

The recursive version is often short and elegant, but it has a potential pitfall: it requires $O(n)$ stack space on the function call stack. Most languages do not provide enough stack space for 1000 calls, but a linked list with 1000 elements is not unusual.

Tail Recursive Processing of List

A function is *tail recursive* if it either returns an answer directly, or the answer is exactly the result of a recursive call. Tail recursion often involves the use of an auxiliary function with extra variables as parameters.

```
(defun length (lst)
  (lengthb lst 0))          ; init extra variable

(defun lengthb (lst answer)
  (if (null lst)            ; test for base case
      answer                ; answer for base case
      (lengthb (rest lst)   ; recursive call
               (+ answer 1)) ) )   ; update answer

public static int length (Cons lst) {
  return lengthb(lst, 0); }

public static int lengthb (Cons lst, int answer) {
  if ( lst == null )
     return answer;
   else return lengthb(rest(lst), answer + 1); }
```

Tail Recursive List Design Pattern

```
(defun fn (lst)  (fnb lst answerinit))

(defun fnb (lst answer)
  (if (null lst)        ; test for base case
      answer            ; answer for base case
      (fnb (rest lst)
           (some-combination-of
             answer
             (something-about (first lst)))) ) )

public int fn (Cons lst) {
  return fnb(lst, answerinit); }

public static int fnb (Cons lst, answer) {
  if ( lst == null )
     return answer;
   else return fnb(rest(lst),
                   someCombinationOf(
                     answer,
                     somethingAbout(first(lst))));}
```

A smart compiler can detect a tail-recursive function and compile it so that it is iterative and uses $O(1)$ stack space.

Constructive Linked List: Reverse

`reverse` makes a new linked list whose elements are in the reverse order of the original list; the original is unchanged.

```
(reverse '(a b c))       ->  (c b a)
```

This function takes advantage of the fact that `cons` creates a list in the reverse order of the conses.

```
(defun reverse (lst)
  (let (answer)
    (setq answer '())
    (while (consp lst)
      (setq answer (cons (first lst) answer))
      (setq lst (rest lst)) )
    answer ))
```

```
public static Cons reverse (Cons lst) {
  Cons answer = null;
  while ( consp(lst) ) {
    answer = cons( first(lst), answer );
    lst = rest(lst); }
  return answer; }
```

Tail Recursive Reverse

```
(defun trrev (lst) (trrevb lst '()))

(defun trrevb (in out)
  (if (null in)
      out
      (trrevb (rest in)
              (cons (first in) out)) ) )
```

With a tail-recursive function, the unwinding of the recursion is all the same, so it can be compressed into one stack frame.

```
>(trrev '(a b c d))
  1> (TRREVB (A B C D) NIL)
    2> (TRREVB (B C D) (A))
      3> (TRREVB (C D) (B A))
        4> (TRREVB (D) (C B A))
          5> (TRREVB NIL (D C B A))
          <5 (TRREVB (D C B A))
        <4 (TRREVB (D C B A))
      <3 (TRREVB (D C B A))
    <2 (TRREVB (D C B A))
  <1 (TRREVB (D C B A))
(D C B A)
```

Tail Recursive Reverse in Java

```
public static Cons trrev (Cons lst) {
  return trrevb(lst, null); }

public static Cons trrevb (Cons in, Cons out) {
  if ( in == null )
     return out;
   else return trrevb( rest(in),
                       cons(first(in), out) ); }
```

Copying a List

Since **reverse** is constructive, we could copy a list by reversing it twice:

```
public static Cons copy_list (Cons lst) {
    return reverse(reverse(lst)); }
```

What is the Big O of this function?

We could criticize the efficiency of this function because it creates $O(n)$ garbage: the list produced by the first **reverse** is unused and becomes garbage when the function exits. However, the Big O of the function is still $O(n) + O(n) = O(n)$.

Append

append concatenates two lists to form a single list. The first argument is copied; the second argument is reused (shared).

```
(append '(a b c) '(d e))     ->  (a b c d e)

(defun append (x y)
  (if (null x)
      y
      (cons (first x)
            (append (rest x) y)) ) )

public static Cons append (Cons x, Cons y) {
  if (x == null)
     return y;
   else return cons(first(x),
                    append(rest(x), y)); }
```

This version of **append** append is simple and elegant, but it takes $O(n_x)$ stack space.

Iterative Append

An iterative version of `append` can copy the first list in a loop, using $O(1)$ stack space. For this function, it is convenient to use the `setrest` function:

```
public static Cons append (Cons x, Cons y) {
        Cons front = null;
        Cons back = null;
        Cons cell;
        if ( x == null ) return y;
        for ( ; x != null ; x = rest(x) ) {
             cell = cons(first(x), null);
             if ( front == null )
                 front = cell;
             else setrest(back, cell);
             back = cell; }
        setrest(back, y);
        return front; }
```

Destructive Linked List Functions

All of the functions we have considered so far are *constructive*: they may construct new lists, but they do not modify their arguments. However, these functions sometimes *share structure*, i.e. the same list structure is part of more than one list.

```
(setq x '(a b c))
(setq y '(d e))
(setq z (append x y))

x  ->  (a b c)
y  ->  (d e)
z  ->  (a b c d e)
```

Appending `x` and `y` to form `z` did not change `x` and `y`. However, `z` and `y` now share structure.

Functions that modify their arguments are sometimes called *destructive*; they are useful, but must be used with care to make sure that *shared* structures are not inadvertently modified. If we made a destructive change to `y`, it would also change `z`.

```
(setf (first y) 3)
z  ->  (a b c 3 e)
```

Nconc

`nconc` concatenates two lists to form a single list; instead of copying the first list as `append` does, `nconc` modifies the end of the first list to point to the second list.

```
(nconc (list 'a 'b 'c) '(d e))   ->  (a b c d e)

(defun nconc (x y)
  (let (ptr)
    (setq ptr x)
    (if (null x)
        y
        (progn              ; progn is like {  }
          (while (not (null (rest x)))
             (setq x (rest x)) )
          (setf (rest x) y)
          ptr) ) ))

public static Cons nconc (Cons x, Cons y) {
  Cons ptr = x;
  if (x == null)
     return y;
   else { while (rest(x) != null)
            x = rest(x);
          setrest(x, y);
          return ptr; } }
```

Nreverse

Destructive functions in Lisp often begin with `n`. `nreverse` reverses a list in place by turning the pointers around.

```
(nreverse (list 'a 'b 'c))       ->  (c b a)

(defun nreverse (lst)
  (let (last next)
    (setq last nil)
    (while (not (null lst))
      (setq next (rest lst))
      (setf (rest lst) last)
      (setq last lst)
      (setq lst next) )
    last))

public static Cons nreverse (Cons lst) {
  Cons last = null; Cons next;
  while (lst != null)
    { next =  rest(lst);
      setrest(lst, last);
      last = lst;
      lst = next; };
  return last; }
```

Set as Linked List

A linked list can be used as a representation of a *set*. `member` (written ∈) tests whether a given item is an element of the list. `member` returns the remainder of the list beginning with the desired element, although usually `member` is used as a *predicate* to test whether the element is present or not.

```
(member 'dick '(tom dick harry))   ->  (dick harry)

(member 'fred '(tom dick harry))   ->  nil

(defun member (item lst)
  (if (null lst)
      nil
      (if (eql item (first lst))
          lst
          (member item (rest lst)) ) ) )

public static Cons member (Object item, Cons lst) {
  if ( lst == null )
     return null;
   else if ( item.equals(first(lst)) )
           return lst;
         else return member(item, rest(lst)); }
```

Intersection

The *intersection* (written $\cap$) of two sets is the set of elements that are members of both sets.

```
(intersection '(a b c) '(a c e))   ->  (c a)

(defun intersection (x y)
  (if (null x)
      nil
      (if (member (first x) y)
          (cons (first x)
                (intersection (rest x) y))
          (intersection (rest x) y) ) ) )

public static Cons intersection (Cons x, Cons y) {
  if ( x == null )
     return null;
   else if ( member(first(x), y) != null )
           return
             cons(first(x),
                  intersection(rest(x), y));
         else return intersection(rest(x), y); }
```

If the sizes of the input lists are m and n, the time required is $O(m \cdot n)$. That is not very good; this version of `intersection` will only be acceptable for small lists.

Tail-Recursive Intersection

```
(defun intersecttr (x y) (intersecttrb x y '()))

(defun intersecttrb (x y result)
  (if (null x)
      result
      (intersecttrb (rest x) y
         (if (member (first x) y)
             (cons (first x) result)
             result))))

>(intersecttr '(a b c) '(a c e))

  1> (INTERSECTTR (A B C) (A C E))
    2> (INTERSECTTRB (A B C) (A C E) NIL)
      3> (INTERSECTTRB (B C) (A C E) (A))
        4> (INTERSECTTRB (C) (A C E) (A))
          5> (INTERSECTTRB NIL (A C E) (C A))
          <5 (INTERSECTTRB (C A))
        <4 (INTERSECTTRB (C A))
      <3 (INTERSECTTRB (C A))
    <2 (INTERSECTTRB (C A))
  <1 (INTERSECTTR (C A))
(C A)
```

Tail-Recursive Intersection in Java

```
public static Cons intersecttr (Cons x, Cons y) {
  return intersecttrb(x, y, null); }

public static Cons intersecttrb (Cons x, Cons y,
                                 Cons result) {
  if ( x == null )
     return result;
   else return intersecttrb(rest(x), y,
                 ( member(first(x), y) != null )
                 ? cons(first(x), result)
                 : result); }
```

Union and Set Difference

The *union* (written $\cup$) of two sets is the set of elements that are members of either set.

```
(union '(a b c) '(a c e))   ->  (b a c e)
```

The *set difference* (written $-$) of two sets is the set of elements that are members of the first set but not the second set.

```
(set-difference '(a b c) '(a c e))   ->  (b)
```

Circularly Linked List

It is possible to have a linked list in which the last link points back to the front of the list rather than to `null` or `nil`. This is called a *circularly linked list.*

This may be convenient for data such as a list of vertices of a polygon, since it makes the data uniform: each vertex is followed by its successor.

Circularly linked lists must be used with care, since they make it easy to write a program that gets into an infinite loop.

A *doubly linked list* has link pointers that point both forward and backward. This makes it easy to insert or remove items at either end. The Java library uses doubly linked lists.

Merge

To *merge* two sorted lists means to combine them into a single list so that the combined list is sorted. We will consider both constructive and destructive versions of `merge`.

The general idea of a merge is to walk down *two* sorted lists simultaneously, advancing down one or the other based on comparison of the top values using the sorting function.

In combining two sorted lists with a merge, we walk down both lists, putting the smaller value into the output at each step.

```
(merge 'list '(3 7 9) '(1 2 4) '<)

            ->  (1 2 3 4 7 9)
```

For simplicity, we will define a function `merj` that assumes a list of comparable objects and an ascending sort.

```
(merj '(3 7 9) '(1 2 4))   ->  (1 2 3 4 7 9)
```

Constructive Merge

The easiest way to understand this idiom is a simple merge that constructs a new list as output.

```
(defun merj (x y)
  (if (null x)
      y
      (if (null y)
          x
          (if (< (first x) (first y))
              (cons (first x)
                    (merj (rest x) y))
              (cons (first y)
                    (merj x (rest y))) ) ) ) )

public static Cons merj (Cons x, Cons y) {
  if ( x == null )
     return y;
   else if ( y == null )
           return x;
  else if ( ((Comparable) first(x))
                       .compareTo(first(y)) < 0 )
           return cons(first(x),
                       merj(rest(x), y));
           else return cons(first(y),
                            merj(x, rest(y))); }
```

What is $O()$? Stack depth? Conses?

Tail Recursive Merge

```
(defun merjtr (x y) (nreverse (merjtrb x y '())))
(defun merjtrb (x y result)
 (if (null x)
   (if (null y)
       result
       (merjtrb x (rest y) (cons (first y) result))
   (if (or (null y)
           (< (first x) (first y)))
       (merjtrb (rest x) y (cons (first x) result))
       (merjtrb x (rest y) (cons (first y) result))
```

```
 1> (MERJTR (3 7 9) (1 2 4))
   2> (MERJTRB (3 7 9) (1 2 4) NIL)
     3> (MERJTRB (3 7 9) (2 4) (1))
       4> (MERJTRB (3 7 9) (4) (2 1))
         5> (MERJTRB (7 9) (4) (3 2 1))
           6> (MERJTRB (7 9) NIL (4 3 2 1))
             7> (MERJTRB (9) NIL (7 4 3 2 1))
               8> (MERJTRB NIL NIL (9 7 4 3 2 1))
               <8 (MERJTRB (9 7 4 3 2 1))
             <7 (MERJTRB (9 7 4 3 2 1))
           <6 (MERJTRB (9 7 4 3 2 1))
         <5 (MERJTRB (9 7 4 3 2 1))
       <4 (MERJTRB (9 7 4 3 2 1))
     <3 (MERJTRB (9 7 4 3 2 1))
   <2 (MERJTRB (9 7 4 3 2 1))
 <1 (MERJTR (1 2 3 4 7 9))
```

Tail Recursive Merge in Java

```
public static Cons merjtr (Cons x, Cons y) {
    return nreverse(merjtrb(x, y, null)); }

public static Cons merjtrb (Cons x, Cons y,
                            Cons result) {
 if ( x == null )
   if ( y == null )
      return result;
      else return merjtrb(x, rest(y),
                          cons(first(y), result));
 else if ( ( y == null ) ||
           ((Comparable) first(x))
                      .compareTo(first(y)) < 0 )
           return merjtrb(rest(x), y,
                          cons(first(x), result));
      else return merjtrb(x, rest(y),
                          cons(first(y), result));}
```

Destructive Merge Function

```
(defun dmerjr (x y)
  (if (null x)
      y
      (if (null y)
          x
          (if (< (first x) (first y))
              (progn (setf (rest x)
                           (dmerjr (rest x) y))
                     x)
              (progn (setf (rest y)
                           (dmerjr  x (rest y)))
                     y) ) ) ) )

public static Cons dmerjr (Cons x, Cons y) {
  if ( x == null )
     return y;
   else if ( y == null )
           return x;
  else if ( ((Comparable) first(x))
                        .compareTo(first(y)) < 0 )
          { setrest(x, dmerjr(rest(x), y));
            return x; }
        else { setrest(y, dmerjr(x, rest(y)));
               return y; } }
```

Iterative Destructive Merge

```
(defun dmerj (x y)
  (let (front end)
    (if (null x)
        y
        (if (null y)
            x
            (progn
              (if (< (first x) (first y))
                  (progn (setq front x)
                         (setq x (rest x)))
                  (progn (setq front y)
                         (setq y (rest y))))
              (setq end front)
              (while (not (null x))
                (if (or (null y)
                        (< (first x) (first y)))
                    (progn (setf (rest end) x)
                           (setq x (rest x)))
                    (progn (setf (rest end) y)
                           (setq y (rest y))))
                (setq end (rest end)) )
              (setf (rest end) y)
              front))) ))
```

Java Iterative Destructive Merge

```
public static Cons dmerj (Cons x, Cons y) {
  if ( x == null ) return y;
   else if ( y == null ) return x;
   else { Cons front = x;
          if ( ((Comparable) first(x))
                          .compareTo(first(y)) < 0)
             x = rest(x);
            else { front = y;
                   y = rest(y); };
          Cons end = front;
          while ( x != null )
            { if ( y == null ||
                   ((Comparable) first(x))
                          .compareTo(first(y)) < 0)
                 { setrest(end, x);
                   x = rest(x); }
                else { setrest(end, y);
                      y = rest(y); };
              end = rest(end); }
          setrest(end, y);
          return front; } }
```

What is $O()$? Stack depth? Conses?

Divide and Conquer

We can make a sorting routine using merge. *Divide and conquer* is a technique for solving a large problem by breaking it into two smaller problems, until the problems become easy. If we cut the problem in half each time, the solution will be $O(log(n))$ or $O(n \cdot log(n))$.

- If divide-and-conquer cuts the problem in half at each step, the problem size will decrease very fast: by a factor of 1,000 in 10 steps, a factor of 1,000,000 in 20 steps, a factor of 1,000,000,000 in 30 steps.
- We soon reach a problem of size 1, which is usually easy to solve.
- Time will be $O(log(n))$ if we only process one of the halves, as in binary search.
- Time will be $O(n \cdot log(n))$ if we process both halves, as in sorting.

Dividing a List

We can find the midpoint of a list by keeping two pointers, moving one by two steps and another by one step, $O(n)$.

```
(defun midpoint (lst)
  (let (prev current)
    (setq current lst)
    (setq prev lst)
    (while (and (not (null lst))
                (not (null (rest lst))))
      (setq lst (rest (rest lst)))
      (setq prev current)
      (setq current (rest current)) )
    prev))
```

```
public static Cons midpoint (Cons lst) {
  Cons current = lst;
  Cons prev = current;
  while ( lst != null && rest(lst) != null) {
    lst = rest(rest(lst));
    prev = current;
    current = rest(current); };
  return prev; }
```

Sorting by Merge

A list of length 0 or 1 is already sorted. Otherwise, break the list in half, sort the halves, and merge them.

```
(defun llmergesort (lst)
  (let (mid half)
    (if (or (null lst) (null (rest lst)))
        lst
        (progn (setq mid (midpoint lst))
               (setq half (rest mid))
               (setf (rest mid) nil)
               (dmerj (llmergesort lst)
                      (llmergesort half)) ) )) ))
```

```
public static Cons llmergesort (Cons lst) {
  if ( lst == null || rest(lst) == null)
     return lst;
   else { Cons mid = midpoint(lst);
          Cons half = rest(mid);
          setrest(mid, null);
          return dmerj( llmergesort(lst),
                        llmergesort(half)); } }
```

What is $O()$? Stack depth? Conses?

Tracing Sort and Merge

```
(trace merj llmergesort)

(llmergesort '(39 84 48 59 86 32))

  1> (LLMERGESORT (39 84 48 59 86 32))
    2> (LLMERGESORT (39 84 48))
      3> (LLMERGESORT (39))
      <3 (LLMERGESORT (39))
      3> (LLMERGESORT (84 48))
        4> (LLMERGESORT (84))
        4> (LLMERGESORT (48))
        4> (MERJ (84) (48))
        <4 (MERJ (48 84))
      <3 (LLMERGESORT (48 84))
      3> (MERJ (39) (48 84))
      <3 (MERJ (39 48 84))
    <2 (LLMERGESORT (39 48 84))
    2> (LLMERGESORT (59 86 32))
      3> (LLMERGESORT (59))
      <3 (LLMERGESORT (59))
      3> (LLMERGESORT (86 32))
        4> (LLMERGESORT (86))
        4> (LLMERGESORT (32))
        4> (MERJ (86) (32))
        <4 (MERJ (32 86))
      <3 (LLMERGESORT (32 86))
      3> (MERJ (59) (32 86))
      <3 (MERJ (32 59 86))
    <2 (LLMERGESORT (32 59 86))
    2> (MERJ (39 48 84) (32 59 86))
    <2 (MERJ (32 39 48 59 84 86))
  <1 (LLMERGESORT (32 39 48 59 84 86))
(32 39 48 59 84 86)
```

On Not Dropping the Ball

`llmergesort` is a *destructive* sorting function that will change the order, but not the location or contents, of list elements.

Suppose that `lst` is `(b c a)` and we perform `llmergesort(lst)`. If we now print out `lst`, it looks like `(b c)`. Has part of the list been lost?

What we need to do is to perform an assignment to save the result of the sort.

```
   lst = llmergesort(lst);
```

This will save the result of sorting. We don't want the old value of `lst` any longer since it now points into the middle of the sorted list.

Rule: Save the result of a function by doing an assignment to a variable. If the function is destructive, use the same variable name as the original, since what it pointed to before is no longer meaningful.

Divide and Conquer Design Pattern

```
(defun myfun (problem)
  (if (basecase? problem)
      (baseanswer problem)
      (combine (myfun (firsthalf problem))
               (myfun (secondhalf problem)))
))
```

```
(defun llmergesort (lst)
  (let (mid half)
    (if (or (null lst) (null (rest lst)))
        lst
        (progn (setq mid (midpoint lst))
               (setq half (rest mid))
               (setf (rest mid) nil)
               (dmerj (llmergesort lst)
                      (llmergesort half)) ) )) ))
```

Intersection by Merge

```
(defun intersection (x y)
  (mergeint (llmergesort x) (llmergesort y)) )
(defun mergeint (x y)
  (if (or (null x) (null y))
      nil
      (if (= (first x) (first y))
          (cons (first x)
                (mergeint (rest x) (rest y)))
          (if (< (first x) (first y))
              (mergeint (rest x) y)
              (mergeint x (rest y)) ) ) ) )

public static Cons intersectionm (Cons x, Cons y) {
  return mergeint(llmergesort(x), llmergesort(y));}

public static Cons mergeint (Cons x, Cons y) {
  if ( x == null || y == null ) return null;
   else if ( first(x).equals(first(y)) )
           return cons(first(x),
                       mergeint(rest(x),rest(y)));
  else if ( ((Comparable) first(x))
                         .compareTo(first(y)) < 0)
      return mergeint(rest(x), y);
  else return mergeint(x, rest(y));}
```

What is $O()$? Stack depth? Conses?

Remember to Merge

A program that sorts and then merges is a good way to perform many kinds of set operations in $O(n \cdot log(n))$ time.

A merge can just as easily be performed with arrays using two array indices as pointers.

Association List

An *association list* or *alist* is a simple lookup table or *map*: a linked list containing a key value and some information associated with the key.

```
(assoc 'two '((one 1) (two 2) (three 3)))
    ->  (two 2)

(defun assoc (key lst)
  (if (null lst)
      nil
      (if (equal key (first (first lst)))
          (first lst)
          (assoc key (rest lst)) ) ) )

public static Cons assoc(Object key, Cons lst) {
  if ( lst == null )
     return null;
  else if ( key.equals(first((Cons) first(lst))) )
      return ((Cons) first(lst));
          else return assoc(key, rest(lst)); }
```

New items can be added to the association list using `cons`.

Adv: Simple code. Table is easily expanded.

Dis: $O(n)$ lookup: Suitable only for small tables.

Stack using Linked List

A *stack*, analogous to a stack of cafeteria plates, supports only two operations, `push` and `pop`, and a test `empty`. A stack is sometimes called a *LIFO queue*, for Last-In First-Out, because the last item pushed is the first item popped.

A linked list is a natural way to implement a stack. `cons` accomplishes `push`, and `first` and `rest` accomplish `pop`. Both `push` and `pop` are provided as macros in Lisp.

```
(setq stack (cons item stack))    ; push

(setq item (first stack))         ; pop
(setq stack (rest stack))

stack = cons(item, stack);        // push

item = first(stack);              // pop
stack = rest(stack);

(stack == null)                   // empty
```

Both `push` and `pop` are $O(1)$ and very fast.

`pop` of an empty stack is an error; this can cause a *null dereference* hardware trap or an exception.

Sentinel Node

The **push** and **pop** operations on a stack both have *side effects* on the pointer to the stack. If that pointer is a variable, we cannot write **push** and **pop** as subroutines. A common technique is to put an extra dummy node or *sentinel* at the front of the list; the sentinel node points to the actual list. Then we can write subroutines:

```
(defun pushb (sentinel item)
  (setf (rest sentinel)
        (cons item (rest sentinel)) )
  sentinel )

(defun popb (sentinel)
  (let (item)
    (setq item (first (rest sentinel)))
    (setf (rest sentinel) (rest (rest sentinel)))
    item ))

public static Cons
         pushb (Cons sentinel, Object item) {
  setrest(sentinel, cons(item,rest(sentinel)));
  return sentinel; }

public static Object popb (Cons sentinel) {
  Object item = first(rest(sentinel));
  setrest(sentinel, rest(rest(sentinel)));
  return item; }
```

Other Uses of Linked Lists

A linked list does not have to be in main memory.

For example, a file on the disk may be kept as a linked list of disk blocks. In this case, a reference or a link is a disk address.

This is the reason that a deleted file is not actually erased: the storage of the disk file is released (as a linked list of free disk blocks), but the contents of the disk blocks remains.

Arrays

An *array* is a contiguous group of elements, all of the same type, indexed by an integer. In a sense, the array is the most basic data structure, since the main memory of a computer is essentially one big array.

The major advantage of an array is *random access* or $O(1)$ access to any element. (Paging and cache behavior can add significantly to access time, but we will ignore that.)

The major disadvantage of an array is rigidity:

- an array cannot be expanded
- two arrays cannot be combined
- Storage may be wasted because an array is made larger so it will have some extra space.

In Java, an array is an Object, so it is possible to expand it in effect by making a new larger array, copying the old array contents into the new array, and letting the old array get garbage-collected.

Stack using Array

```
final class MyStack {
    private int top;
    private int [] stack;

    public MyStack(int size)
      { top = 0;
        stack = new int[size]; }

    public boolean empty() {
        return (top == 0); }

    public MyStack push(int val) {
        stack[top++] = val;
        return this; }

    public int pop() {
        int val = stack[--top];
        return val; }
}
```

Uses of Stacks

Stacks are used in many places in computer science:

- Most programming languages keep variable values on a *runtime stack*.
- The SPARC architecture has a *register file stack* in the CPU.
- Operating systems keep the state of interrupted processes on a stack.
- A web browser keeps a stack of previously visited web pages; this stack is popped when the Back button is clicked.
- Compilers use a stack when parsing programming languages.
- In general, a stack is needed to traverse a tree.

Recursion and Runtime Stack

The *runtime stack* keeps a fresh set of values for each variable in a *stack frame*. A new stack frame is pushed onto the stack each time a function is entered. Consider a function to compute the factorial function, written n!

```
   (defun fact (n)
     (if (= n 0)
         1
         (* n (fact (- n 1))) ) )

>(trace fact)

>(fact 3)

  1> (FACT 3)
    2> (FACT 2)
      3> (FACT 1)
        4> (FACT 0)
        <4 (FACT 1)
      <3 (FACT 1)
    <2 (FACT 2)
  <1 (FACT 6)
6
```

Recursive Function Execution

Consider the computation of `(fact 3)`. A new stack frame is created in which `n` = `3`:

`n` = `3`

Now we can execute the code of `fact`. We test `(if (<= n 0) ...)` and, since `n` = `3`, evaluate `(* n (fact (- n 1)))`. `n` evaluates to `3`, and then we evaluate `(fact (- n 1))` which is `(fact 2)`. This creates a new stack frame in which `n` = `2`:

`n` = `2`
`n` = `3`

Note that the older binding, `n` = `3`, has not gone away, but is now *shadowed* by a new binding `n` = `2`.

Now we test `(if (<= n 0) ...)` and call `(fact 1)`:

`n` = `1`
`n` = `2`
`n` = `3`

and then we call `(fact 0)`:

`n` = `0`
`n` = `1`
`n` = `2`
`n` = `3`

Recursive Execution ...

This time, our test `(if (<= n 0) ...)` is true and we return the value `1` as the value of `(fact 0)`:

`n` = `1`
`n` = `2`
`n` = `3`

Now the top stack frame has been popped off. We multiply `1` by the value of `(fact 0)` (also `1`) and return `1` as the value of `(fact 1)`.

Now `(fact 1)` = `1` and our stack looks like:

`n` = `2`
`n` = `3`

We multiply 2 by the value of `(fact 1)` and return `2` as the value of `(fact 2)`:

`n` = `3`

We multiply 3 by the value of `(fact 2)` and return `6` as the final value of `(fact 3)`. Now the stack is empty.

The `trace` printout two pages above shows the argument values and return values.

Recursive Length Function Execution

```
>(defun length (lst)
  (if (null lst)       ; test for base case
      0                ; answer for base case
      (+ 1
         (length (rest lst))) ) )  ; recursive call

>(length '(a b c))

  1> (LENGTH (A B C))
    2> (LENGTH (B C))
      3> (LENGTH (C))
        4> (LENGTH NIL)
        <4 (LENGTH 0)
      <3 (LENGTH 1)
    <2 (LENGTH 2)
  <1 (LENGTH 3)
3
```

Balancing Parentheses

A stack is a good way to test whether parentheses are balanced. An open paren must be matched by a close paren of the same kind, and whatever is between the parens must be balanced. `({[][][]})` is balanced, but `({[)` is not.

```
public class charStack {
    int n;
    char[] stack;
    public charStack()
        { n = 0;
          stack = new char[100]; }
    public void push(char c) {
        stack[n++] = c; }
    public char pop() {
        return stack[--n]; }
    public boolean empty() {
        return ( n == 0 ); }
```

This example illustrates that it is easy to roll your own stack.

Balance Test Using Stack

```
public static boolean testBalance(String s) {
   charStack stk = new charStack();
   boolean okay = true;
   for (int i=0; okay && (i < s.length()); i++)
    {  char c = s.charAt(i);
       switch ( c ) {
       case '(': case '{': case '[':
           stk.push(c);
           break;
       case ')':
           if ( stk.empty() ||
                ( stk.pop() != '(' ) )
               okay = false;
           break;
       // same for '}' and ']'
       default: break;   // accept other chars
       } }
       return (okay && stk.empty() ); }
```

Linked List Stack

We could just as easily use a linked list to implement the stack. The `testBalance` program is the same for either version. The stack *abstract data type* uses the same interface for both implementations.

```
public class charStack {
    charStack link;
    char contents;
    public void push(char c) {
        charStack newitem = new charStack();
        newitem.link = link;
        newitem.contents = c;
        link = newitem; }
    public char pop() {
        char c = link.contents;
        link = link.link;
        return c; }
    public boolean empty() {
        return ( link == null ); } }
```

This data structure is essentially the same as the one for `Cons`.

Tree Traversal and Stack

A *grammar* can be written to describe the *syntax* of the language of balanced parentheses:

$S \rightarrow$ (S)
$S \rightarrow$ [S]
$S \rightarrow$ { S }
$S \rightarrow SS$
$S \rightarrow \epsilon$

This grammar describes a balanced parenthesis string as a tree. As we check the syntax of a string, the stack always contains the unbalanced symbols between the current symbol and the root of the tree.

XML

XML , for *Extensible Markup Language*, allows users to put tags around their data to describe what pieces of the data mean.

```
<CATALOG>
  <CD>
    <TITLE>Empire Burlesque</TITLE>
    <ARTIST>Bob Dylan</ARTIST>
    <COUNTRY>USA</COUNTRY>
    <COMPANY>Columbia</COMPANY>
    <PRICE>10.90</PRICE>
    <YEAR>1985</YEAR>
  </CD>
  <CD> ... </CD>
</CATALOG>
```

We can see that XML provides a hierarchical tree structure for data. The task of checking the validity of an XML file is essentially the same as checking for balanced parentheses; XML simply allows the users to define their own parenthesis names.

Queues

A *queue* data structure implements waiting in line: items are inserted (*enqueued*) at the end of the queue and removed (*dequeued*) from the front. Sometimes the term *FIFO queue* or just *FIFO* is used, for First-In First-Out.

A queue is a *fair* data structure: an entry in the queue will eventually be removed and get service. (A stack, in contrast, is *unfair*.)

Queues are frequently used in operating systems:

- queues of processes that are ready to execute
- queues of jobs to be printed
- queues of packets that are ready to be transmitted over a network

Two Pointer Queue using Linked List

A linked list makes a good queue if we keep a pointer to the end of the list as well as the front.

```
public class MyQueue {
    private Cons front;
    private Cons end;
    public MyQueue() { front = null; }

    public MyQueue insert(Object val) {
      Cons element = Cons.list(val);
      if ( front == null )
         front = element;
        else Cons.setrest(end, element);
      end = element;
      return this; }

    public Object remove () {       // same as pop
        Object val = Cons.first(front);
        front = Cons.rest(front);
        return val; }
```

Circular Queue using Array

We can easily keep a queue of elements in an array. This is the same as a stack, except that we remove items from the other end.

We make the array *circular* by assuming that index `[0]` follows index `[n - 1]`.

```
public class CirQueue {
    private int [] queue;
    private int front;
    private int end;

    public CirQueue()
      { queue = new int[10];
        front = 0;
        end = 0; }

    public boolean empty()
    { return ( front == end ); }
```

Circular Queue Code

```
   public void insert(int val) {
     int next = (end + 1) % queue.length;
                              // wrap around
//      if ( next == front )
//         expand the array, or
//         throw new QueueFullException();
     queue[end] = val;
     end = next; }

   public int remove () {
       int val = queue[front];
       front = (front + 1) % queue.length;
                               // wrap around
       return val; }
```

Java Collection API

The Java `Collection` interface (in package `java.util`) provides a common way for various kinds of collections to be used. This may make it possible to change the kind of collection used by an application. It also allows use of Java iterators to iterate over the members of a collection.

```
public interface Collection<AnyType>
       extends Iterable<AnyType> {
int size();
boolean isEmpty();
void clear();
boolean contains ( AnyType x );
boolean add ( AnyType x );
boolean remove ( AnyType x );
java.util.Iterator<AnyType> iterator();

Object[] toArray();
<T> T[] toArray(T[] a);  }
```

The `toArray` method allows any Collection to be converted easily to an array.

Java Collections Iteration

Two iterator patterns can be used with collections. The first is a simple iteration through all elements:

```
for ( AnyType item : coll )
```

The second form allows more complicated processing, including removal of an item.

```
Iterator<AnyType> itr = coll.iterator();

while ( itr.hasNext() ) {
  AnyType item = itr.next();
  ...           // process item
  if ( ... )    itr.remove();   }
```

In this pattern, an iterator object `itr` is created. This object can be queried to test whether any items remain using `itr.hasNext()`. If there are objects, `itr.next()` gets the next one. `itr.remove()` removes the last object returned by `itr.next()` and can only be used once until there is another call to `itr.next()` .

A collection should not be modified while an iteration over it is in progress, except by the `itr.remove()` .

Filter Pattern

A *filter* is an important concept in CS. Just as a coffee filter removes coffee grounds while letting liquid pass through, a filter program removes items from a Collection if they meet some condition:

```
static void filter(Collection<?> c) {
  for (Iterator<?> it = c.iterator();
                   it.hasNext(); )
    if ( condition(it.next()) )
       it.remove();  }
```

This filter is destructive, removing items from the collection if they satisfy the `condition`. One can also write a constructive filter that makes a new collection, without modifying the original collection.

Using Library Packages

Lead us not into temptation,
but deliver us from evil.

This seems like good advice for designers of library packages, but it is not necessarily followed.

Temptation: Oh, look, there's a method that does what I want.

Evil: If you use that method, your program is $O(n^2)$.

The Moral: You need to understand the Big-O of library methods in order to use them appropriately. The Big-O of different methods varies between packages that do essentially the same thing, such as `ArrayList` and `LinkedList`, so that the packages are *not* interchangeable. There may also be restrictions on how library packages can be used.

Java List Interface

```
public interface List<AnyType>
    extends Collection<AnyType> {
  int size();
  boolean isEmpty();
  AnyType get( int index );
  AnyType set( int index, AnyType newVal );
  void    add( int index, AnyType x );
  boolean add( AnyType x );    // at end
  void    remove( int index );        }
```

`get` and `set` allow access to elements of the list by position, as in an array; `set` returns the previous value at that position. `add` adds a new element to the list *after* the specified position.

ArrayList

`ArrayList` provides a growable array implementation of a `List`.

Advantages:

- `get` and `set` are $O(1)$
- `add` and `remove` at the end are $O(1)$, so an `ArrayList` makes a good implementation of a stack.

Disadvantages:

- `add` and `remove` are $O(n)$ for random positions: the rest of the list has to be moved down to make room.
- `contains` is $O(n)$. `contains` uses the `equals()` method of the contents type.
- Some space in the array is wasted, $O(n)$, because the array grows by a factor of $3/2$ each time it is expanded.
- `clear` is $O(n)$, because `clear` replaces all existing entries with `null` to allow those items to possibly be garbage collected.

LinkedList

`LinkedList` implements a `List` as a doubly-linked list, with both forward and backward pointers.

This provides a good implementation of a linked list, stack, queue, and *deque* (often pronounced "deck") or double-ended queue.

Advantage:

- `getFirst`, `addFirst`, `removeFirst`, `getLast`, `addLast`, `removeLast` are $O(1)$.

Disadvantages:

- `get`, `set`, `add` and `remove` are $O(n)$ for random positions: it is necessary to step down the list to find the correct position. However, a `remove` during an iteration is $O(1)$.
- `contains` is $O(n)$.
- `LinkedList` seems more like an array than a true linked list. There is no method equivalent to `setrest`, so the structure of a `LinkedList` cannot be changed. It is not possible to write the destructive merge sort for linked lists that we saw earlier.

ListIterator

```
public interface ListIterator<AnyType>
       extends Iterator<AnyType> {
boolean hasPrevious();
AnyType previous();
void add( AnyType x );
void set( AnyType newVal );
```

`ListIterator` extends the functionality of `Iterator`:

- It is possible to move backwards through the list as well as forwards.
- `add` adds an element at the current position: $O(1)$ for `LinkedList` but $O(n)$ for random positions in `ArrayList`.
- `set` sets the value of the last item seen: $O(1)$ for both.

Comparing ArrayList and LinkedList

If the list contains n elements, the Big-O of operations is as follows:

Method	**ArrayList**	**LinkedList**
`get`, `set` i^{th}	$O(1)$	$O(n)$
... at front or end	$O(1)$	$O(1)$
`add`, `remove` i^{th}	$O(n)$	$O(n)$
... at front	$O(n)$	$O(1)$
... at end	$O(1)$	$O(1)$
... in `ListIterator`	$O(n)$	$O(1)$
`contains`	$O(n)$	$O(n)$

To choose the best Collection, you need to understand which methods you will use and how often you use them.

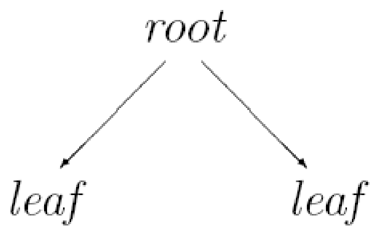
If an $O(n)$ method is used within a loop up to n, the total time required is $O(n^2)$ even though the code may look like it is $O(n)$. Thus, library methods can be a Big-O trap for the unwary.

Trees

A *tree* is a kind of *graph*, composed of *nodes* and *links*, such that:

- A link is a directed pointer from one node to another.
- There is one node, called the *root*, that has no incoming links.
- Each node, other than the root, has exactly one incoming link from its *parent*.
- Every node is reachable from the root.

A node can have any number of *children*. A node with no children is called a *leaf* ; a node with children is an *interior node*.

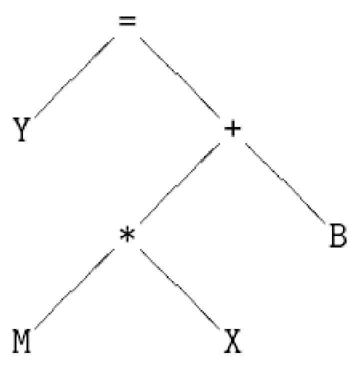


Trees occur in many places in computer systems and in nature.

Arithmetic Expressions as Trees

Arithmetic expressions can be represented as trees, with operands as leaf nodes and operators as interior nodes.

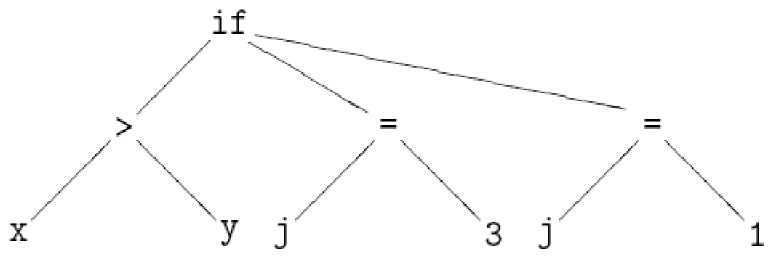
```
y = m * x + b          (= y (+ (* m x) b))
```



Computer Programs as Trees

When a compiler parses a program, it often creates a tree. When we indent the source code, we are emphasizing the tree structure.

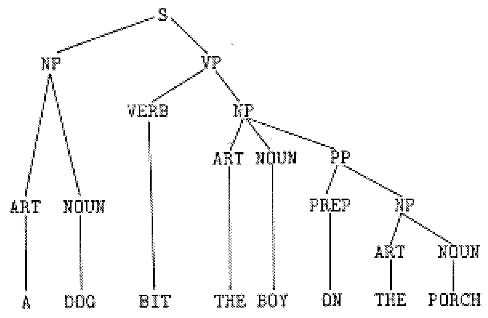
```
if ( x > y )
   j = 3;
 else
   j = 1;
```



This is called an *abstract syntax tree* or *AST*.

English Sentences as Trees

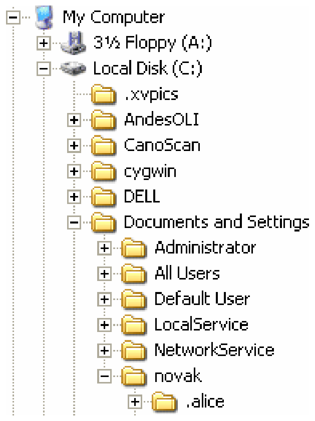
Parsing is the assignment of structure to a linear string of words according to a grammar; this is much like the diagramming of a sentence taught in grammar school.



The speaker wants to communicate a structure, but must make it linear in order to say it. The listener needs to re-create the structure intended by the speaker. Parts of the *parse tree* can then be related to object symbols in memory.

File Systems as Trees

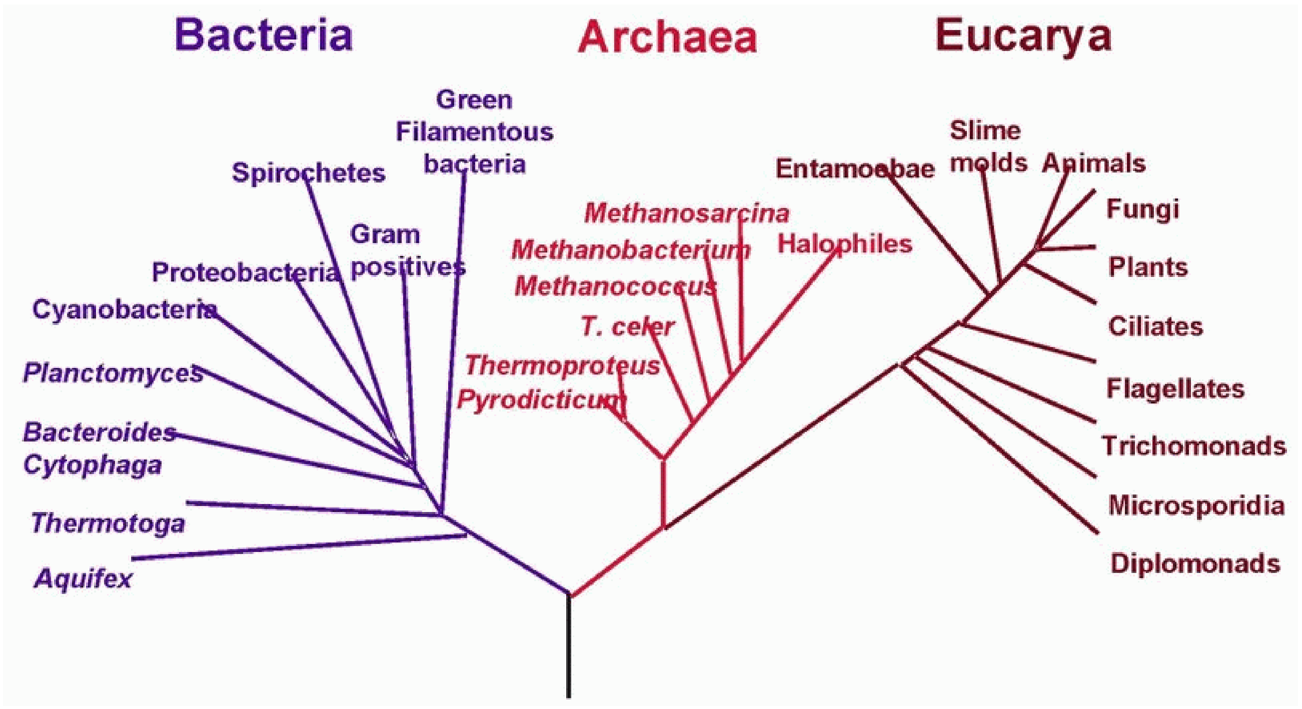
Most computer operating systems organize their file systems as trees.



A directory or folder is an interior node; a file is a leaf node.

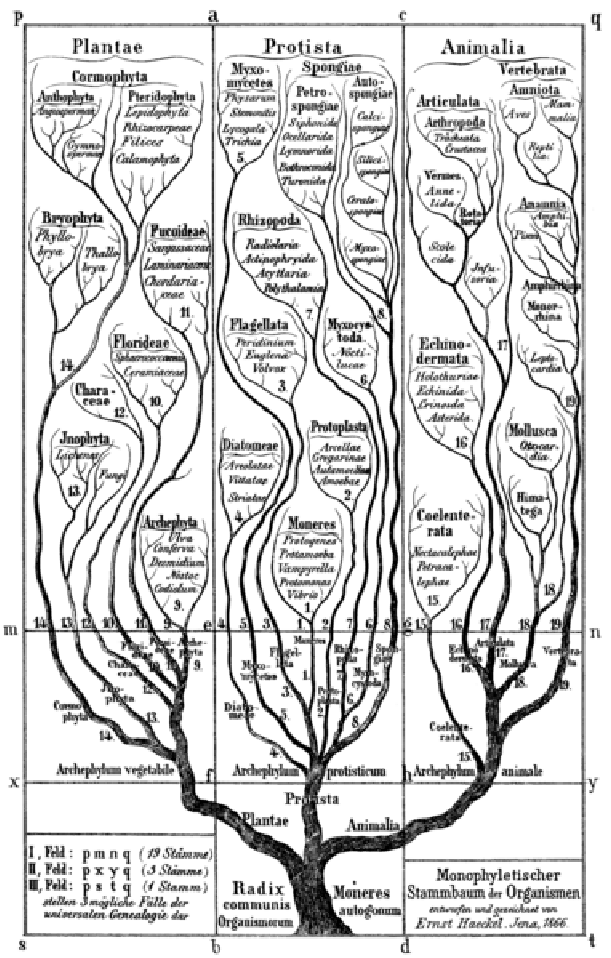
Phylogenetic Trees

As new species evolve and branch off from ancestor species, they retain most of the DNA of their ancestors. DNA of different species can be compared to reconstruct the phylogenetic tree.



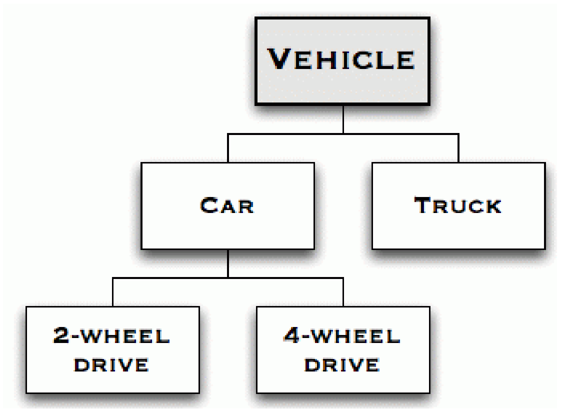
Taxonomies as Trees

Taxonomy, from the Greek words *taxis* (order) and *nomos* (law or science), was introduced to biology by Carl Linnaeus in 1760. This 1866 figure is by Ernst Haeckel.



Ontologies as Trees

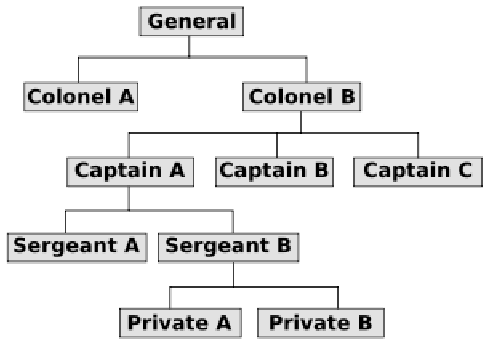
An *ontology*, from the Greek words *ontos* (of being) and *logia* (science, study, theory), is a classification of the concepts in a domain and relationships between them. An ontology describes the kinds of things that exist in our model of the world.



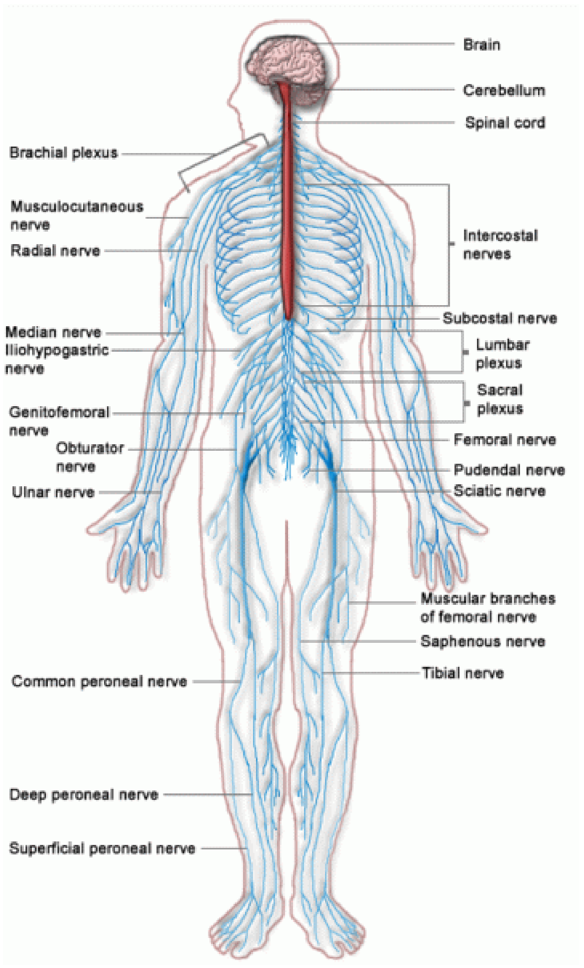
Ontologies can be represented in Java as class hierarchies, which have tree structure.

Organizations as Trees

Most human organizations are hierarchical and can be represented as trees.



Nerves


Brain
Cerebellum
Spinal cord
Brachial plexus
Musculocutaneous nerve
Radial nerve
Intercostal nerves
Subcostal nerve
Median nerve
Iliohypogastric nerve
Lumbar plexus
Sacral plexus
Genitofemoral nerve
Femoral nerve
Obturator nerve
Pudendal nerve
Ulnar nerve
Sciatic nerve
Muscular branches of femoral nerve
Saphenous nerve
Tibial nerve
Common peroneal nerve
Deep peroneal nerve
Superficial peroneal nerve

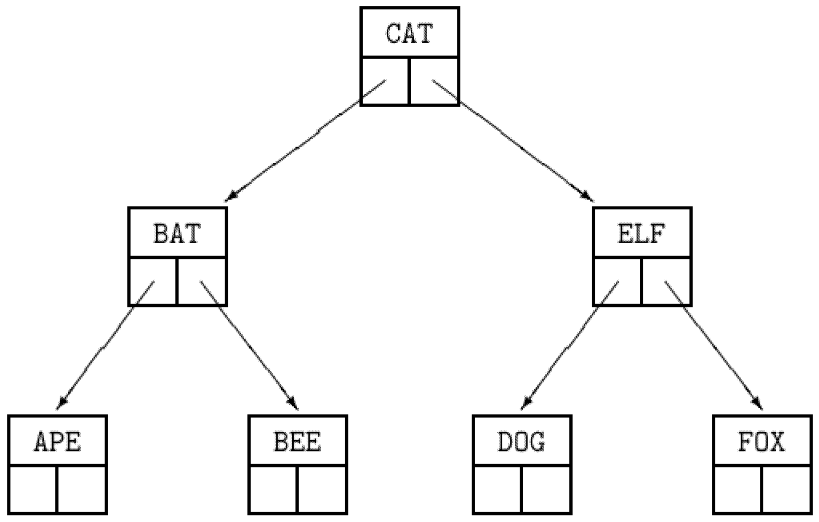
Representations of Trees

Many different representations of trees are possible:

- Binary tree: contents and left and right links.
- First-child/next-sibling: contents, first-child, next sibling.
- Linked list: the first element contains the contents, the rest are a linked list of children.
- Implicit: a node may contain only the contents; the children can be generated from the contents or from the location of the parent.

Binary Tree

```
public class Tree {
    private String contents;
    private Tree left, right;
    public Tree(String stuff, Tree lhs, Tree rhs)
      { contents = stuff;
        left = lhs;
        right = rhs; }
    public String str() { return contents; }
```



First-Child / Next-Sibling Tree

A node may not have a fixed number of children. Dedicating a large number of links would be wasteful if the average number of children is much smaller than the maximum, and it would still limit the possible number of children.

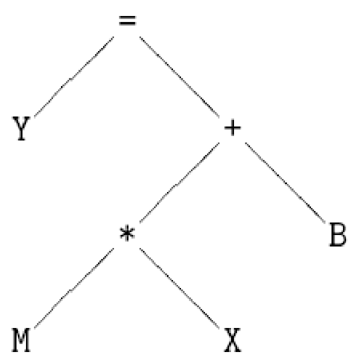
Luckily, we can use the same structure as the binary tree, with just two links, and have unlimited children with no wasted space.

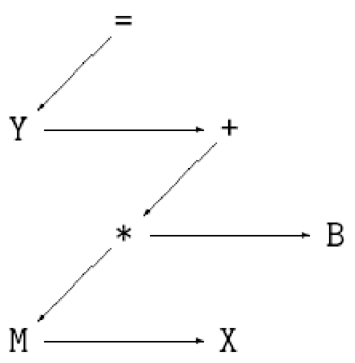
```
public class Tree {
    private Object contents;
    private Tree first;
    private Tree next;

    public Tree(Object stuff, Tree child,
                              Tree sibling)
      { contents = stuff;
        first = child;
        next = sibling; }
```

First-Child / Next-Sibling Example

```
y = m * x + b
```

Tree: **Representation:**





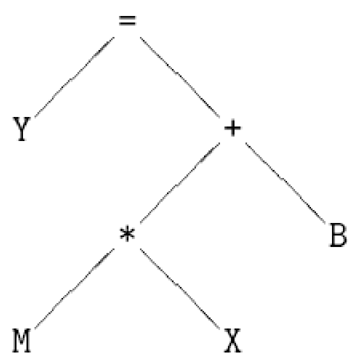
The layout of nodes in this kind of tree is the same as for a traditional tree; we are just putting the links in a different place.

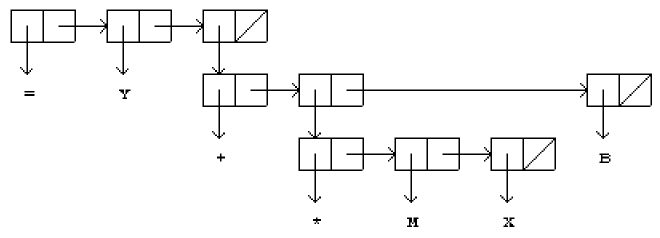
Down arrows represent the `first` child, while side arrows represent the `next` sibling.

Linked List Tree

We can think of a linked list as a tree. The first node of the list contains the contents of the node, and the rest of the list is a list of the children. The children, in turn, can be lists; a non-list is a leaf node.

```
(= Y (+ (* M X) B))
```

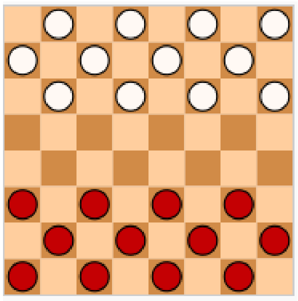




Implicit Tree

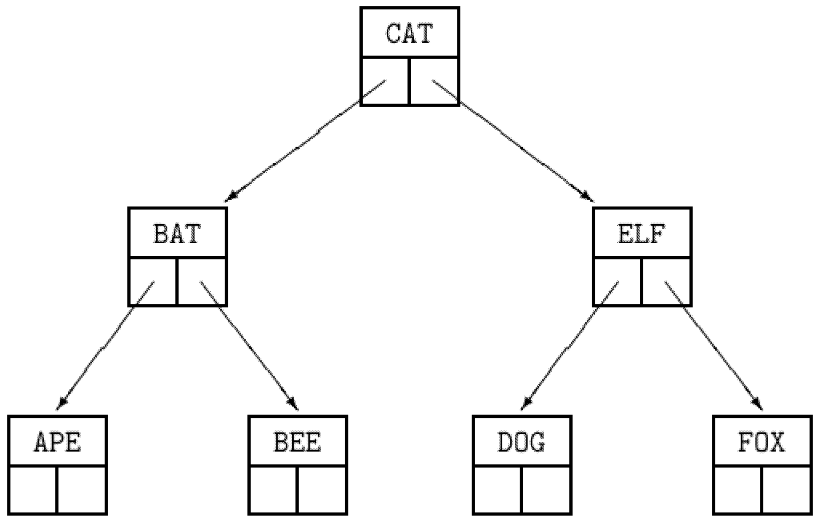
In some cases, it may be possible to compute the children from the state or the location of the parent.

For example, given the board state of a game of checkers, and knowing whose turn it is to move, it is possible to compute all possible moves and thus all child states.



Binary Search Tree (BST)

Suppose that we have a binary tree that is ordered, such that each node has `contents`, and all of its `left` descendants are less than the `contents`, and all of its `right` descendants are greater than the `contents`.



Binary Tree Search

We can efficiently search for a desired element of the tree by taking advantage of the ordering and using divide-and-conquer.

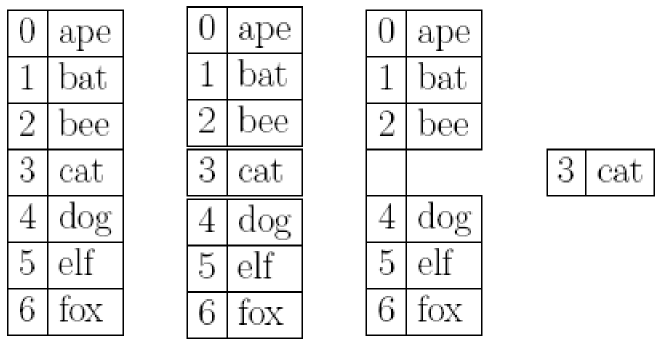
```
public static String search
                    (Tree start, String goal) {
    if ( start == null )
        return null;       // goal not found
    else
     { int test = goal.compareTo(start.contents);
       if ( test == 0 )
          return start.contents;     // found
         else if ( test < 0 )
                return search(start.left, goal);
           else return search(start.right, goal);
     } }
```

If the tree is balanced, the search takes only $O(log(n))$ time.

We return the `contents` because it is likely that the application will need to do something with it. For example, we might search on a customer name and retrieve the customer's account information.

Binary Search of Array

If objects are arranged in a sorted array, we can think of the array as a *gedanken* tree and use our binary search pattern on it.



We can think of a tree as being described by two array indexes, **`low`** and **`high`**, as follows:

- **`root = (low + high) / 2`**
- **`left = (low, root - 1)`**
- **`right = (root + 1, high)`**
- empty: **`high < low`**

Binary Tree Array Search

This version returns the index of the goal, or `-1` if not found.

```
public static int search (String [] arr, int low,
                   int high, String goal) {
  if ( high < low )
     return -1;       // goal not found
    else
     { int root = (low + high) / 2;
       int test = goal.compareTo(arr[root]);
       if ( test == 0 )
          return root;     // found
         else if ( test < 0 )
                 return search(arr, low,
                               root - 1, goal);
           else return search(arr, root + 1,
                               high, goal); } }
```

This binary search uses divide-and-conquer and takes $O(log(n))$ time.

Binary Array Search Example

```
(defun searcharr (arr low high goal)
  (let (root)
    (setq root (truncate (+ low high) 2))
    (if (< high low)
        -1               ; arr[root]
        (if (string= (aref arr root) goal)
            root
            (if (string< goal (aref arr root))
                (searcharr arr low
                           (- root 1) goal)
                (searcharr arr (+ root 1)
                           high goal) ) ) ) ))

>(setq myarr '#("ape" "bat" "bee" "cat" "dog" ...))

>(searcharr myarr 0 6 "bee")
  1> (SEARCHARR #("ape" "bat" "bee" "cat" ...) 0 6
                "bee")
    2> (SEARCHARR #("ape" "bat" "bee" ...) 0 2 "bee
      3> (SEARCHARR #("ape" "bat" "bee" ...) 2 2
                    "bee")
      <3 (SEARCHARR 2)
    <2 (SEARCHARR 2)
  <1 (SEARCHARR 2)
2
```

Binary Tree Recursion

While recursion is not always the best method for linked lists, it usually *is* the best method for trees. We don't have a problem with stack depth because depth is only $O(log(n))$ if the tree is balanced.

Suppose that we want to add up all the numbers in a Lisp tree.

```
(defun addnums (tree)
  (if (consp tree)                    ; interior node
      (+ (addnums (first tree))
         (addnums (rest tree)) )
      (if (numberp tree)              ; leaf node
          tree
          0) ) )
```

Suppose that we want the set of symbols in a Lisp tree.

```
(defun symbolset (tree)
  (if (consp tree)
      (union (symbolset (first tree))
             (symbolset (rest tree)) )
      (if (and tree (symbolp tree))
          (list tree)
          '()) ) )
```

Design Pattern: Binary Tree Recursion

This pattern is like the one for lists, except that it calls itself *twice* for interior nodes. This is essentially the same as the divide-and-conquer design pattern.

```
(defun myfun (tree)
  (if (interior? tree)
      (combine (myfun (first tree))   ; left
               (myfun (rest tree)))   ; right
      (baseanswer tree) ))            ; leaf node
```

```
(defun addnums (tree)  ; sum all numbers in tree
  (if (consp tree)
      (+ (addnums (first tree))
         (addnums (rest tree)) )
      (if (numberp tree)
          tree
          0) ) )
```

```
public static Integer addnums (Object tree) {
    if ( consp(tree) )
        return ( addnums(first((Cons)tree)) +
                 addnums(rest((Cons)tree)) );
    else if ( tree instanceof Integer )
        return (Integer) tree;
    else return Integer.valueOf(0); }
```

Design Pattern: Binary Tree Recursion

If we are using the form of binary tree that has `contents`, `left` and `right`, we can alter the design pattern slightly:

```
(defun myfun (tree)
  (if (not (null tree))
      (combine (contents tree)
               (myfun (left tree))
               (myfun (right tree)))
      (nullanswer tree) ))
```

Searching Directories for a File

Given a tree-structured directory of files and a file path, we want to find the file specified by the path. We will use a Lisp representation in which a symbol is a file or directory name and a list is a directory whose `first` element is the directory name and whose `rest` is the contents.

```
(defun findpath (dirtree path)
  (if (null dirtree)
      nil                          ; file not found
      (if (consp (first dirtree))  ; directory?
          (if (eq (first path)       ; dir name ==
                  (first (first dirtree)))
              (findpath              ; yes: go in
                (rest (first dirtree)) ; dir conten
                (rest path))           ; pop path
              (findpath              ; no: keep
                (rest dirtree)       ;     looking
                path))
          (if (eq (first path)       ; file name ==
                  (first dirtree))
              (first path)           ; yes: success
              (findpath              ; no: keep
                (rest dirtree)       ;     looking
                path)) ) ) )
```

Java Version of Findpath

```
public static String
  findpath(Cons dirtree, Cons path) {
    if ( dirtree == null )
       return null;                          // file
       else if ( consp(first(dirtree)) )     // is t
         if ( ((String) first(path))         // is f
              .equals(first((Cons)           //    d
                           first(dirtree))) )
           return findpath( rest((Cons)         // y
                                first(dirtree)),
                           rest(path)) ;
           else return findpath(rest(dirtree), // n
                               path);          //
       else if ( ((String) first(path))         // i
                   .equals(first(dirtree)) )   //
              return (String) first(path);     // y
              else return findpath(rest(dirtree), /
                                  path); }      /
```

Searching Directories Example

```
>(findpath directory '(/usr bill course cop3212 fall06 grades))

  1> (FINDPATH
         ((/USR (MARK (BOOK CH1.R CH2.R CH3.R)
                      (COURSE (COP3530 (FALL05 SYL.R) (SPR06 SYL.R)
                                       (SUM06 SYL.R)))
                      (JUNK))
                (ALEX (JUNK))
                (BILL (WORK)
                      (COURSE (COP3212 (FALL05 GRADES PROG1.R PROG2.R)
                                       (FALL06 PROG2.R PROG1.R GRADES))))
         (/USR BILL COURSE COP3212 FALL06 GRADES))
    2> (FINDPATH
           ((MARK (BOOK CH1.R CH2.R CH3.R)
                  (COURSE (COP3530 (FALL05 SYL.R) (SPR06 SYL.R)
                                   (SUM06 SYL.R)))
                  (JUNK))
            (ALEX (JUNK))
            (BILL (WORK)
                  (COURSE (COP3212 (FALL05 GRADES PROG1.R PROG2.R)
                                   (FALL06 PROG2.R PROG1.R GRADES)))))
           (BILL COURSE COP3212 FALL06 GRADES))
      3> (FINDPATH
             ((ALEX (JUNK))
              (BILL (WORK)
                    (COURSE (COP3212 (FALL05 GRADES PROG1.R PROG2.R)
                                     (FALL06 PROG2.R PROG1.R GRADES)))))
             (BILL COURSE COP3212 FALL06 GRADES))
        4> (FINDPATH
               ((BILL (WORK)
                      (COURSE (COP3212 (FALL05 GRADES PROG1.R PROG2.R)
                                       (FALL06 PROG2.R PROG1.R GRADES))))
               (BILL COURSE COP3212 FALL06 GRADES))
```

Searching Directories Example ...

```
          5> (FINDPATH
                 ((WORK)
                  (COURSE (COP3212 (FALL05 GRADES PROG1.R PROG2.R)
                                   (FALL06 PROG2.R PROG1.R GRADES))))
                 (COURSE COP3212 FALL06 GRADES))
            6> (FINDPATH
                   ((COURSE (COP3212 (FALL05 GRADES PROG1.R PROG2.R)
                                     (FALL06 PROG2.R PROG1.R GRADES))))
                   (COURSE COP3212 FALL06 GRADES))
              7> (FINDPATH
                     ((COP3212 (FALL05 GRADES PROG1.R PROG2.R)
                               (FALL06 PROG2.R PROG1.R GRADES)))
                     (COP3212 FALL06 GRADES))
                8> (FINDPATH
                       ((FALL05 GRADES PROG1.R PROG2.R)
                        (FALL06 PROG2.R PROG1.R GRADES))
                       (FALL06 GRADES))
                  9> (FINDPATH ((FALL06 PROG2.R PROG1.R GRADES))
                         (FALL06 GRADES))
                    10> (FINDPATH (PROG2.R PROG1.R GRADES) (GRADES))
                    11> (FINDPATH (PROG1.R GRADES) (GRADES))
                    12> (FINDPATH (GRADES) (GRADES))
                    <12 (FINDPATH GRADES)
                    <11 (FINDPATH GRADES)
                    <10 (FINDPATH GRADES)
                  <9 (FINDPATH GRADES)
                <8 (FINDPATH GRADES)
              <7 (FINDPATH GRADES)
            <6 (FINDPATH GRADES)
          <5 (FINDPATH GRADES)
        <4 (FINDPATH GRADES)
      <3 (FINDPATH GRADES)
    <2 (FINDPATH GRADES)
  <1 (FINDPATH GRADES)
GRADES
```

Depth First Search

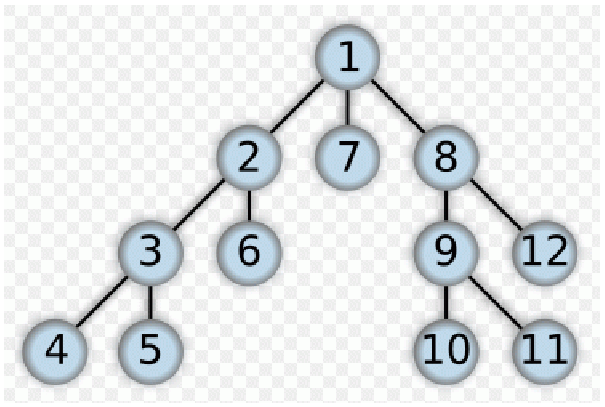
Many kinds of problems can be solved by *search*, which involves finding a goal from a starting state by applying *operators* that lead to a new state.

Suppose that a robot can take 4 actions: move west, north, east, or south. We could represent the *state* of the robot as its position. From any given state, the robot has 4 possible actions, each of which leads to a new state. This can be represented as a root node (initial state) with 4 branches to new states.

Depth-first search (*DFS*) follows an implicit tree of size $O(b^{depth})$, where b is the *branching factor*. Given a state, we test whether it is a goal or a terminal failure node; if not, we generate successor states and try searching from each of them. Most of these searches will fail, and we will *backtrack* and try a different branch.

The program execution stack records the state of examining each node. We may need our own stack of previous states to avoid getting into a loop, wandering in circles. We will return a stack of operators so we can record how we reached the goal. All of these stacks are $O(depth)$.

Depth First Search Order



Depth-first search is so named because the recursion goes deep in the tree before it goes across. The above diagram shows the order in which nodes of a search tree are examined.

Usually, the search will quit and return an answer when a node is either a terminal failure node or a goal node.

Depth-first search is often preferred because of its low $O(log(n))$ storage requirement.

Robot Mouse in Maze

Depth-first search of an implicit tree can simulate a robot mouse in a maze. The goal is to return a sequence of steps to guide the mouse to the cheese.

```
(defun mouse (maze x y prev)
  (let (path here)
    (if (or (eq (aref maze y x) '*)
            (member (setq here (list x y))
                    prev :test 'equal))
        nil                         ; fail
        (if (eq (aref maze y x) 'c)
            '(cheese)               ; success
            (if (setq path
                      (mouse maze (- x 1) y
                             (cons here prev)))
                (cons 'w path)
            (if (setq path
                      (mouse maze x (- y 1) (cons here
                (cons 'n path)
            (if (setq path
                      (mouse maze (+ x 1) y (cons here
                (cons 'e path)
            (if (setq path
                      (mouse maze x (+ y 1) (cons here
                (cons 's path)
                nil)))))) ))        ; fail
```

Robot Mouse in Java

```
public static Cons
  mouse (String[][] maze, int x, int y, Cons prev){
  Cons path;
  if ( maze[y][x].equals("*") ) return null;
  Cons here = list(new Integer(x),new Integer(y));
  if ( memberv(here, prev) != null ) return null;
  else if ( maze[y][x].equals("C") )
      return list("C");
    else if ((path = mouse(maze, x - 1, y,
                           cons(here,prev)))
             != null )
        return cons("W", path);
      else if ((path = mouse(maze, x, y - 1,
                             cons(here,prev)))
               != null )
          return cons("N", path);
        else if ((path = mouse(maze, x + 1, y,
                               cons(here,prev)))
                 != null )
            return cons("E", path);
          else if ((path = mouse(maze, x, y + 1,
                                 cons(here,prev)))
                   != null )
              return cons("S", path);
            else return null; }
```

Robot Mouse Program

- The maze is a 2-D array. `*` represents a wall. `0` represents an open space. `C` represents cheese.
- The mouse starts in an open position.
- The mouse has 4 possible moves at each point: `w`, `n`, `e` or `s`.
- We have to keep track of where the mouse has been, or it might wander infinitely in circles. The list `prev` is a stack of previous states. If the mouse re-visits a position on `prev`, we want to fail.
- We need to return an answer:
  - `nil` for failure
  - a list of moves that will lead from the current position to the goal; for the goal itself, we will use the sentinel `(cheese)`. As we unwind the recursion, we will push the operator that led to the goal onto the answer list at each step.

Robot Mouse Example

```
(setq maze (make-array '(10 10)
 :initial-contents

;  0 1 2 3 4 5 6 7 8 9
'((* * * * * * * * * *)    ; 0
  (* 0 0 * * * * * * *)    ; 1
  (* 0 * * * * * * * *)    ; 2
  (* 0 * * * * * * * *)    ; 3
  (* 0 0 0 0 0 0 * * *)    ; 4
  (* * * * 0 * 0 * * *)    ; 5
  (* * * * 0 * 0 * C *)    ; 6
  (* * * * 0 * 0 * 0 *)    ; 7
  (* * * * 0 * 0 0 0 *)    ; 8
  (* * * * 0 * * * * *)))) ; 9

>(mouse maze 4 9 '())

(N N N N N E E S S S S E E N N CHEESE)
```

Tracing the Robot Mouse

```
>(mouse maze 4 9 '())
  1> (MOUSE #2A((* * * * * * * * * *)
              (* 0 0 * * * * * * *)
              (* 0 * * * * * * * *)
              (* 0 * * * * * * * *)
              (* 0 0 0 0 0 0 * * *)
              (* * * * 0 * 0 * * *)
              (* * * * 0 * 0 * C *)
              (* * * * 0 * 0 * 0 *)
              (* * * * 0 * 0 0 0 *)
              (* * * * 0 * * * * *)) 4 9 NIL)
    2> (MOUSE 3 9 ((4 9)))          ; west
    <2 (MOUSE NIL)                  ; hit the wall
    2> (MOUSE 4 8 ((4 9)))          ; north
      3> (MOUSE 3 8 ((4 8) (4 9)))    ; west
      <3 (MOUSE NIL)                  ; hit the wall
      3> (MOUSE 4 7 ((4 8) (4 9)))    ; north
        4> (MOUSE 4 6 ((4 7) (4 8) (4 9)))    ; north
          5> (MOUSE 4 5 ((4 6) (4 7) (4 8) (4 9)))    ; north
            6> (MOUSE 4 4 ((4 5) (4 6) (4 7) (4 8) (4 9))) ; north
              7> (MOUSE 3 4 ((4 4) (4 5) (4 6) (4 7) (4 8) ; west
                8> (MOUSE 2 4 ((3 4) (4 4) (4 5) (4 6)     ; west
                  9> (MOUSE 1 4 ((2 4) (3 4) (4 4) (4 5)   ; west
                    10> (MOUSE 0 4 ((1 4) (2 4) (3 4) (4 4) ; west
                    <10 (MOUSE NIL)         ; hit the wall
                    10> (MOUSE 1 3 ((1 4) (2 4) (3 4) (4 4) ; north
                    11> (MOUSE 1 2 ((1 3) (1 4) (2 4) (3 4)
                    12> (MOUSE 1 1 ((1 2) (1 3) (1 4) (2 4)
                    13> (MOUSE 1 0 ((1 1) (1 2) (1 3) (1 4) ; north
                    <13 (MOUSE NIL)         ; hit the wall
                    13> (MOUSE 2 1 ((1 1) (1 2) (1 3) (1 4) ; east
                    14> (MOUSE 1 1 ((2 1) (1 1) (1 2) (1 3) ; west
                    <14 (MOUSE NIL)   ; ! loop
```

Tracing the Robot Mouse ...

```
                14> (MOUSE 3 1 ((2 1) (1 1) (1 2) (1 3)
                <14 (MOUSE NIL)
                14> (MOUSE 2 2 ((2 1) (1 1) (1 2) (1 3)
                <14 (MOUSE NIL)
                <13 (MOUSE NIL) ...
          <7 (MOUSE NIL)                  ; fail back to (4 4)
          7> (MOUSE 5 4 ((4 4) (4 5) (4 6) (4 7)        ; east
            8> (MOUSE 6 4 ((5 4) (4 4) (4 5) (4 6)
              9> (MOUSE 6 5 ((6 4) (5 4) (4 4) (4 5)    ; south
                10> (MOUSE 6 6 ((6 5) (6 4) (5 4)
                11> (MOUSE 6 7 ((6 6) (6 5) (6 4)
                12> (MOUSE 6 8 ((6 7) (6 6) (6 5)       ; south
                13> (MOUSE 7 8 ((6 8) (6 7) (6 6)       ; east
                14> (MOUSE 8 8 ((7 8) (6 8) (6 7)       ; east
                15> (MOUSE 8 7 ((8 8) (7 8) (6 8)       ; north
                16> (MOUSE 8 6 ((8 7) (8 8) (7 8)       ; north
                <16 (MOUSE (CHEESE))          ; found the cheese!
                <15 (MOUSE (N CHEESE))        ; last move was N
                <14 (MOUSE (N N CHEESE))      ; push on operators
                <13 (MOUSE (E N N CHEESE))      ; as we backtrack
                <12 (MOUSE (E E N N CHEESE))
                <11 (MOUSE (S E E N N CHEESE))
                <10 (MOUSE (S S E E N N CHEESE))
              <9 (MOUSE (S S S E E N N CHEESE))
            <8 (MOUSE (S S S S E E N N CHEESE))
          <7 (MOUSE (E S S S S E E N N CHEESE))
        <6 (MOUSE (E E S S S S E E N N CHEESE))
      <5 (MOUSE (N E E S S S S E E N N CHEESE))
    <4 (MOUSE (N N E E S S S S E E N N CHEESE))
  <3 (MOUSE (N N N E E S S S S E E N N CHEESE))
 <2 (MOUSE (N N N N E E S S S S E E N N CHEESE))
<1 (MOUSE (N N N N N E E S S S S E E N N CHEESE))
(N N N N N E E S S S S E E N N CHEESE)
```

Tree Traversal

For some applications, we need to traverse an entire tree, performing some action as we go. There are three basic orders of processing:

- **Preorder:** process the parent node before its children.
- **Inorder:** process one child, then the parent, then the other child.
- **Postorder:** process children first, then the parent.

Thus, the name of the order tells when the parent is processed.

We will examine each of these with an example.

Preorder

In preorder, the parent node is processed before its children.

Suppose that we want to print out a directory name, then the contents of the directory. We will also indent to show the depth. We assume a directory tree as shown earlier: a directory is a list of the directory name followed by its contents; a non-list is a file.

```
(defun printdir (dir level)
  (spaces (* 2 level))
  (if (symbolp dir)
      (progn (prin1 dir)
             (terpri))
      (progn (prin1 (first dir))
             (terpri)
             (dolist (contents (rest dir))
               (printdir contents
                         (+ level 1)) ) ) ) )
```

Preorder Example

```
>(printdir (first directory) 0)
/USR
  MARK
    BOOK
      CH1.R
      CH2.R
      CH3.R
    COURSE
      COP3530
        FALL05
          SYL.R
        SPR06
          SYL.R
        SUM06
          SYL.R
    JUNK
  ALEX
    JUNK
  BILL
    WORK
    COURSE
      COP3212
        FALL05
          GRADES
          PROG1.R
          PROG2.R
        FALL06
          PROG2.R
          PROG1.R
          GRADES
```

Preorder Example in Java

```
public static void printdir (Object dir, int level) {
    for ( int i = 0; i < level; i++ )
        System.out.print("  ");
    if ( ! consp(dir) )
        System.out.println(dir);
    else {  System.out.println(first((Cons) dir));
            while ( rest((Cons) dir) != null ) {
                dir = rest((Cons) dir);
                printdir(first((Cons) dir),
                         level + 1); } } }
```

```
(/usr (mark (book ch1.r ch2.r ch3.r)
            (course (cop3530 (fall05 syl.r)
                      (spr06 syl.r) (sum06 syl.r)))
            (junk))
      (alex (junk))
      (bill (work) ... ))
```

```
/usr
  mark
    book
      ch1.r
      ch2.r
      ch3.r
    course   ...
```

Inorder

There are several ways of writing arithmetic expressions; these are closely related to the orders of tree traversal:

- **Prefix** or *Cambridge Polish*, as in Lisp: `(+ x y)`
- **Infix**, as in Java: `x + y`
- **Polish Postfix**: `x y +`

An expression tree can be printed as infix by an inorder traversal:

```
(defun op  (x) (first x))    ; access functions
(defun lhs (x) (second x))   ; left-hand side
(defun rhs (x) (third x))    ; right-hand side

(defun infix (x)
  (if (consp x)
      (progn (princ "(")
             (infix (lhs x))    ; first child
             (prin1 (op x))     ; parent
             (infix (rhs x))    ; second child
             (princ ")") )
      (prin1 x)))

>(infix '(* (+ x y) z))
((X+Y)*Z)
```

Inorder Printing of Binary Tree

An ordered binary tree can be printed in sorted order by an inorder traversal. It is clear that inorder is the right algorithm since the parent is between the two children in the sort ordering.

```
(defun printbt (tree)
  (if (consp tree)
      (progn (printbt (lhs tree))    ; 1. L child
             (print (op tree))       ; 2. parent
             (printbt (rhs tree)))   ; 3. R child
      (if tree (print tree)) ) )
```

```
>(printbt '(cat (bat ape
                     bee)
                (elf dog
                     fox)))
APE
BAT
BEE
CAT
DOG
ELF
FOX
```

Flattening Binary Tree

An ordered binary tree can be flattened into an ordered list by a backwards inorder traversal. We do the inorder backwards so that pushing onto a stack (using `cons`) can be used to accumulate the result.

```
(defun flattenbt (tree) (flattenbtb tree '()) )
(defun flattenbtb (tree result)
  (if (consp tree)
      (flattenbtb (lhs tree)          ; 3. L child
                  (cons (op tree)     ; 2. parent
                        (flattenbtb
                          (rhs tree) ; 1. R child
                          result)))
      (if tree
          (cons tree result)
          result) ) )

>(flattenbt '(cat (bat ape
                       bee)
                  (elf dog
                       fox)))

(APE BAT BEE CAT DOG ELF FOX)
```

Tracing Flattening Binary Tree

```
>(flattenbt '(cat (bat ape
                       bee)
                  (elf dog
                       fox)))

  1> (FLATTENBT (CAT (BAT APE BEE) (ELF DOG FOX)))
    2> (FLATTENBTB (CAT (BAT APE BEE) (ELF DOG FOX)) NIL)
      3> (FLATTENBTB (ELF DOG FOX) NIL)
        4> (FLATTENBTB FOX NIL)
        <4 (FLATTENBTB (FOX))
        4> (FLATTENBTB DOG (ELF FOX))
        <4 (FLATTENBTB (DOG ELF FOX))
      <3 (FLATTENBTB (DOG ELF FOX))
      3> (FLATTENBTB (BAT APE BEE) (CAT DOG ELF FOX))
        4> (FLATTENBTB BEE (CAT DOG ELF FOX))
        <4 (FLATTENBTB (BEE CAT DOG ELF FOX))
        4> (FLATTENBTB APE (BAT BEE CAT DOG ELF FOX))
        <4 (FLATTENBTB (APE BAT BEE CAT DOG ELF FOX))
      <3 (FLATTENBTB (APE BAT BEE CAT DOG ELF FOX))
    <2 (FLATTENBTB (APE BAT BEE CAT DOG ELF FOX))
  <1 (FLATTENBT (APE BAT BEE CAT DOG ELF FOX))
(APE BAT BEE CAT DOG ELF FOX)
```

Flattening Binary Tree in Java

```
public static Cons flattenbt (Object tree) {
    return flattenbtb(tree, null); }

public static Cons flattenbtb (Object tree,
                               Cons result) {
 if ( consp(tree) )
    return flattenbtb( lhs((Cons) tree),
                       cons( op((Cons) tree),
                             flattenbtb(
                               rhs((Cons) tree),
                               result)));
    return ( tree == null ) ? result
        : cons(tree, result); }

btr = (cat (bat ape bee)
           (elf dog fox))

flatten = (ape bat bee cat dog elf fox)
```

Postorder

The Lisp function `eval` evaluates a symbolic expression. We can write a version of `eval` using postorder traversal of an expression tree with numeric leaf values. Postorder follows the usual rule for evaluating function calls, *i.e.*, arguments are evaluated before the function is called.

```
(defun myeval (x)
  (if (numberp x)
      x
      (funcall (op x)               ; execute the op
               (myeval (lhs x))
               (myeval (rhs x)) ) ) )
```

```
>(myeval '(* (+ 3 4) 5))
  1> (MYEVAL (* (+ 3 4) 5))
    2> (MYEVAL (+ 3 4))
      3> (MYEVAL 3)
      <3 (MYEVAL 3)
      3> (MYEVAL 4)
      <3 (MYEVAL 4)
    <2 (MYEVAL 7)
    2> (MYEVAL 5)
    <2 (MYEVAL 5)
  <1 (MYEVAL 35)
35
```

Balanced Binary Trees

We have seen that searching a binary tree is very efficient, $O(log(n))$; however, this is only true if the tree is *balanced*, i.e. the two branches of a node have approximately the same height. If the tree is unbalanced, search time could be worse, perhaps even $O(n)$.

There are several clever algorithms that maintain *self-balancing* binary trees; these algorithms re-balance the tree as needed.

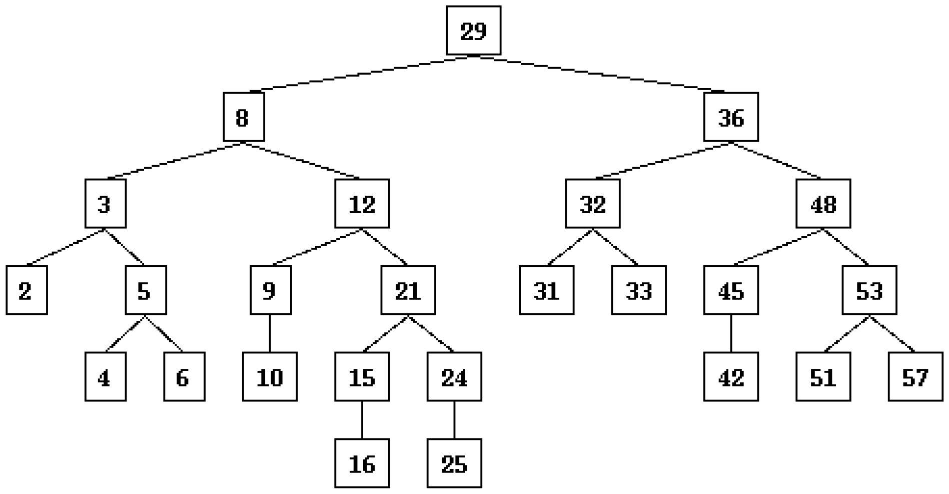
- **AVL Tree**: heights of subtrees differ by at most 1. A node contains a *balance* value of -1, 0, or 1, which is the difference in height of its subtrees. If the balance goes out of this range, the tree is rebalanced.
- **Red-Black Tree**: nodes are *colored* red or black. The longest path from root to a leaf is no more than twice the length of the shortest path.
- **Splay Tree**: the tree is rebalanced so that recently accessed elements can be accessed quickly the next time.

AVL Tree

An AVL Tree [3] is a binary tree that is approximately height-balanced: left and right subtrees of any node differ in height by at most 1.

Advantage: approximately $O(log(n))$ search and insert time.

Disadvantage: complex code (120 - 200 lines).

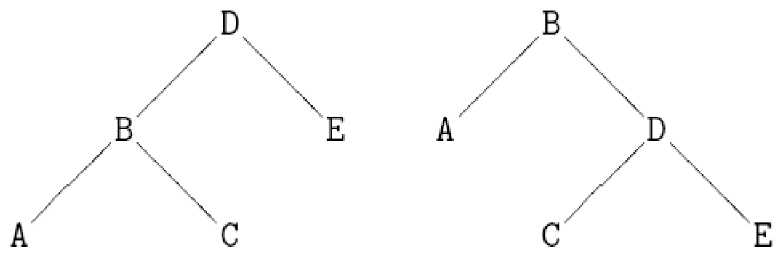


`http://www.cs.utexas.edu/users/novak/cgi/apserver.cgi` will generate AVL programs in your choice of language.

[3] G. M. Adel'son-Vel'skiĭ and E. M. Landis, *Soviet Math.* **3**, 1259-1263, 1962; D. Knuth, *The Art of Computer Programming, vol. 3: Sorting and Searching*, Addison-Wesley, 1973, section 6.2.3.

Tree Rotation

The basic idea upon which self-balancing trees are based is *tree rotation*. Rotations change the height of subtrees but do not affect the ordering of elements required for binary search trees.



The tree on the left is converted to the tree on the right with a *right rotation*, or vice versa with a *left rotation*. Both trees maintain the required ordering, but the heights of the subtrees have changed. In this diagram, `B` and `D` are single nodes, while `A`, `C`, and `E` could be subtrees.

There are several other rotation cases, but this is the basic idea.

B-Tree

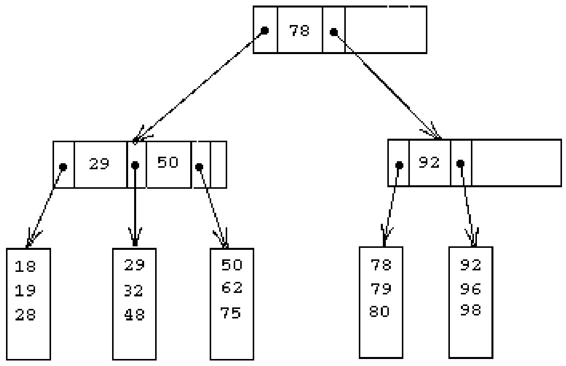
Suppose that a tree is too big to be kept in memory, and thus must be kept on a disk. A disk has large capacity (e.g. a terabyte, 10^{12} bytes) but slow access (e.g. 10 milliseconds). A computer can execute millions of instructions in the time required for one disk access.

We would like to minimize the number of disk accesses required to get to the data we want. One way to do this is to use a tree with a very high branching factor.

- Every interior node (except the root) has between $m/2$ and m children. m may be large, e.g 256, so that an interior node fills a disk block.
- Each path from the root to a leaf has the same length.
- The interior nodes, containing keys and links, may be a different type than the leaf nodes.
- A link is a disk address.
- The real data (e.g. customer record) is stored at the leaves; this is sometimes called a *B+ tree*. Leaves will have between $l/2$ and l data items.

B-Tree Implementation

Conceptually, an interior node is an array of n pointers and $n-1$ key values. A pointer is between the two key values that bound the entries that it covers; we imagine that the array is bounded by key values of $-\infty$ and ∞. (In practice, two arrays, pointers and key values, may be used since they are of different types.)



Binary search of the array of key values can be used to find the right pointer to follow. If we have the sequence

$key_i \;\; pointer_i \;\; key_{i+1}$

$pointer_i$ covers all keys such that $key_i \le key < key_{i+1}$.

Advantages of B-Trees

- The desired record is found at a shallow depth (few disk accesses). A tree with 256 keys per node can index millions of records in 3 steps or 1 disk access (keeping the root node and next level in memory).
- In general, there are many more searches than insertions.
- Since a node can have a wide range of children, $m/2$ to m, an insert or delete will rarely go outside this range. It is rarely necessary to rebalance the tree.
- Inserting or deleting an item within a node is $O(blocksize)$, since on average half the block must be moved, but this is fast compared to a disk access.
- Rebalancing the tree on insertion is easy: if a node would become over-full, break it into two half-nodes, and insert the new key and pointer into its parent. Or, if a leaf node becomes over-full, see if a neighbor node can take some extra children.
- In many cases, rebalancing the tree on deletion can simply be ignored: it only wastes disk space, which is cheap.

Quadtree

A *quadtree* is a tree in which each interior node has 4 descendants. Quadtrees are often used to represent 2-dimensional spatial data such as images or geographic regions.

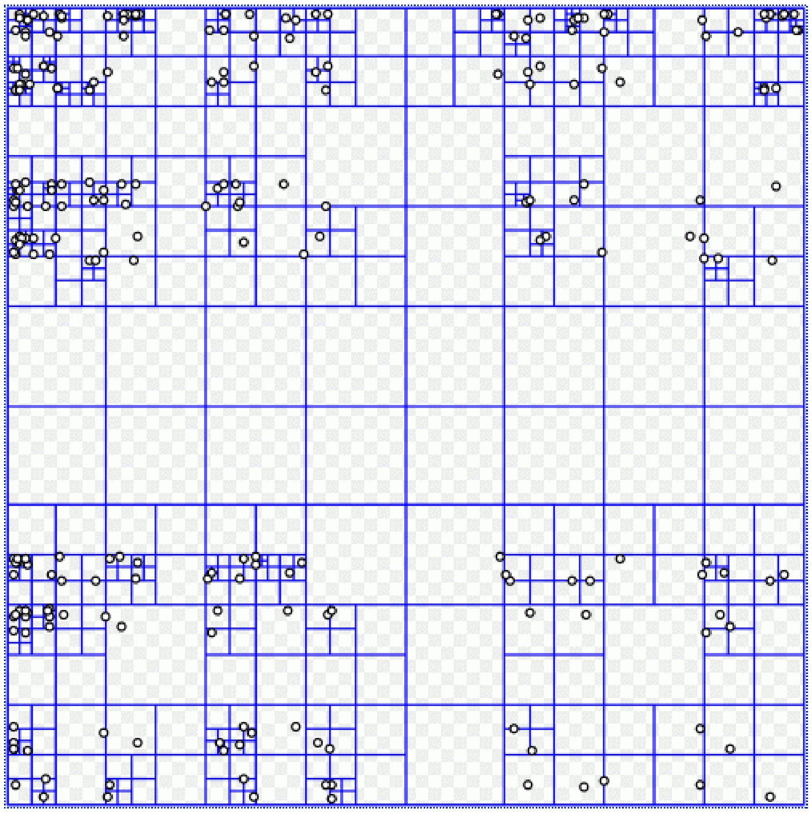
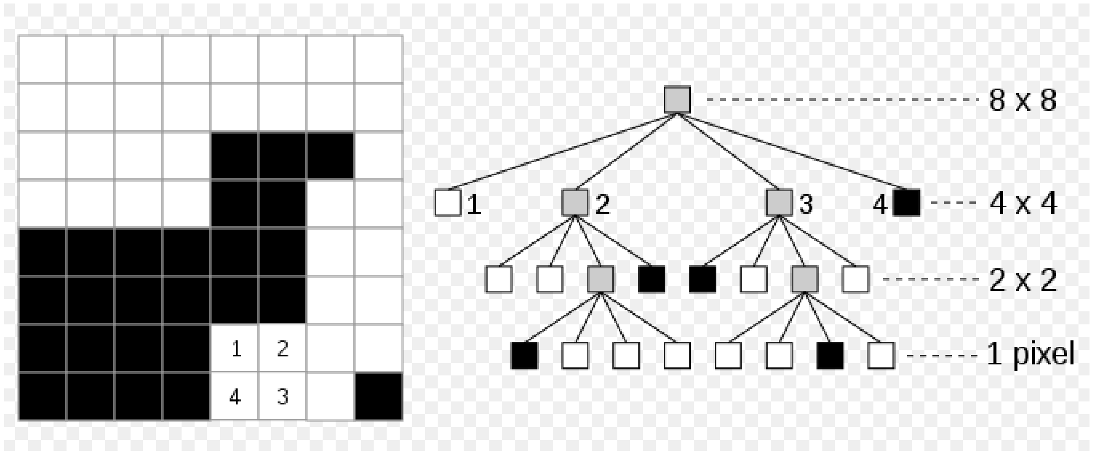
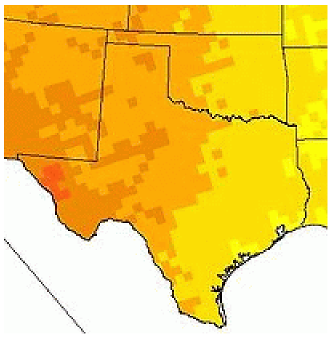


Image Quadtree

An image quadtree can represent an image more compactly than a pixel representation if the image contains homogeneous regions (which most real images do). Even though the image is compressed, the value at any point can be looked up quickly, in $O(log\ n)$ time.





Intersection of Quadtrees

Quadtrees A and B can be efficiently intersected:

- If $A = 0$ or $B = 0$, the result is 0.
- If $A = 1$, the result is B.
- If $B = 1$, the result is A.
- Otherwise, make a new interior node whose values are corresponding intersections of children of A and B.

Notice that the intersection can often reuse parts of the input trees. Uses include:

- Geospatial modeling, e.g. what is the intersection of area of forecast rain with area of corn?
- Graphics, games: view frustum culling
- Collision detection

Aggregate Data in Quadtrees

A quadtree node can hold aggregate data about the area covered by the node:

- Addition, averaging, statistics: each node can aggregate the data of its children.
- Color: a node can average the color of its children.
- Updating: aggregate data can be kept up-to-date in $O(log\ n)$ time.

Uses of Quadtrees

- Spatial indexing: $O(log\ n)$ lookup of data values by spatial position, while using less storage than an array.
- Numerical algorithms: spatial algorithms can be much more efficient by using lower-resolution data for interactions that are far apart.
- Graphics: reduce resolution except where user is looking.

An *octtree* is a 3-dimensional tree similar to a quadree.

Be Extreme!

Sometimes an extreme solution to a problem may be best:

- Buy enough memory to put everything in main memory. A 32-bit PC can address 4 GB of memory, or 200 bytes for every person in Texas.
- Buy a lot of PC's and put part of the data on each PC.
- Use a SSN (9 digits = 1 Gig) as an array index to index a big array stored on disk: no disk accesses to find the disk address. Not all 9-digit numbers are valid SSN's, so some disk will be wasted; but who cares?
- Buy a million PC's if that is what it takes to do your computation.

Sparse Arrays

In a *sparse* array, most elements are zero or empty.

A two-dimensional array of dimension n takes $O(n^2)$ storage; that gets expensive fast as n increases. However, a sparse array might have only $O(n)$ nonzero elements.

What we would like to do is to store only the nonzero elements in a lookup structure; if we look up an element and don't find it, we can return zero.

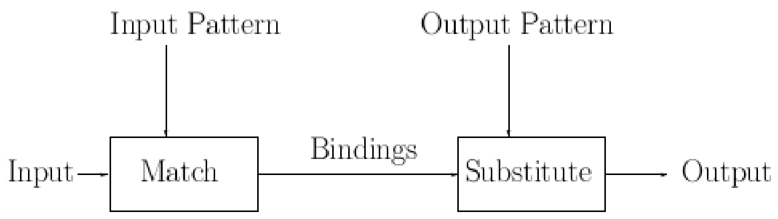
For example, we could have an array of size n for the first index; this could contain pointers to a linked list or tree of values using the second index as the search key.

```
(defun sparse-aref (arr i j)
  (or (second (assoc j (aref arr i)))
      0))

(setq myarr (make-array '(10) :initial-contents
'(nil nil ((3 77) (2 13)) nil ((5 23))
  nil nil ((2 47) (6 52)) nil nil)))

>(sparse-aref myarr 7 6)
52

>(sparse-aref myarr 7 4)
0
```

Pattern Matching Overview

We have emphasized the use of *design patterns* in writing programs. We would like to use patterns automatically to generate, improve, or transform programs, equations, and other tree-like data structures.

We will use *rewrite rules*, each consisting of an input pattern and an output pattern.



- **Input Pattern:** `(- (- ?x ?y))`
- **Output Pattern:** `(- ?y ?x)`
- **Rewrite Rule:** `( (- (- ?x ?y))  (- ?y ?x))`
- **Example Input:** `(- (- (sin theta) z))`
- **Bindings:** `((?y z) (?x (sin theta)))`
- **Output:** `(- z (sin theta))`

Copy Tree and Substitute

It is easy to write a function to copy a binary tree:

```
(defun copy-tree (z)
  (if (consp z)
      (cons (copy-tree (first z))
            (copy-tree (rest z)))
      z) )
```

Why make an exact copy of a tree that we already have? Well, if we modify `copy-tree` slightly, we can make a copy with a substitution:

```
; substitute x for y in z
(defun subst (x y z)
  (if (consp z)
      (cons (subst x y (first z))
            (subst x y (rest z)))
      (if (eql z y) x z)) )
```

```
>(subst 'axolotl 'banana '(banana pudding))
(AXOLOTL PUDDING)


>(subst 10 'r '(* pi (* r r)))
(* PI (* 10 10))


>(eval (subst 10 'r '(* pi (* r r))))
314.1592653589793
```

Copy Tree and Substitute in Java

```
public static Object copy_tree(Object tree) {
  if ( consp(tree) )
    return cons(copy_tree(first((Cons) tree)),
         (Cons) copy_tree(rest( (Cons) tree)));
  return tree; }

public static Object
  subst(Object gnew, String old, Object tree) {
    if ( consp(tree) )
      return cons(subst(gnew, old,
                        first((Cons) tree)),
           (Cons) subst(gnew, old,
                        rest((Cons) tree)));
  return (old.equals(tree)) ? gnew : tree; }
```

Binding Lists

A *binding* is a correspondence of a name and a value. In Lisp, it is conventional to represent a binding as a `cons`: `(cons` *name value*`)`, e.g. `(?X . 3)`; we will use a list, `(list` *name value*`)`, e.g. `(?X 3)` . We will use names that begin with `?` to denote variables.

A set of bindings is represented as a list, called an *association list*, or *alist* for short. A new binding can be added by:
`(cons (list` *name value*`)` *binding-list* `)`

A name can be looked up using `assoc`:
`(assoc` *name binding-list* `)`

```
(assoc '?y '((?x 3) (?y 4) (?z 5)))
   =  (?Y 4)
```

The value of the binding can be gotten using `second`:

```
(second (assoc '?y '((?x 3) (?y 4) (?z 5))))
   =  4
```

Multiple Substitutions

The function `(sublis alist form)` makes multiple substitutions simultaneously:

```
>(sublis '((rose peach) (smell taste))
         '(a rose by any other name
           would smell as sweet))

(A PEACH BY ANY OTHER NAME WOULD TASTE AS SWEET)

; substitute in z with bindings in alist
(defun sublis (alist z)
  (let (pair)
    (if (consp z)
        (cons (sublis alist (first z))
              (sublis alist (rest z)))
        (if (setq pair (assoc z alist))
            (second pair)
            z)) ))
```

Sublis in Java

```
public static Object
  sublis(Cons alist, Object tree) {
    if ( consp(tree) )
      return cons(sublis(alist,
                         first((Cons) tree)),
           (Cons) sublis(alist,
                         rest((Cons) tree)));
    if ( tree == null ) return null;
    Cons pair = assoc(tree, alist);
    return ( pair == null ) ? tree
                            : second(pair); }
```

Instantiating Design Patterns

`sublis` can be used to instantiate design patterns. For example, we can instantiate a tree-recursive accumulator pattern to make various functions:

```
(setq pattern
  '(defun ?fun (tree)
     (if (consp tree)
         (?combine (?fun (first tree))
                   (?fun (rest tree)))
         (if (?test tree) ?trueval ?falseval))))

>(sublis '((?fun      nnums  )
           (?combine  +      )
           (?test     numberp)
           (?trueval  1      )
           (?falseval 0      ))      pattern)

(DEFUN NNUMS (TREE)
  (IF (CONSP TREE)
      (+ (NNUMS (FIRST TREE))
         (NNUMS (REST TREE)))
      (IF (NUMBERP TREE) 1 0)))

>(nnums '(+ 3 (* i 5)))
2
```

Tree Equality

It often is necessary to test whether two trees are *equal*, even though they are in different memory locations. We will say two trees are equal if:

- the structures of the trees are the same
- the leaf nodes are equal, using an appropriate test

```
(defun equal (pat inp)
  (if (consp pat)           ; interior node?
      (and (consp inp)
           (equal (first pat) (first inp))
           (equal (rest pat) (rest inp)))
      (eql pat inp) ) )     ; leaf node

>(equal '(+ a (* b c)) '(+ a (* b c)))

T
```

Tree Equality in Java

```
public static boolean
  equal(Object tree, Object other) {
    if ( consp(tree) )
        return ( consp(other) &&
                 equal(first((Cons) tree),
                       first((Cons) other)) &&
                 equal(rest((Cons) tree),
                       rest((Cons) other)) );
    return eql(tree, other); }

public static boolean       // leaf equality
  eql(Object tree, Object other) {
    return ( (tree == other) ||
             ( (tree != null) &&
               (other != null) &&
               tree.equals(other) ) ); }
```

Tracing Equal

```
>(equal '(+ a (* b c)) '(+ a (* b c)))

  1> (EQUAL (+ A (* B C)) (+ A (* B C)))
    2> (EQUAL + +)
    <2 (EQUAL T)
    2> (EQUAL (A (* B C)) (A (* B C)))
      3> (EQUAL A A)
      <3 (EQUAL T)
      3> (EQUAL ((* B C)) ((* B C)))
        4> (EQUAL (* B C) (* B C))
          5> (EQUAL * *)
          <5 (EQUAL T)
          5> (EQUAL (B C) (B C))
            6> (EQUAL B B)
            <6 (EQUAL T)
            6> (EQUAL (C) (C))
              7> (EQUAL C C)
              <7 (EQUAL T)
              7> (EQUAL NIL NIL)
              <7 (EQUAL T)
            <6 (EQUAL T)
          <5 (EQUAL T)
        <4 (EQUAL T)
        4> (EQUAL NIL NIL)
        <4 (EQUAL T)
      <3 (EQUAL T)
    <2 (EQUAL T)
  <1 (EQUAL T)
T
```

This is our old friend, depth-first search.

Design Pattern: Nested Tree Recursion

Binary tree recursion can be written in a nested form, analogous to tail recursion. We carry the answer along, adding to it as we go. This form is useful if it is easier to combine an item with an answer than to combine two answers. Compare to p. 125

```
(defun myfun (tree) (myfunb tree init) )

(defun myfunb (tree answer)
  (if (interior? tree)
      (myfunb (right tree)
              (myfunb (left tree) answer))
      (combine (baseanswer tree) answer)))
```

```
; count numbers in a tree
(defun nnums (tree) (nnumsb tree 0))
(defun nnumsb (tree answer)
  (if (consp tree)
      (nnumsb (rest tree)
              (nnumsb (first tree) answer))
      (if (numberp tree)
          (+ 1 answer)
          answer) ) )
```

Tracing Nested Tree Recursion

```
>(nnums '(+ (* x 3) (/ z 7)))
  1> (NNUMSB (+ (* X 3) (/ Z 7)) 0)
    2> (NNUMSB + 0)
    <2 (NNUMSB 0)
    2> (NNUMSB ((* X 3) (/ Z 7)) 0)
      3> (NNUMSB (* X 3) 0)
        4> (NNUMSB * 0)
        <4 (NNUMSB 0)
        4> (NNUMSB (X 3) 0)
          5> (NNUMSB X 0)
          <5 (NNUMSB 0)
          5> (NNUMSB (3) 0)
            6> (NNUMSB 3 0)
            <6 (NNUMSB 1)
            6> (NNUMSB NIL 1)
            <6 (NNUMSB 1)
          <5 (NNUMSB 1)
        <4 (NNUMSB 1)
      <3 (NNUMSB 1)
      3> (NNUMSB ((/ Z 7)) 1)
        4> (NNUMSB (/ Z 7) 1)
          5> (NNUMSB / 1)
          <5 (NNUMSB 1)
          5> (NNUMSB (Z 7) 1)
            6> (NNUMSB Z 1)
            <6 (NNUMSB 1)
            6> (NNUMSB (7) 1)
              7> (NNUMSB 7 1)
              <7 (NNUMSB 2)
              7> (NNUMSB NIL 2)
              <7 (NNUMSB 2)
            <6 (NNUMSB 2)
     ...
2
```

Pattern Matching

Pattern matching is the inverse of substitution: it tests to see whether an input is an instance of a pattern, and if so, how it matches.

```
>(match '(go ?expletive yourself)
        '(go bleep yourself))

((?EXPLETIVE BLEEP) (T T))

(match '(defun ?fun (tree)
          (if (consp tree)
              (?combine (?fun (car tree))
                        (?fun (cdr tree)))
              (if (?test tree) ?trueval ?falseval)))

       '(DEFUN NNUMS (TREE)
          (IF (CONSP TREE)
              (+ (NNUMS (CAR TREE))
                 (NNUMS (CDR TREE)))
              (IF (NUMBERP TREE) 1 0))) )

((?FALSEVAL 0) (?TRUEVAL 1) (?TEST NUMBERP)
 (?COMBINE +) (?FUN NNUMS) (T T))
```

Specifications of Match

- Inputs: a pattern, `pat`, and an input, `inp`
- Constants in the pattern must match the input exactly.
- Structure that is present in the pattern must also be present in the input.
- Variables in the pattern are symbols (strings) that begin with `?`
- A variable can match anything, but it must do so consistently.
- The result of `match` is a list of bindings: `null` indicates failure, not `null` indicates success.
- The dummy binding `(T T)` is used to allow an empty binding list that is not `null`.

Match Function

```
(defun equal (pat inp)
  (if (consp pat)          ; interior node?
      (and (consp inp)
           (equal (first pat) (first inp))
           (equal (rest pat) (rest inp)))
      (eql pat inp) ) )    ; leaf node

(defun match (pat inp) (matchb pat inp '((t t))))
(defun matchb (pat inp bindings)
  (and bindings
    (if (consp pat)          ; interior node?
        (and (consp inp)
             (matchb (rest pat)
                     (rest inp)
                     (matchb (first pat)
                             (first inp) bindings)))
        (if (varp pat)       ; leaf: variable?
            (let ((binding (assoc pat bindings)))
              (if binding
                  (and (equal inp (second binding))
                       bindings)
                  (cons (list pat inp) bindings)))
            (and (eql pat inp) bindings)) ) ) )
```

Match Function in Java

```
public static Cons dummysub = list(list("t", "t"));

public static Cons match(Object pattern, Object input) {
    return matchb(pattern, input, dummysub); }

public static Cons
  matchb(Object pattern, Object input, Cons bindings) {
    if ( bindings == null ) return null;
    if ( consp(pattern) )
        if ( consp(input) )
            return matchb( rest( (Cons) pattern),
                           rest( (Cons) input),
                           matchb( first( (Cons) pattern),
                                   first( (Cons) input),
                                   bindings) );
        else return null;
    if ( varp(pattern) ) {
        Cons binding = assoc(pattern, bindings);
        if ( binding != null )
            if ( equal(input, second(binding)) )
                return bindings;
            else return null;
        else return cons(list(pattern, input), bindings); }
    if ( eql(pattern, input) )
        return bindings;
    return null; }
```

Transformation by Patterns

Matching and substitution can be combined to *transform* an input from a `pattern-pair`: a list of an input pattern and an output pattern.

```
(defun transform (pattern-pair input)
  (let (bindings)
    (if (setq bindings
              (match (first pattern-pair)
                     input))
        (sublis bindings
                (second pattern-pair))) ))

>(transform '((I aint got no ?x)
              (I do not have any ?x))

             '(I aint got no bananas))

(I DO NOT HAVE ANY BANANAS)
```

Transformation Patterns

Optimization:

```
(defpatterns 'opt
  '( ((+ ?x 0)            ?x)
     ((* ?x 0)             0)
     ((* ?x 1)            ?x)
     ((setq ?x (+ ?x 1))  (incf ?x)) ))
```

Language translation:

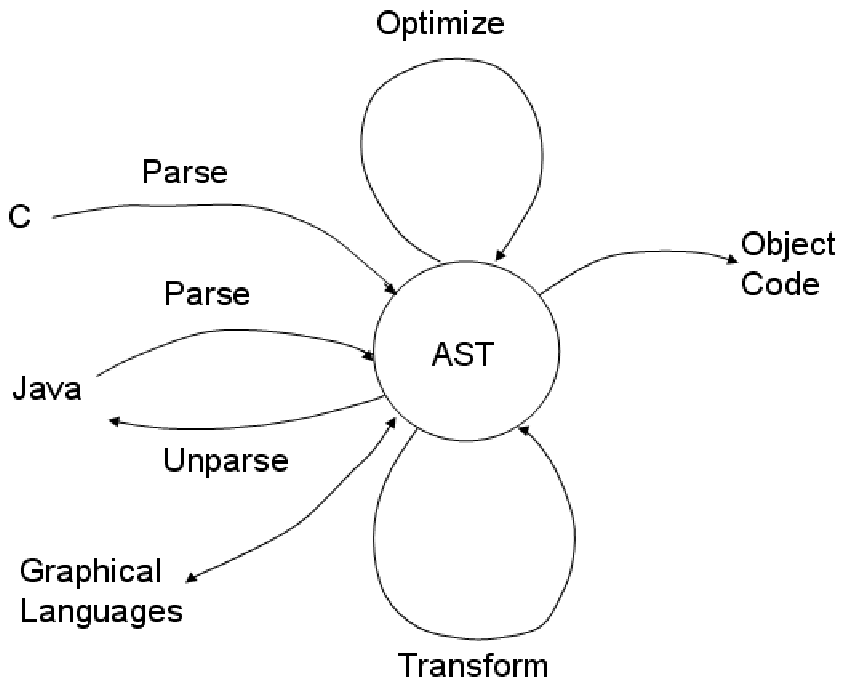
```
(defpatterns 'lisptojava
  '( ((aref ?x ?y)       ("" ?x "[" ?y "]"))
     ((incf ?x)          ("++" ?x))
     ((setq ?x ?y)       ("" ?x " = " ?y))
     ((+ ?x ?y)          ("(" ?x " + " ?y ")"))
     ((= ?x ?y)          ("(" ?x " == " ?y ")"))
     ((and ?x ?y)        ("(" ?x " && " ?y ")"))
     ((if ?c ?s1 ?s2)    ("if (" ?c ")" #\Tab
                           #\Return ?s1
                           #\Return ?s2))
```

Program Transformation using Lisp

```
>code
(IF (AND (= J 7) (/= K 3))
    (PROGN (SETQ X (+ (AREF A I) 3))
           (SETQ I (+ I 1))))

>(cpr (trans (trans code 'opt)
             'lisptojava))

if (((j == 7) && (k != 3)))
  {
    x = (a[i] + 3);
    ++i;
    }
```

Programs and Trees

- Fundamentally, programs are trees, sometimes called *abstract syntax trees* or *AST*.
- *Parsing* converts programs in the form of character strings (source code) into trees.
- It is easy to convert trees back into source code form (*unparsing*).
- Parsing - Transformation - Unparsing allows us to transform programs.



Set API in Java

`ArrayList` and `LinkedList` are inefficient, $O(n)$, for searching.

The `Set` interface is a `Collection` that does not contain duplicate (`.equals()`) elements. (A set-like object that allows duplicates is sometimes called a *bag*).

`TreeSet` maintains a set in sorted order, using the natural order of elements (`.compareTo()`), or a comparator. `add`, `remove`, and `contains` are $O(log(n))$.

The `add` method adds a new item; it returns `true` if the `add` succeeded, or `false` if the item was already there. Since a `Set` is a `Collection`, it has iterators.

```
Set<String>
   s = new TreeSet<String>
           ( new CaseInsensitiveCompare() );

s.add( "Hello" );
```

There are also a `HashSet` and a `LinkedHashSet`.

Map API in Java

A *map* $M : D \rightarrow R$ associates an element of its *domain* with an element of its *range*.

We can think of a map as a lookup table that allows us to look up information given a key value; for example, a telephone directory lets us look up a name to find a phone number: $Directory : Name \rightarrow Number$.

A `Map` is an interface to a `Collection` that associates keys with values. A key is unique, but different keys can map to the same value. A `TreeMap` maintains a map in sorted order; operations are $O(log(n))$. There is also a `HashMap`, $O(1)$.

```
boolean containsKey( KeyType key )
ValueType get ( KeyType key )
ValueType put ( KeyType key, ValueType value )
ValueType remove ( KeyType key )
```

`put` returns the old value associated with `key`, or `null` if there was no old value.

Iteration over a `Map`

A `Map` does not provide an iterator, so to iterate over it, it must be converted to a `Collection` that has an iterator. There are three possible collections: keys, values, and key-value pairs.

```
Set<KeyType> keySet()

Collection<ValueType> values()

Set<Map.Entry<KeyType,ValueType>> entrySet()
```

The latter option, `Map.Entry`, has methods for accessing the components:

```
KeyType getKey()
ValueType getValue()
ValueType setValue( ValueType newValue )
```

Array as a `Map`

Suppose that:

- The `KeyType` is an integer.
- The size of the domain (largest possible key minus smallest) is not too large.

In this case, the simplest map to use is an array:

- $O(1)$
- very simple code: `a[i]`
- no space used for keys

When it is applicable, *use an array* in your programs rather than a `switch` statement:

- runs faster
- uses less storage
- better software engineering

Avoid Repeated Code

Repetition of code that is almost the same is an indication that your program should be restructured: have only a single copy of the code, use tables for the differences.[4]

```
if (month == 1) {
    System.out.println("January");
} else if (month == 2) {
    System.out.println("February");
}
...

switch (month) {
  case 1:  System.out.println("January"); break;
  case 2:  System.out.println("February"); break;
  case 3:  System.out.println("March"); break;
  ...
  default: System.out.println("Invalid month.");bre
  }
```

Better:

```
System.out.println(months[month]);
```

[4]Sadly, the examples of bad code here are from Sun's Java Tutorial web site.

Initializing Array

Java makes it easy to initialize an array with constants:

```
String[] months = {"", "January", "February",
                   "March", ... };

String[] days = {"Monday", "Tuesday",
                 "Wednesday", ... };

String[][] phone = {{"Vader",     "555-1234"},
                    {"Skywalker", "472-2123"},
                    ... }
```

Key of a `Map`

Should you use a person's name as a key?

- 10,000 people named *Wang Wang* in Beijing.
- 100,000 people named *Ivan Ivanov Ivanovich* in Russia.
- 18 people (both sexes) with the same Chinese name at UT, 4 of them in CS and Computational Math.

Clearly, names do not make unique keys.

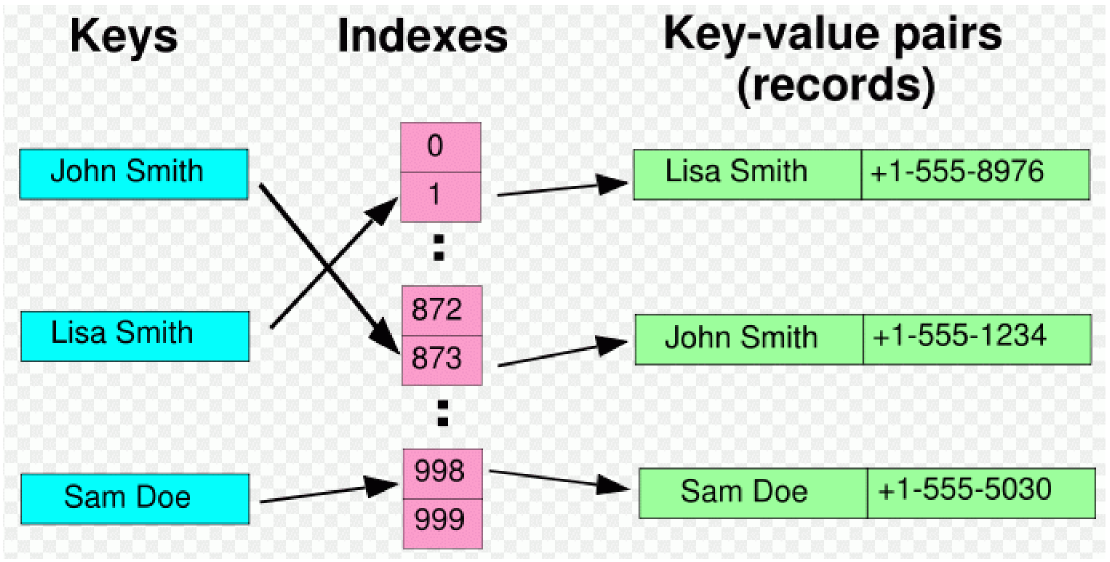
Hashing

We have seen that the array is the best way to implement a map if the key type is integer and the range of possible keys is not too large: access to an array is $O(1)$.

What if the key type does not fit these criteria? The idea of *hashing* is to use a *hash function*

$$h : KeyType \rightarrow integer$$

- Convert a key to an integer so that it can be used as an array index.
- Randomize, in the sense that two different keys usually produce different hash values, and hash values are equally likely.
- h should not be too expensive to compute.



Hash Function

We want a hash function to be easy to compute and to avoid *clustering* , the hashing of many keys to the same value.

One hash function that is usually good is to treat the key (e.g., a string of characters) as an integer and find the remainder modulo a prime p:

```
int hash = key % p;
```

Since this produces an integer from 0 to $p - 1$, we make the array size p.

The Java `.hashCode()` for `String` is roughly: [5]

```
public static int hashCode(String input) {
    int h = 0;
    int len = input.length();
    for (int i = 0; i < len; i++) {
        h = 31 * h + input.charAt(i);
    }
    return h; }
```

[5]from Wikipedia.

Java `.hashCode()`

The Java **`.hashCode()`** returns **`int`**, which could be positive or negative. To use the **`.hashCode()`** as an array index, make it positive (e.g. with **`Math.abs()`**) and make it modulo the table size.

The requirements of a **`.hashCode()`** include:

1. When called on the same object, **`.hashCode()`** must always return the same value during a given execution of a program (but could be different on different executions).
2. If two objects are **`.equals()`**, **`.hashCode()`** must return the same value for both.

The **`.hashCode()`** that is inherited from `Object` may use the memory address of an object as the hash code; this will differ between executions. Therefore, it is preferable to write your own **`.hashCode()`** for application data structures.

Exclusive OR

An important Boolean function is the bitwise *exclusive or* function $\oplus$:

a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

Sometimes called a *half-adder* or XOR, $\oplus$ is true if a or b is true, but not both. The Java operator for $\oplus$ is ^.

An important fact is that $(a \oplus b) \oplus b \ = \ a$.

Exclusive OR has many important uses:

- Cryptography: $text \oplus code$ looks like garbage, but another $\oplus code$ recovers the $text$.

Uses of Exclusive OR

- Hashing: hash functions are closely related to encryption.
- Graphics: $background \oplus picture$ paints $picture$, but another $\oplus picture$ erases it and restores $background$. This is especially good for animation and for cursors, which move across a background.
- Linked lists: both forward and backward pointers can be stored in the same space using $next \oplus previous$. Since we will be coming from either $next$ or $previous$, we can $\oplus$ with the link value to get the other one.
- Wireless networking: if message $(a \oplus b)$ is broadcast, those who know a can get b, and those who know b can get a.

Hash Function for Strings

It isn't always easy to find a good hash function; statistical tests are needed to ensure that the distribution of values is good. Java provides the function `hashCode()` that can be used for strings.

```
public static int jenkinshash(String key) {
    int hash = 0;
    for (int i = 0; i < key.length(); i++) {
        hash += key.charAt(i);
        hash += (hash << 10);
        hash ^= (hash >> 6); }
    hash += (hash << 3);
    hash ^= (hash >> 11);
    hash += (hash << 15);
    return hash; }
public static int hashfix(int hashcode, int size) {
    hashcode %= size;
    if ( hashcode < 0 ) hashcode += size;
    return hashcode; }
```

The function `jenkinshash` is by Bob Jenkins, *Dr. Dobbs Journal*, 1997; this returns a 32-bit integer that could be negative. The function `hashfix` makes a hash code positive and returns the hash code modulo the table `size`, which should be prime.

Hash Function for Application Types

Although Java has a default **`hashCode()`** method that is inherited by every class, it may be desirable to write a custom **`hashCode()`** method for application types:[6]

```
public class Employee{
    private int employeeId;
    private String firstName;
    private String lastName;
    private Department dept;

    public int hashCode() {
        int hash = 1;
        hash = hash * 31 + lastName.hashCode();
        hash = hash * 29 + firstName.hashCode();
        hash = hash * 17 + employeeId;
        hash = hash * 13 + (dept == null ? 0
                              : dept.hashCode());
        return hash;  } }
```

This illustrates a good pattern for combining hash codes: multiply one by a prime and add the other. XOR can also be used.

[6]example from Wikipedia.

Collisions

Even if the hash function is randomizing, it is inevitable that sometimes different keys will hash to the same value; this is called a *collision.*

The *load factor* λ is the ratio of hash table entries to table size. Obviously, the higher λ is, the greater the probability of a collision.

There are two basic ways to handle a collision:

- *Rehash* the key using a different hash function. The simplest method of rehashing is *linear probing*, simply adding 1 to the previous hash value (modulo table size). Other options are *quadratic probing* and *double hashing*, using a different hash function. With rehashing, the λ should be kept well below 1; this wastes some storage.
- Let each table entry point to a *bucket* of entries, an auxiliary data structure such as a linked list.

Hashing with Buckets

Hashing with buckets, also called *separate chaining*, is a simple method that works well. The array that is indexed by the hash function points to a linked list of entries for the items with that hash value; such a list is called a *bucket*.

Insertion is simply a push of a new linked list entry onto the list given by the array entry indexed by the hash value. Search is linked list search on that entry.

Expected time is $n/(2 \cdot tablesize)$. Although formally this is $O(n)$, in practice one can make it $O(1)$ by making $tablesize$ large enough.

If $tablesize$ is $n_{max}/10$, the expected time would be 5 comparisons, but the dedicated table size would not be too large.

Hashing with Buckets: Code

The basic code for hashing with buckets is very simple:

```
public void insert( String key ) {
   int hashindex = hashfix(key.hashCode(),
                           hashtable.length);
   hashtable[hashindex] =
        cons(key, hashtable[hashindex]); }

public boolean contains( String key ) {
   int hashindex = hashfix(key.hashCode(),
                           hashtable.length);
   Cons item =
     member(key, hashtable[hashindex]);
   return ( item != null ); }
```

Rehashing

If a fixed-size hash table is used with some secondary hash function to deal with collisions, the number of comparisons will become larger as λ increases, with a knee around $\lambda = 0.7$.

When this happens, it will be necessary to expand the array by:

- making a new array, larger by a factor of 1.5 or 2.
- rehashing all elements of the existing array into the new array.
- replacing the old array by the new one, letting the old one be garbage-collected.

The rehashing process is $O(n)$, but only has to be done once every n times, so its amortized cost is $O(1)$.

Hash Tables in Java

The Java library provides **`HashSet`** and **`HashMap`**; the item type for these must provide methods **`equals`** and **`hashCode`**.

A **`HashMap`**, $O(1)$, is a good choice when it is not necessary to be able to access the items in a sorted order; otherwise, use a **`TreeMap`**, $O(log(n))$.

In Java, a **`String`** object *caches* its hash code by storing the hash code in the **`String`** object the first time it is computed. Thereafter, the hash code can be gotten quickly. This is trading some space for time.

Extendible Hashing

Extendible hashing can be used when a table is too large to fit in main memory and must be placed on disk.

Extendible hashing is similar to the idea of a B-tree. Instead of having key ranges and disk addresses at the top level, we can hash the key to get an index into a table of disk addresses.

Uses of Hashing

There are many useful applications of hashing, including:

- A compiler keeps a *symbol table* containing all the names declared within a program, together with information about the objects that are named.
- A hash table can be used to map *names* to *numbers*. This is useful in graph theory and in networking, where a domain name is mapped to an IP address: `willie.cs.utexas.edu` = `128.83.130.16`
- Programs that play games such as chess keep hash tables of board positions that have been seen and evaluated before. In general, hashing is a good way to look up items that do not have a natural sort order.
- The Rabin-Karp string search algorithm uses hashing to tell whether strings of interest are present in the text being searched. This algorithm is used in plagiarism detection and DNA matching.
- If an expensive function may be called repeatedly with the same argument value, pairs of (*argument*, *result*) can be saved in a hash table, with *argument* as the key, and *result* can be reused. This is called *memoization*.

Randomization

Hashing is our first example of *randomized algorithms*, since a hash function is a somewhat random function of the key.

Randomized algorithms can be used to avoid clustering.

As an example, suppose that a table contains an equal number of a and b. The worst case of searching for a b with any predefined strategy is $O(n)$; for example, a linear search would be $O(n)$ if all the a entries are at the front.

If we use a randomized search, though, the expected time is $O(1)$ and the probability of a $O(n)$ search is very small. The expected time forms a *geometric series*:

$$1/2 + 1/4 + 1/8 + 1/16 + ... = 1$$

- Randomized time delays for retry can prevent collisions.
- Randomized choice of a communication path or server can avoid failures.

Priority Queue

A *priority queue* is conceptually an array of queues, indexed by priority. To remove an item from the priority queue, the first item of the highest-priority non-empty queue is removed and returned.

In an operating system, there often is a fixed and small number of priorities. A high-priority process, such as responding to a device interrupt, can *interrupt* a lower-priority process such as a user program. The processes that are ready to run are kept on a priority queue. The high-priority processes are short but need fast service. Whenever a process ceases execution, the operating system removes the next highest-priority process from the *ready queue* and runs it.

If we use an array of circular queues or two-pointer queues, both insertion and removal are $O(1)$.

We will assume that the highest-priority item is the one with the lowest priority number. The operations on the priority queue are `insert` and `deleteMin`. (We will discuss a *min queue*; a *max queue* could be implemented similarly.)

Priority Queue with Array or Binary Tree

If the number of possible priorities is small and fixed, as in an operating system, an array of queues can be used, with the priority being the array index.

If there are many possible priority values, an easy way to implement a priority queue is to use a binary search tree such as an AVL tree, indexed by priority, with a queue at the leaves.

This makes both `insert` and `deleteMin` be $O(log(n))$.

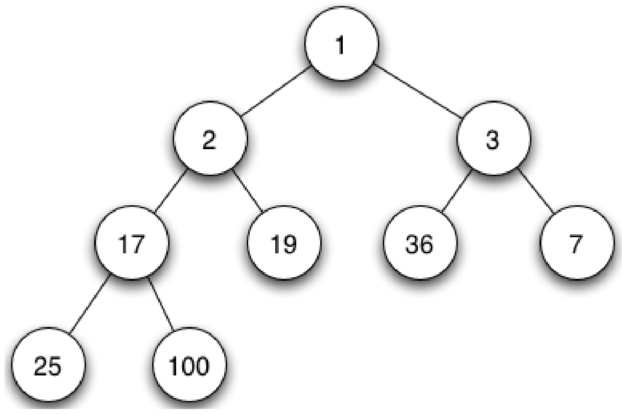
However, we can do slightly better with less cumbersome machinery.

Binary Heap

A *binary heap* is a binary tree that is mapped onto an array. Because of the mapping that is used, links between nodes are easily computed without storing them.

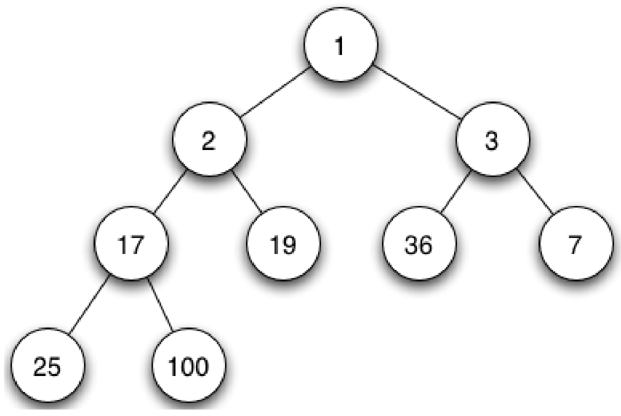
A heap has two fundamental properties:

- **Structure Property:** The heap is a *complete* binary tree, meaning that all nodes are filled except for the right-hand part of the bottom row.



- **Heap Order Property:** For any subtree of the heap, the value at the root is less than the value of any of its descendants. Since the minimum element is at the root, it is easy to find.

Mapping to Array



This heap maps to an array:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	1	2	3	17	19	36	7	25	100	

The links for an element at position i are easily computed:

- parent: $i/2$
- left child: $2 * i$
- right child: $2 * i + 1$

Insertion into Heap

`insert` will place a new element in the next available array position; this keeps the binary tree complete.

To maintain the heap order property, we must *percolate up* the new value until it reaches its proper position. This is easy: if the new item is less than its parent, swap the new item with its parent; then percolate up from the parent.

```
public void insert( AnyType item ) {
  if ( currentSize >= array.length - 1 )
     enlargeArray( array.length * 2 + 1 );
  int hole = ++currentSize;
  while ( hole > 1 &&
          item.compareTo( array[hole / 2] ) < 0 )
    { array[hole] = array[hole / 2];     // swap
      hole /= 2; }      // change hole to parent
  array[hole] = item; }
```

This is simple and gives $O(log(n))$ `insert`.

Removal from Heap

Finding the item to remove with `deleteMin` is easy: the smallest item is at the root, array position `[1]`. Removing the top element creates a hole at the top, and we must somehow move the last element in the array into the hole in order to keep the tree complete.

To maintain the order property, we see whether the last element can be put into the hole. If so, we are done; if not, exchange the hole with its smaller child and repeat.

```
public AnyType deleteMin( ) {
  AnyType top = array[1];
  int hole = 1;
  boolean done = false;
  while ( hole * 2 < currentSize && ! done )
   { int child = hole * 2;
     if ( array[child]
             .compareTo( array[child + 1] ) > 0 )
        child++;    // child now points to smaller
     if (array[currentSize]
             .compareTo( array[child] ) > 0 )
        { array[hole] = array[child];
        hole = child; }
      else done = true; }
  array[hole] = array[currentSize--];
  return top; }
```

Uses of Priority Queues

Priority queues have many uses in Computer Science:

- Operating systems use priority queues to run the most important jobs first, print the shortest files first, etc.
- *Discrete event simulation* simulates a system by executing programs that represent *events*; an event will cause future events. A priority queue is used to store events, with the scheduled time of an event as the priority. `deleteMin` removes the event that will occur next, and time is jumped forward to its time.
- *Greedy algorithms* are algorithms that try the thing that looks the best, in hopes of finding a solution quickly. A priority queue can store possibilities, with the priority indicating how good they look.

`PriorityQueue` in Java

The Java library provides a generic class `PriorityQueue`:

- `add(item)` is the method we called `insert`
- `peek()` returns, but does not remove, the minimum element
- `poll()` removes and returns the minimum element, the method we called `deleteMin()` above.

The `PriorityQueue` can be set up using `.compareTo()` to order events according to their natural ordering, or using a specified `Comparator`.

Example Discrete Event Simulation

A traffic light cycles through green, yellow, and red; this can easily be simulated:

```
public static void
  red (PriorityQueue pq, int time) {
      turnOff("green");
      turnOff("yellow");
      turnOn("red");
      pq.add(new Event("green", time + 60));
      }
```

This program is typical of Discrete Event Simulation:

- It may examine the state of the world at the time it is executed.
- It performs one or more simple actions, in this case turning on the red light and turning off the others.
- It schedules one or more new events for future times, i.e. the green event for the current time plus 60 seconds.

Sorting

Sorting a set of items into order is a very common task in Computer Science.

There are a number of bad algorithms that sort in $O(n^2)$ time; we will not discuss those much. We have already seen algorithms that do sorting in $O(n \cdot log(n))$ time, and in fact it can be proved that that is the best possible.

An *internal sort* is performed entirely in main memory. If the set of items is too large to fit in memory, an *external sort* using disk or other external storage can be done.

Sorting is based on *comparison* between items. In general, complex data can be compared and sorted in many ways. It is common to furnish a comparison function to the sorting program:

```
>(sort '(32 29 62 75 48 14 80 98 28 19) '<)

(14 19 28 29 32 48 62 75 80 98)


>(sort '(32 29 62 75 48 14 80 98 28 19) '>)

(98 80 75 62 48 32 29 28 19 14)
```

Comparison in Java

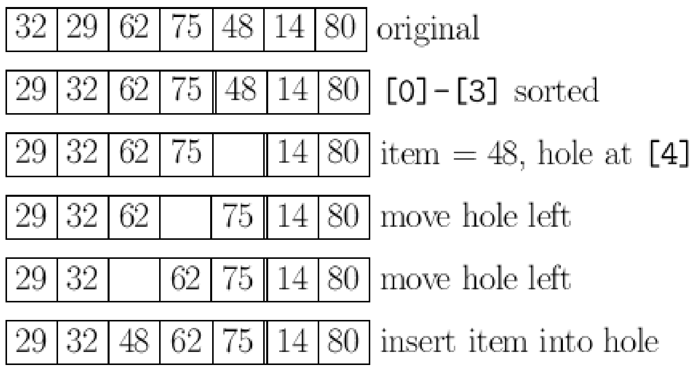
Java does not allow a function to be passed as a function argument, and its various types must be compared in different ways:

- The primitive types `int`, `long`, `float` and `double` are compared using `<` and `>`. They cannot use `.compareTo()` .
- `String` uses `.compareTo()`; it cannot use `<` and `>`.
- An application object type can be given a `.compareTo()` method, but that allows only one way of sorting.
- A `Comparator` can be passed as a function argument, and this allows a custom comparison method.

```
public static void
  mySort( AnyType[] a,
          Comparator<? super AnyType> cmp) {...}

class MyOrder implements Comparator<MyObject>
  {
  public int compare(MyObject x, MyObject y)
    { return ( x.property() - y.property() ); }}
```

Insertion Sort

Insertion sort is similar to the way people sort playing cards: given some cards that are sorted and a new card to be added, make a hole where the new card should go and put the new card into the hole.


32 29 62 75 48 14 80 original
29 32 62 75 48 14 80 [0]-[3] sorted
29 32 62 75 14 80 item = 48, hole at [4]
29 32 62 75 14 80 move hole left
29 32 62 75 14 80 move hole left
29 32 48 62 75 14 80 insert item into hole

```
public static void insertionSort( Integer[] a ) {
  for ( int i = 1; i < a.length; i++ )
    {   int hole;
        Integer item = a[i];
        for ( hole = i;
              hole > 0 &&
                item.compareTo( a[hole - 1] ) < 0;
              hole-- )
            a[hole] = a[hole - 1];
        a[hole] = item; } }
```

Insertion Sort Performance

Since insertion sort has a triangle of an outer loop and an inner loop that grows to the index of the outer loop, it is $O(n^2)$. This would be unacceptable unless n is small.

However, if the input is almost sorted, insertion sort can run quickly, $O(n+d)$, where d is the number of *inversions* or pairs of items that are not in correct order. The reason for this is that the hole is usually in the right place already, so the inner loop terminates quickly. Real applications often have almost-sorted input, e.g. the telephone book with a few new customers.

Insertion sort is:

- *stable*: does not change the relative position of items with equal keys.
- *in-place*: does not require any additional storage.
- *on-line*: can sort items as it receives them one at a time.

Heapsort

We have already seen that a Heap can store a set of items and return the smallest one, in $O(log(n))$ time per item.

Therefore, an easy way to sort would be to put all the items into a heap, then remove items one at a time with `deleteMin` and put them into the output array.

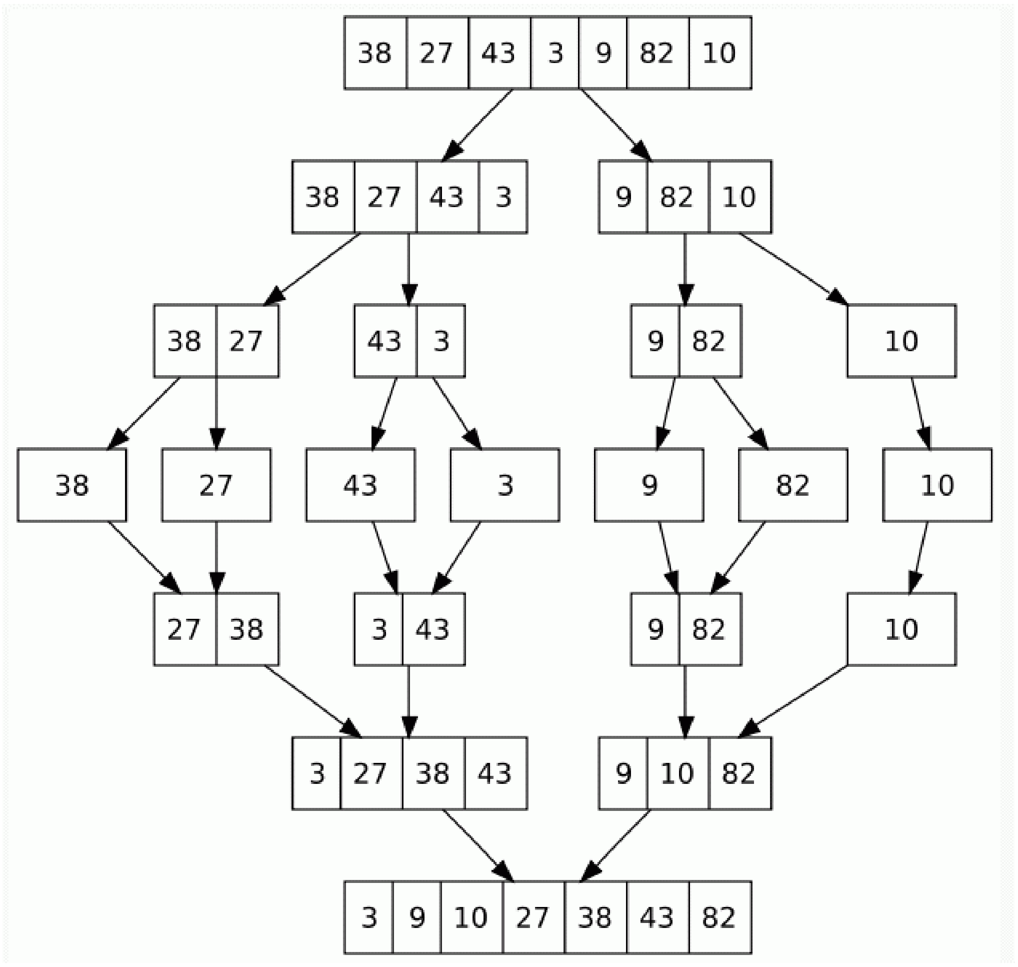
The problem with this easy method is that it uses an extra array. However, we can do it using the original array:

1. Make the original array into a max heap. This can be done in $O(n)$ time.
2. For the size of the heap,
   (a) Remove the largest item from the heap. This makes the heap smaller, so that there is a hole at the end of the heap.
   (b) Put the largest item into the hole.

This gives a sort in $O(n \cdot log(n))$ time. Heapsort is in-place, but not stable.

Merge Sort

We have already seen Merge Sort with linked lists. The idea with arrays is the same: break the array in half, sort the halves, and merge the halves into a sorted whole.



One problem with Merge Sort is that it is not in-place: it requires an extra array of size n. Although it can be coded to be in-place, this is very complicated.

Merge Sort Performance

As we have seen, Merge Sort is $O(n \cdot log(n))$. Despite requiring extra storage, Merge Sort has a number of advantages:

- 39% fewer comparisons than Quicksort
- Merge Sort is inherently sequential:
  - has good *memory locality* and cache performance
  - can be done with off-line media such as disk
  - can be parallelized easily: parts of the input array can be sorted independently by separate computers.
- stable
- can efficiently merge in updates to an existing sorted array
- If the highest element in the low sublist is less than the lowest element in the high sublist, the merge can be omitted.

Memory Hierarchy and Locality

Most computers have a *memory hierarchy* :

- Fastest memory is most expensive and smallest, e.g. 1 KB registers.
- On-chip memory is expensive and small, e.g. 1 MB cache.
- Medium-speed memory is larger, e.g. 1 GB main memory.
- Slowest memory is cheapest and largest, e.g. 1 TB disk.

Operations can only be performed on registers. There is a continual migration of data between slower memories and faster memories where it can be processed.

Computer hardware and software is often optimized to run faster when there is *memory locality* , i.e. successive accesses to memory are in nearby locations by memory address.

Algorithms that step through an array using a `for` loop to access adjacent locations are likely to be faster than algorithms that jump around.

Does Array Index Order Matter?

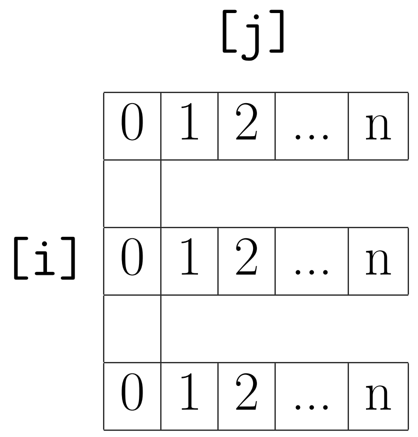
```
static double arr[1000][1000];

double test1 ()
{ double sum; int i, j;
  sum = 0.0;
  for (i=0; i<1000; i++)
    for (j=0; j<1000; j++)
      sum = sum + arr[i][j];
  return (sum); }

double test2 ()
{ double sum; int i, j;
  sum = 0.0;
    for (j=0; j<1000; j++)
  for (i=0; i<1000; i++)
      sum = sum + arr[i][j];
  return (sum); }
```

The two programs compute the same result, but performance is quite different:

```
test1 result = 1000000.0    time =  430000
test2 result = 1000000.0    time = 1940000
```

Array Storage and Indexing

Most modern computer languages (except Fortran) use *row-major order*, in which elements of a matrix row are adjacent in memory: `A[i][j]` is stored as:



To get the best performance, `j` should be the index of the inner loop, so that `j` will vary fastest and accesses will be adjacent:

```
for (i=0; i<1000; i++)
  for (j=0; j<1000; j++)
    sum = sum + arr[i][j];
```

Quicksort

As its name suggests, *Quicksort* is one of the better sort algorithms; it is $O(n \cdot log(n))$ (though it can be $O(n^2)$ in the worst case). In practice it is significantly faster than other algorithms.

Quicksort is a divide-and-conquer algorithm that chooses a *pivot* value, then *partitions* the array into two sections with the pivot in its final position in the middle:

$elements \leq pivot$	$pivot$	$elements > pivot$

The outside sections are then sorted recursively.

The partitioning can be done in-place, making Quicksort an in-place sort. Since partitioning is done in-place by swapping elements, Quicksort is not stable.

Quicksort Code[7]

```
public static void quicksort(
  Integer[] a, int lo, int hi ) {
    int i=lo, j=hi; Integer h;
    Integer pivot = a[(lo+hi)/2];

    do                              //  partition
    {
        while (a[i] < pivot) i++;           // move
        while (a[j] > pivot) j--;
        if (i<=j)
        {
            h=a[i]; a[i]=a[j]; a[j]=h;      // swap
            i++; j--;
        }
    } while (i<=j);

    if (lo<j) quicksort(a, lo, j);  //  recursion
    if (i<hi) quicksort(a, i, hi);
}
```

[7]This version of Quicksort is from H. W. Lang, Fachhochschule Flensburg, `http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/quick/quicken.htm`

Partitioning

`lo`									`hi`	initial
32	29	62	75	48	14	80	98	28	19	
`i`									`j`	pivot = *48*
32	29	**62**	**75**	*48*	14	**80**	**98**	28	19	
		`i`							`j`	move i and j
32	29	**62**	**75**	*48*	14	**80**	**98**	28	19	
			`i`					`j`		swap
32	29	19	**75**	*48*	14	**80**	**98**	28	**62**	
				`i`			`j`			swap
32	29	19	28	*48*	14	**80**	**98**	**75**	**62**	
				`i`	`j`					move
32	29	19	28	*48*	14	**80**	**98**	**75**	**62**	
`lo`				`j`	`i`				`hi`	swap
32	29	19	28	14	*48*	**80**	**98**	**75**	**62**	

Quicksort Example

```
Quicksort:  lo 0  hi 9   [32, 29, 62, 75, 48, 14, 80, 98, 28, 19]
Partition:   i 5   j 4   [32, 29, 19, 28, 14, 48, 80, 98, 75, 62]
Quicksort:  lo 0  hi 4   [32, 29, 19, 28, 14]
Partition:   i 2   j 1   [14, 19, 29, 28, 32]
Quicksort:  lo 0  hi 1   [14, 19]
Partition:   i 1  j -1   [14, 19]
   sorted:  lo 0  hi 1   [14, 19]
Quicksort:  lo 2  hi 4           [29, 28, 32]
Partition:   i 3   j 2           [28, 29, 32]
Quicksort:  lo 3  hi 4               [29, 32]
Partition:   i 4   j 2               [29, 32]
   sorted:  lo 3  hi 4               [29, 32]
   sorted:  lo 2  hi 4           [28, 29, 32]
   sorted:  lo 0  hi 4   [14, 19, 28, 29, 32]
Quicksort:  lo 5  hi 9                       [48, 80, 98, 75, 62]
Partition:   i 9   j 8                       [48, 80, 62, 75, 98]
Quicksort:  lo 5  hi 8                       [48, 80, 62, 75]
Partition:   i 8   j 7                       [48, 75, 62, 80]
Quicksort:  lo 5  hi 7                       [48, 75, 62]
Partition:   i 7   j 6                       [48, 62, 75]
Quicksort:  lo 5  hi 6                       [48, 62]
Partition:   i 6   j 4                       [48, 62]
   sorted:  lo 5  hi 6                       [48, 62]
   sorted:  lo 5  hi 7                       [48, 62, 75]
   sorted:  lo 5  hi 8                       [48, 62, 75, 80]
   sorted:  lo 5  hi 9                       [48, 62, 75, 80, 98]
   sorted:  lo 0  hi 9   [14, 19, 28, 29, 32, 48, 62, 75, 80, 98]
```

Quicksort Performance

Quicksort usually performs quite well; we want to avoid the $O(n^2)$ worst case and keep it at $O(n \cdot log(n))$.

The choice of pivot is important; choosing the first element is a very bad choice if the array is almost sorted. Choosing the median of the first, middle, and last elements makes it very unlikely that we will get a bad choice.

IntroSort ("introspective sort") changes from Quicksort to a different sort for some cases:

- Change to Insertion Sort when the size becomes small (≤ 20), where Insertion Sort may be more efficient.
- Change to Heapsort after a certain depth of recursion, which can protect against the unusual $O(n^2)$ worst case.

Quicksort is easily parallelized, and no synchronization is required: each subarray can be sorted independently.

Radix Sort

Radix sort is an old method that is worth knowing because it can be used as an external sort for data sets that are too large to fit in memory.

We will assume that we are sorting integers in decimal notation; in modern practice, groups of bits would be more sensible.

The idea of radix sort is to sort the input into bins based on the lowest digit; then combine the bins in order and sort on the next-highest digit, and so forth.

The bins can be mostly on external storage media, so that the size of the data to be sorted can exceed the size of memory.

The sorting process can also be parallelized.

The performance of radix sort is $O(n \cdot k)$ where k is the key length. It makes sense to think of k as being approximately $log(n)$, but if there many items for each key, radix sort would be more efficient than $O(n \cdot log(n))$.

Radix sort is stable.

Radix Sort Example

Original: `(32 29 62 75 48 14 80 98 28 19)`

0	`(80)`
1	`()`
2	`(32 62)`
3	`()`
4	`(14)`
5	`(75)`
6	`()`
7	`()`
8	`(48 98 28)`
9	`(29 19)`

Sorted into bins on lowest digit

Appended: `(80 32 62 14 75 48 98 28 29 19)`

0	`()`
1	`(14 19)`
2	`(28 29)`
3	`(32)`
4	`(48)`
5	`()`
6	`(62)`
7	`(75)`
8	`(80)`
9	`(98)`

Sorted into bins on second digit

Appended: `(14 19 28 29 32 48 62 75 80 98)`

Sorting in Java Library

Java provides generic `sort` methods for arrays:

```
void sort( Object [] arr )

void sort( Object [] arr, Comparator cmp )
```

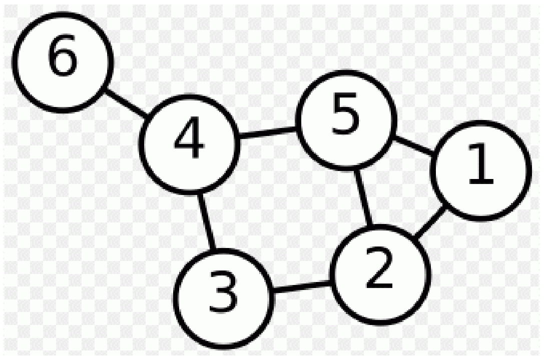
The first method uses the natural `compareTo` method, while the second allows a `Comparator` to be specified. The sort algorithm is a modified mergesort, stable and guaranteed $O(n \cdot log(n))$.

Since the `Collection` interface provides a `toArray` method, it is easy to sort any `Collection` by first converting it to an array.

The Java `List` class provides `sort` methods that operate by dumping the List into an array, sorting the array, and then resetting the List elements in the sorted order.

Graphs

A *graph* $G = (V, E)$ has a set of *vertices* or *nodes* V and a set of *edges* or *arcs* or *links* E, where each edge connects two vertices. We write this mathematically as $E \subseteq V \times V$, where $\times$ is called the *Cartesian product* of two sets. We can write an edge as a pair (v_1, v_2), where v_1 and v_2 are each a vertex.



A *path* is a sequence of vertexes connected by edges: $v_1, v_2, ..., v_n$ where $(v_i, v_{i+1}) \in E$. A *simple path* is a path with no nodes repeated, except possibly at the ends. The *length* of a path is the number of edges in it. A *cycle* is a path from a node back to itself; a graph with no cycles is *acyclic.*

An edge may have a *weight* or *cost* associated with it.

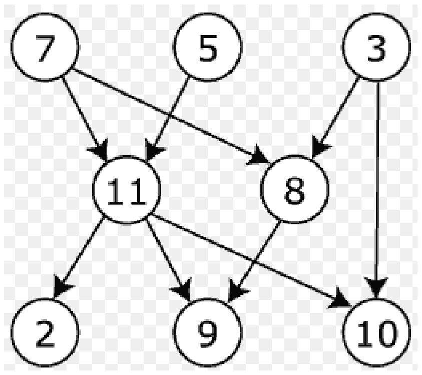
Examples of Graphs

There are many examples of graphs:

- The road network forms a graph, with cities as vertices and roads as edges. The distance between cities can be used as the cost of an edge.
- The airline network is a graph, with airports as vertices and airline flights as edges.
- Communication networks such as the Internet: computers and switches are nodes, connections between them are links.
- *Social networks* such as the graph of people who call each other on the telephone, or friends on MySpace: people are nodes and there are links to the people they communicate with.
- Distribution networks that model the flow of goods: an oil terminal is a node, and an oil tanker or pipeline is a link.

Directed Acyclic Graph

If connections are one-way, from v_1 to v_2, we say the graph is *directed* ; otherwise, it is *undirected.* A directed graph is sometimes called a *digraph. Directed acyclic graphs* or *DAG* representations such as trees are important in Computer Science.



Graph Representations

We want the internal representation of a graph to be one that can be efficiently manipulated.

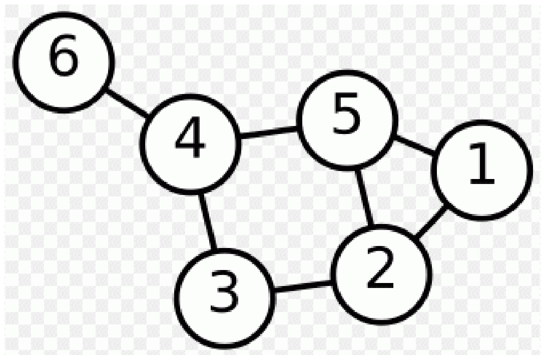
If the external representation of a node is a string, such as a city name, we can use a `Map` or symbol table to map it to an internal representation such as:

- a node number, convenient to access arrays of information about the node
- a pointer to a node object.

A graph is called *dense* if $|E| = O(|V|^2)$; if $|E|$ is less, the graph is called *sparse* . Most graphs used in applications are sparse.

Adjacency List

In the *adjacency list* representation of a graph, each node has a list of nodes that are *adjacent* to it, i.e. connected by an edge. A linked list is a natural representation.



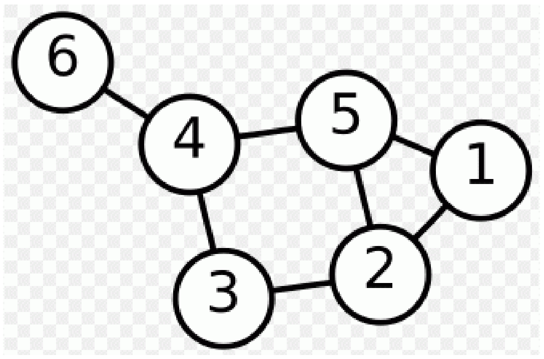
```
1 (5 2)
2 (3 5 1)
3 (4 2)
4 (6 3 5)
5 (4 2 1)
6 (4)
```

This graph is undirected, so each link is represented twice.

The storage required is $O(|V| + |E|)$. This is a good representation if the graph is *sparse*, i.e. each node is not linked to many others.

Adjacency Matrix

In the *adjacency matrix* representation of a graph, a *Boolean matrix* contains a 1 in position (i, j) iff there is a link from v_i to v_j, otherwise 0.



	1	2	3	4	5	6
1	0	1	0	0	1	0
2	1	0	1	0	1	0
3	0	1	0	1	0	0
4	0	0	1	0	1	1
5	1	1	0	1	0	0
6	0	0	0	1	0	0

Since this graph is undirected, each link is represented twice, and the matrix is *symmetric*.

The storage required is $O(|V|^2)$; even though only one bit is used for each entry, the storage can be excessive.

Implicit Graphs

Some graphs must be represented implicitly because they cannot be represented explicitly. For example, the graph of all possible chess positions is larger than the number of elementary particles in the universe. In such cases, only part of the graph will be explicitly considered, such as the chess positions that can be reached from the current position in 7 moves or less.

Topological Sort

Some graphs specify an order in which things must be done; a common example is the course prerequisite structure of a university.

A *topological sort* orders the vertices of a directed acyclic graph (DAG) so that if there is a path from vertex v_i to v_j, v_j comes after v_i in the ordering. A topological sort is not necessarily unique. An example of a topological sort is a sequence of taking classes that is legal according to the prerequisite structure.

An easy way to find a topological sort is:

- initialize a queue to contain all vertices that have no incoming arcs.
- While the queue is not empty,
  - remove a vertex from the queue,
  - put it into the sort order
  - remove all of its arcs
  - If the target of an arc now has zero incoming arcs, add the target to the queue.

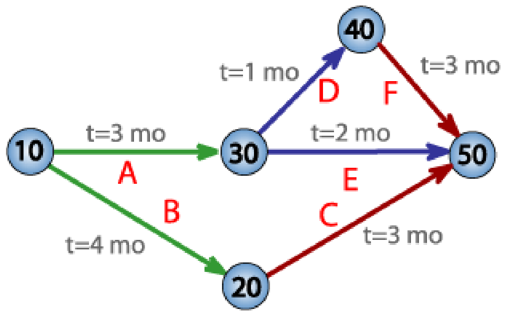
Uses of Topological Sort

Topological Sort has many uses:

- PERT technique for scheduling of tasks
- Instruction scheduling in compilers
- Deciding what to update in spreadsheets
- Deciding what files to re-compile in makefiles
- Resolving symbol dependencies in linkers.

PERT Chart

PERT, for Program Evaluation and Review Technique, is a project management method using directed graphs.



- Nodes represent events, milestones, or time points.
- Arcs represent activities, which take time and resources. Each arc is labeled with the amount of time it takes.
- A node has the maximum time of its incoming arcs.
- An activity cannot start before the time of its preceding event.
- The *critical path* is the longest path from start to finish.
- The *slack* is the amount of extra time available to an activity before it would become part of the critical path.

PERT Chart: Calculating Times

The time of events can be found as follows:

- The time of the initial node is 0.
- For each node j in the topological sort after the first,

$$time_j = max_{(i,j) \in E}(time_i + cost_{i,j})$$

By considering nodes in topological sort order, we know that the time of each predecessor will be computed before it is needed.

We can compute the latest completion time for each node, in reverse topological order, as:

- The latest time of the final node n is $time_n$.
- For node i,

$$latest_i = min_{(i,j) \in E}(latest_j - cost_{i,j})$$

Slack for an edge (i, j) is:

$$slack_{i,j} = latest_j - time_i - cost_{i,j}$$

Shortest Path Problem

An important graph problem is the *shortest path* problem, namely to find a path from a start node to a goal node such that the sum of weights along the path is minimized. We will assume that all weights are non-negative.

Shortest path *routing* algorithms are used by web sites that suggest driving directions, such as MapQuest.

Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path to all nodes in a weighted graph from a specified starting node.

Dijkstra's algorithm is a good example of a *greedy* algorithm, one that tries to follow the best-looking possibility at each step.

The basic idea is simple:

- Set the cost of the start node to 0, and all other nodes to ∞.
- Let the current node be the lowest-cost node that has not yet been visited. Mark it as visited. For each edge from the current node, if the sum of the cost of the current node and the cost of the edge is less than the cost of the destination node,
  - Update the cost of the destination node.
  - Set the parent of the destination node to be the current node.

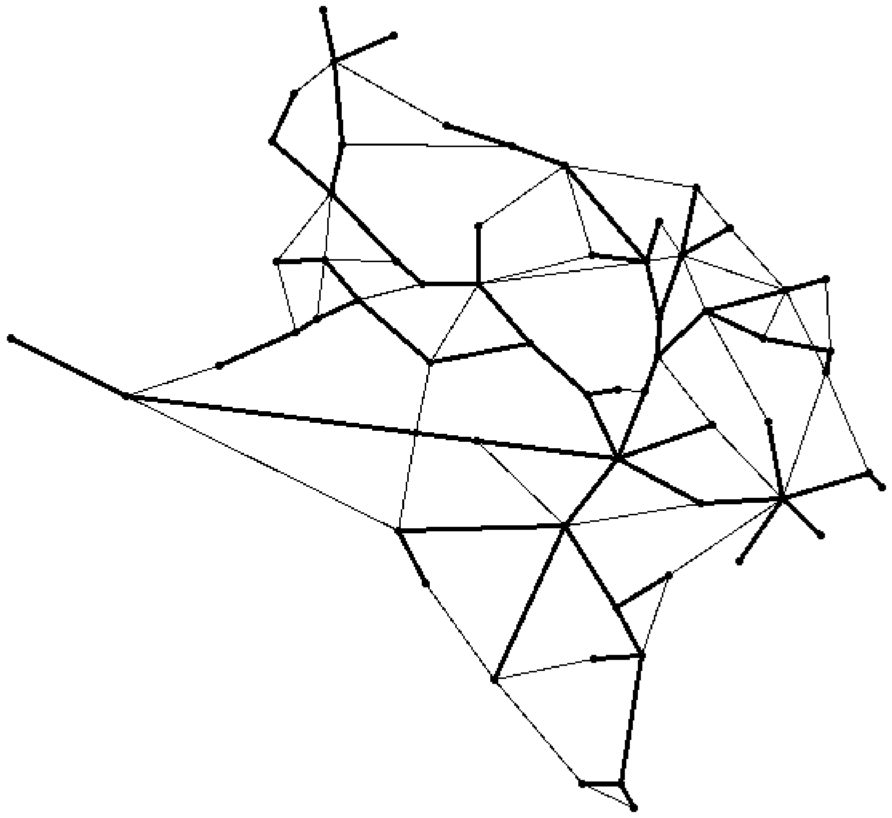
When we get done visiting all nodes, each node has a cost and a path back to the start; we can reverse that to get a forward path.

Dijkstra's Algorithm

```
public void dijkstra( Vertex s ) {
  for ( Vertex v : vertices ) {
    v.visited = false;
    v.cost = 999999; }
  s.cost = 0;
  s.parent = null;
  PriorityQueue<Vertex>
      fringe = new PriorityQueue<Vertex>(20,
        new Comparator<Vertex>() {
          public int compare(Vertex i, Vertex j) {
              return (i.cost - j.cost); }});
  fringe.add(s);
  while ( ! fringe.isEmpty() ) {
    Vertex v = fringe.remove();  // lowest-cost
    if ( ! v.visited )
      { v.visited = true;
        for ( Edge e : v.edges )
          { int newcost = v.cost + e.cost;
            if ( newcost < e.target.cost )
             { e.target.cost = newcost;
               e.target.parent = v;
               fringe.add(e.target);  } } } } }
```

Dijkstra's Algorithm Example

Dijkstra's Algorithm finds the shortest path to all nodes from a given start node, producing a tree.



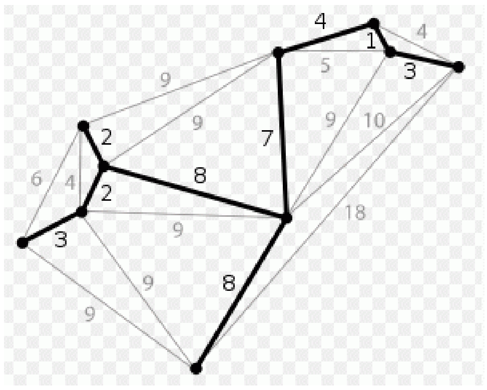
Minimum Spanning Tree

A *minimum spanning tree* is a subgraph of a given undirected graph, containing all the nodes and a subset of the arcs, such that:

- All nodes are connected.
- The resulting graph is a tree, i.e. there are no cycles.
- The sum of weights on the remaining arcs is as low as possible.

Example: Connect a set of locations to the Internet using a minimum length of cable.

The minimum spanning tree may not be unique, but all MST's will have the same total cost of arcs.



Prim's Algorithm

Prim's Algorithm for finding a minimum spanning tree is similar to Dijkstra's Algorithm for shortest paths. The basic idea is to start with a part of the tree (initially, one node of the graph) and add the lowest-cost arc between the existing tree and another node that is not part of the tree.

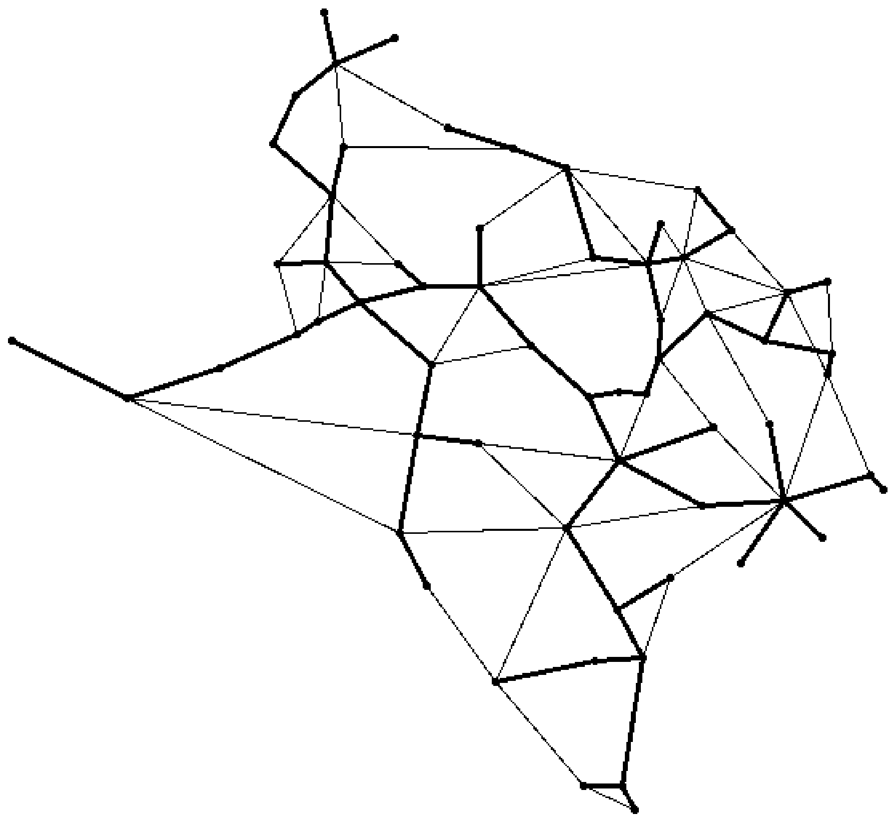
As with Dijkstra's Algorithm, each node has a parent node pointer and a cost, which is the least cost of an arc connecting to the tree that has been found so far.

Prim's Algorithm

```
public void prim( Vertex s ) {
  for ( Vertex v : vertices ) {
    v.visited = false;
    v.parent = null;
    v.cost = 999999; }
  s.cost = 0;
  PriorityQueue<Vertex>
      fringe = new PriorityQueue<Vertex>(20,
        new Comparator<Vertex>() {
          public int compare(Vertex i, Vertex j) {
              return (i.cost - j.cost); }});
  fringe.add(s);
  while ( ! fringe.isEmpty() ) {
    Vertex v = fringe.remove();  // lowest-cost
    if ( ! v.visited )
      { v.visited = true;
        for ( Edge e : v.edges )
          {  if ( (! e.target.visited) &&
                  ( e.cost < e.target.cost ) )
               { e.target.cost = e.cost;
                 e.target.parent = v;
                 fringe.add(e.target); } } } } }
```

Prim's Algorithm Example

Prim's Algorithm finds a way to connect all nodes as a tree for the minimum total cost.



Directed Search

Dijkstra's algorithm finds the shortest path to *all* nodes of a graph from a given starting node. If the graph is large and we only want a path to a single destination, this is inefficient.

We might have some *heuristic* information that gives an estimate of how close a given node is to the goal.

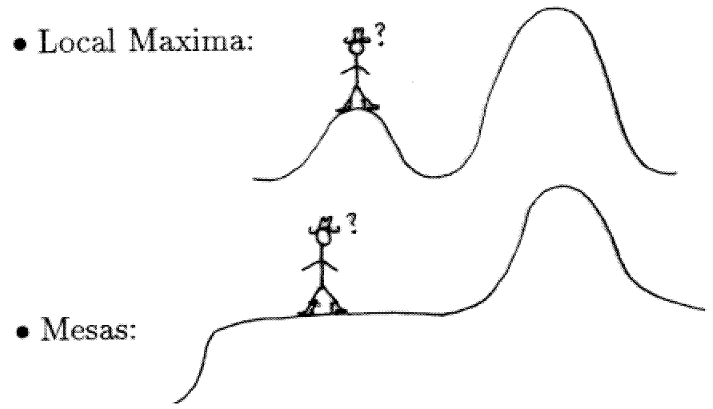
Using the heuristic, we can search the more promising parts of the graph and ignore the rest.

Hill Climbing

A strategy for climbing a hill in a fog is to move upward.

A heuristic that estimates distance to the goal can be used to guide a hill-climbing search. A discrete depth-first search guided by such a heuristic is called *greedy best-first search*; it can be very efficient. For example, in route finding, hill climbing could be implemented by selecting the next city that is closest to the goal.

Unfortunately, hill-climbing sometimes gets into trouble on a *local maximum*:



Heuristic Search: A*

The A* algorithm uses both actual distance (as in Dijkstra's algorithm) and a heuristic estimate of the remaining distance. It is both very efficient and able to overcome local maxima.

A* chooses the next node based on *lowest estimated total cost* of a path through the node.

Estimated Total Cost $f(n) = g(n) + h(n)$
$g(n)$ = Cost from Start to n [known]
$h(n)$ = Cost from n to Goal [estimated]

```
             g(n)              h(n)?
Start    ------->    n    ------->    Goal
```

The *heuristic function*, h, estimates the cost of getting from a given node n to the goal. If h is good, the search will be highly directed and efficient; for example, airline distance is an excellent heuristic distance estimator for route finding. If h is not so good or has pathologies, inclusion of the known cost g keeps the search from getting stuck or going too far astray.

A* Algorithm

The A* algorithm is similar to Dijkstra's algorithm, except that it puts entries into the priority queue based on the f value (estimated total cost) rather than g value (lowest cost found so far).

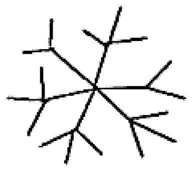
Note that A* does not recognize that a node is a goal until the node is removed from the priority queue.

Theorem: If the h function does not over-estimate the distance to the goal, A* finds an optimum path.

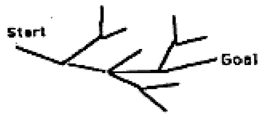
A* will get the same result as Dijkstra's algorithm while doing less work, since Dijkstra searches all nodes while A* searches only nodes that may be on a path to the goal.

Ordered Search for Route Finding

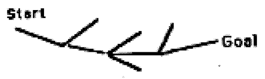
- $h(n) = 0$: Search proceeds in all directions



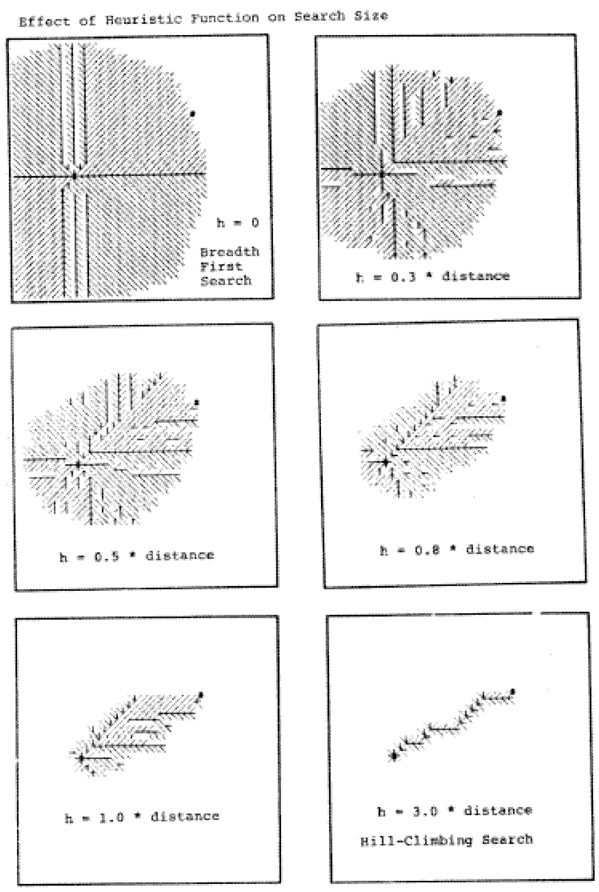
- $f(n) = h(n) + g(n)$: Search directed toward goal



- $h = h*$: Search proceeds directly to goal



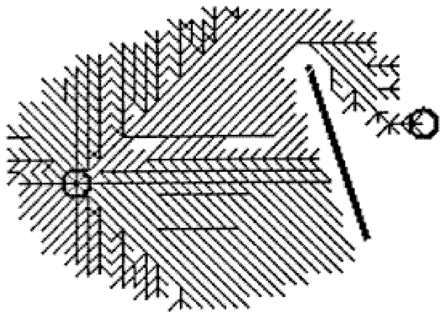
Effect of Heuristic Function



Heuristic Search Handles Local Maxima

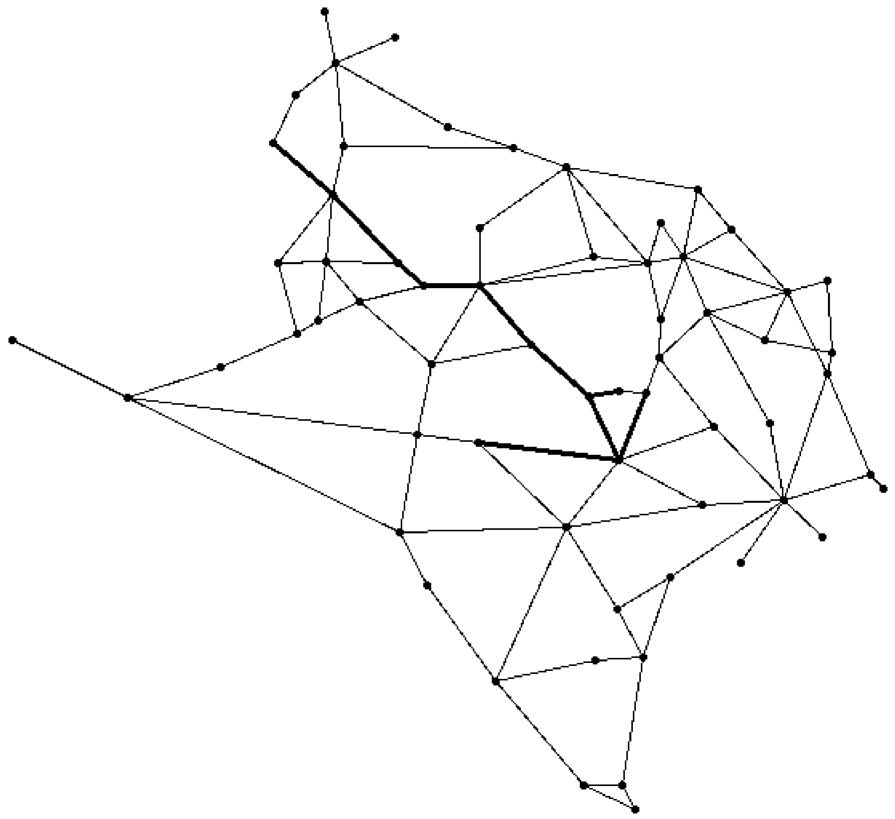
A barrier in the route-finding space creates a local maximum where hill-climbing would get stuck. Heuristic search will widen the search until it gets around the barrier.

$h = 0.9 * distance$, with barrier.



A* Algorithm Example

A* finds the shortest path to a single goal node from a given start. A* will do less work than Dijkstra because it focuses its search on the goal using the heuristic.



Mapping

A *mapping* $M : D \rightarrow R$ specifies a correspondence between elements of a *domain* D and a *range* R.

If each element of D maps to a single element of R, the mapping is *one-to-one* or *injective* .

If every element of R is mapped to by some element of D, the mapping is *onto* or *surjective* .

A mapping that is both one-to-one and onto is *bijective.*

Implementation of Mapping

We have seen several ways in which a mapping can be implemented:

- A function such as `sqrt` maps from its argument to the target value.
- If the domain is a finite, compact set of integers, we can store the target values in an array and look them up quickly.
- If the domain is a finite set, we can use a lookup table such as an association list, `TreeMap` or `HashMap`.
- If the domain is a finite set represented as an array or linked list, we can create a corresponding array or list of target values.

Functional Programming

A *functional* program is one in which:

- all operations are performed by functions
- a function does not modify its arguments or have *side-effects* (such as printing, setting the value of a global variable, writing to disk).

A subset of Lisp, with no destructive functions, is an example of a functional language.

```
(defun hypotenuse (x y)
  (sqrt (+ (expt x 2)
           (expt y 2))) )
```

Functional programming is easily adapted to parallel programming, since the program can be modeled as flow of data through functions that could be on different machines.

Associative and Commutative

An operation $\circ$ is *associative* if $a \circ (b \circ c) = (a \circ b) \circ c$.

An operation $\circ$ is *commutative* if $a \circ b = b \circ a$.

If an operation $\circ$ is *both* associative and commutative, then the arguments of the operation can be in any order, and the result will be the same. For example, the arguments of integer + can be in any order.

This gives great freedom to process the arguments of a function independently on multiple processors.

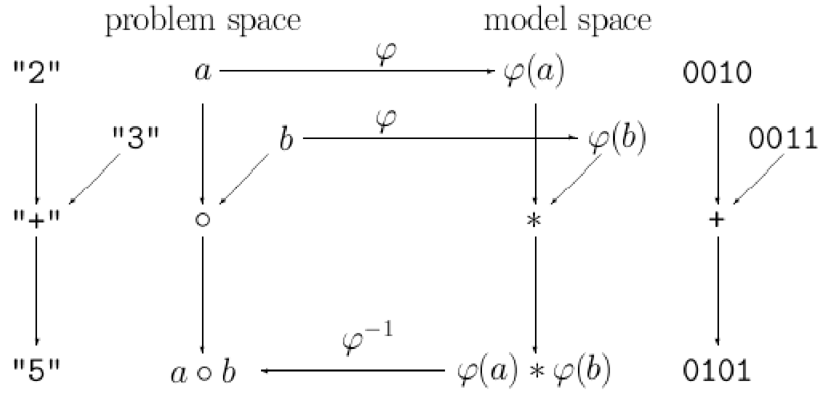
In many cases, parts of the operation (e.g. partial sums) can be done independently as well.

Computation as Simulation

It is useful to view computation as simulation, *cf.*: *isomorphism of semigroups.*[8]

> Given two semigroups $G_1 = [S, \circ]$ and $G_2 = [T, *]$, an invertible function $\varphi : S \to T$ is said to be an *isomorphism* between G_1 and G_2 if, for every a and b in S, $\quad \varphi(a \circ b) = \varphi(a) * \varphi(b)$

from which: $\quad a \circ b = \varphi^{-1}(\varphi(a) * \varphi(b))$



```
(princ-to-string (+ (read-from-string "2")
                    (read-from-string "3")))
"5"
```

[8]Preparata, F. P. and Yeh, R. T., *Introduction to Discrete Structures*, Addison-Wesley, 1973, p. 129.

Mapping in Lisp

Lisp has several functions that compute mappings from a linked list. The one we have seen is `mapcar`, which makes a new list whose elements are obtained by applying a specified function to each element (`car` or `first`) of the input list(s).

```
>(defun square (x) (* x x))

>(mapcar 'square '(1 2 3 17))

(1 4 9 289)

>(mapcar '+ '(1 2 3 17) '(2 4 6 8))

(3 6 9 25)

>(mapcar '> '(1 2 3 17) '(2 4 6 8))

(NIL NIL NIL T)
```

Mapcan

The Lisp function **mapcan** works much like **mapcar**, but with a different way of gathering results:

- The function called by **mapcan** returns a *list* of results (perhaps an empty list).
- **mapcan** concatenates the results; empty lists vanish.

```
(defun filter (lst predicate)
  (mapcan #'(lambda (item)
              (if (funcall predicate item)
                  (list item)
                  '()))
          lst) )

>(filter '(a 2 or 3 and 7) 'numberp)

(2 3 7)

>(filter '(a 2 or 3 and 7) 'symbolp)

(A OR AND)
```

Input Filtering and Mapping

We can use `mapcan` to both filter input and map input values to intermediate values for the application.

- **filter:** get rid of uninteresting parts of the input.
- **map:** convert an interesting part of the input to a useful intermediate value.

A key point is that we are not trying to compute the final answer, but to set up the inputs for another function to compute the final answer.

Suppose that we want to count the number of `z`'s in an input list. We could map a `z` to a `1`, which must be `(1)` for `mapcan`; anything else will map to `()` or `nil`.

```
(defun testforz (item)
  (if (eq item 'z)    ; if it is a z
      (list 1)        ;   emit 1    (map)
      '()) )          ;   else emit nothing
                      ;             (filter)

>(mapcan 'testforz '(z m u l e z r u l e z))

(1 1 1)
```

Reduce in Lisp

The function `reduce` applies a specified function to the first two elements of a list, then to the result of the first two and the third element, and so forth.

```
>(reduce '+ '(1 2 3 17))

23

>(reduce '* '(1 2 3 17))

102
```

`reduce` is just what we need to process a result from `mapcan`:

```
>(reduce '+ (mapcan 'testforz
                    '(z m u l e z r u l e z)))

3
```

Combining Map and Reduce

A combination of `map` and `reduce` can provide a great deal of power in a compact form.

The *Euclidean distance* between two points in n-space is the square root of the sum of squares of the differences between the points in each dimension.

Using `map` and `reduce`, we can define Euclidean distance compactly for any number of dimensions:

```
(defun edist (pointa pointb)
  (sqrt (reduce '+
          (mapcar 'square
                  (mapcar '- pointa pointb)))) )

>(edist '(3) '(1))

2.0

>(edist '(3 3) '(1 1))

2.8284271247461903

>(edist '(3 4 5) '(2 4 8))

3.1622776601683795
```

MapReduce and Massive Data

At the current state of technology, it has become difficult to make individual computer CPU's faster; however, it has become cheap to make lots of CPU's. Networks allow fast communication between large numbers of cheap CPU's, each of which has substantial main memory and disk.

A significant challenge of modern CS is to perform large computations using networks of cheap computers operating in parallel.

Google specializes in processing massive amounts of data, particularly the billions of web pages now on the Internet. `MapReduce` makes it easy to write powerful programs over large data; these programs are mapped onto Google's network of hundreds of thousands of CPU's for execution.

Distributed Programming is Hard!

- 1000's of processors require 1000's of programs.
- Need to keep processors busy.
- Processors must be *synchronized* so they do not interfere with each other.
- Need to avoid bottlenecks (most of the processors waiting for service from one processor).
- Some machines may become:
  - slow
  - dead
  - evil

  and they may change into these states while your application is running.
- If a machine does not have the data it needs, it must get the data via the network.
- Many machines share one (slow) network.
- Parts of the network can fail too.

What MapReduce Does for Us

`MapReduce` makes it easy to write powerful programs over large data to be run on thousands of machines.

All the application programmer has to do is to write two small programs:

- Map: Input → intermediate value
- Reduce: list of intermediate values → answer

These two programs are small and easy to write!

MapReduce does all the hard stuff for us.

Map Sort Reduce

`MapReduce` extends the Lisp `map` and `reduce` in one significant respect: the `map` function produces not just one result, but a set of results, each of which has a *key* string.

When our function `testforz` found a `z`, it would output `(1)`. But now, we will always produce a key as well, e.g. `(z (1))`. In Java, we would say:

```
   mr.collect_map("z", list("1"));
```

because the intermediate values are always strings.

There is an intermediate *Sort* process that groups the results for each key. Then `reduce` is applied to the results for each key, returning the key with the reduced answer for that key.

Simplified MapReduce

We think of the `map` function as taking a single input, typically a `String`, and *emitting* zero or more outputs, each of which is a (*key*, (*value*)) pair. For example, if our program is counting occurrences of the word *liberty*, the input `"Give me liberty"` would emit one output, `("liberty", ("1"))`.

As an example, consider the problem of finding the nutritional content of a cheeseburger. Each component has a variety of features such as calories, protein, etc. MapReduce can add up the features individually.

We will present a simple version of MapReduce in Lisp to introduce how it works.

Mapreduce in Lisp

```
(defun mapreduce (mapfn reducefn lst)
  (let (db keylist)
    (dolist (item lst)
      (dolist (resitem (funcall mapfn item))
        (or (setq keylist
                  (assoc (first resitem) db
                         :test 'equal))
            (push (setq keylist
                        (list (first resitem)))
                  db))
        (push (second resitem) (rest keylist)) ) )
    (mapcar #'(lambda (keylist)
                (list (first keylist)
                      (reduce reducefn
                              (rest keylist))))
            db) ))

>(mapreduce 'identity '+ '(((a 3) (b 2) (c 1))
                           ((b 7) (d 3) (c 5))))

((D 3) (C 6) (B 9) (A 3))
```

Simple MapReduce Example

```
>(mapreduce 'identity '+
            '(((a 3) (b 2) (c 1))
              ((b 7) (d 3) (c 5))) t)
Mapping: ((A 3) (B 2) (C 1))
   Emitted: (A 3)
   Emitted: (B 2)
   Emitted: (C 1)
Mapping: ((B 7) (D 3) (C 5))
   Emitted: (B 7)
   Emitted: (D 3)
   Emitted: (C 5)
Reducing: D (3)  = 3
Reducing: C (5 1)  = 6
Reducing: B (7 2)  = 9
Reducing: A (3)  = 3

((D 3) (C 6) (B 9) (A 3))
```

MapReduce Example

```
(defun nutrition (food)
  (rest (assoc food
    '((hamburger (calories 80) (fat 8)
                 (protein 20))
      (bun (calories 200) (carbs 40) (protein 8)
           (fiber 4))
      (cheese (calories 100) (fat 15) (sodium 150))
      (lettuce (calories 10) (fiber 2))
      (tomato (calories 20) (fiber 2))
      (mayo (calories 40) (fat 5) (sodium 20)) ) ))

>(nutrition 'bun)

((CALORIES 200) (CARBS 40) (PROTEIN 8) (FIBER 4))

>(mapreduce 'nutrition '+ '(hamburger bun cheese
                            lettuce tomato mayo))

((SODIUM 170) (FIBER 8) (CARBS 40) (PROTEIN 28)
 (FAT 28) (CALORIES 450))
```

Hamburger Example

```
>(mapreduce 'nutrition '+
            '(hamburger bun cheese lettuce tomato mayo) t)
Mapping: HAMBURGER
   Emitted: (CALORIES 80)
   Emitted: (FAT 8)
   Emitted: (PROTEIN 20)
Mapping: BUN
   Emitted: (CALORIES 200)
   Emitted: (CARBS 40)
   Emitted: (PROTEIN 8)
   Emitted: (FIBER 4)
Mapping: CHEESE
   Emitted: (CALORIES 100)
   Emitted: (FAT 15)
   Emitted: (SODIUM 150)
Mapping: LETTUCE
   Emitted: (CALORIES 10)
   Emitted: (FIBER 2)
Mapping: TOMATO
   Emitted: (CALORIES 20)
   Emitted: (FIBER 2)
Mapping: MAYO
   Emitted: (CALORIES 40)
   Emitted: (FAT 5)
   Emitted: (SODIUM 20)
Reducing: SODIUM (20 150)  = 170
Reducing: FIBER (2 2 4)  = 8
Reducing: CARBS (40)  = 40
Reducing: PROTEIN (8 20)  = 28
Reducing: FAT (5 15 8)  = 28
Reducing: CALORIES (40 20 10 100 200 80)  = 450

((SODIUM 170) (FIBER 8) (CARBS 40) (PROTEIN 28) (FAT 28)
 (CALORIES 450))
```

PageRank

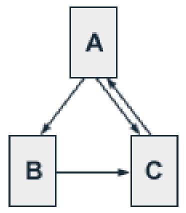
The *PageRank* algorithm used by Google expresses the ranking of a web page in terms of two components:

- a base value, $(1 - d)$, usually 0.15
- $d * \Sigma_{i \in links} PR_i / n_i$ where PR_i is the page rank of a page that links to this page, and n_i is the number of links from that page.

The PageRank values can be approximated by *relaxation* by using this formula repeatedly within MapReduce. Each page is initially given a PageRank of 1.0; the sum of all values will always equal the number of pages.

- **Map:** Share the love: each page distributes its PageRank equally across the pages it links to.
- **Reduce:** Each page sums the incoming values, multiplies by 0.85, and adds 0.15 .

PageRank Example



Iterative PageRank converges fairly quickly for this net:[9]

```
     A              B              C
1.00000000     1.00000000     1.00000000
1.00000000     0.57500000     1.42500000
1.36125000     0.57500000     1.06375000
1.05418750     0.72853125     1.21728125
1.18468906     0.59802969     1.21728125
1.18468906     0.65349285     1.16181809
1.13754537     0.65349285     1.20896178
1.17761751     0.63345678     1.18892571
1.16058685     0.65048744     1.18892571
1.16058685     0.64324941     1.19616374
1.16673918     0.64324941     1.19001141
1.16150970     0.64586415     1.19262615
1.16373223     0.64364162     1.19262615
  ...
1.16336914     0.64443188     1.19219898
```

The sum of PageRank values is the total number of pages. The value for each page is the expected number of times a random web surfer, who starts as many times as there are web pages, would land on that page.

[9]`http://pr.efactory.de/e-pagerank-algorithm.shtml`

Running PageRank Example

```
Starting MapReduce on:
  ((a (1.0 (b c))) (b (1.0 (c))) (c (1.0 (a))))
  mapping:  key = a  val = (1.0 (b c))
    emitting:  key = b  val = (0.5)
    emitting:  key = c  val = (0.5)
    emitting:  key = a  val = ((b c))
  mapping:  key = b  val = (1.0 (c))
    emitting:  key = c  val = (1.0)
    emitting:  key = b  val = ((c)) ...
  reducing:  key = a  val = (((b c)) (1.0))
    result:  key = a  val = (1.0 (b c))
  reducing:  key = b  val = ((0.5) ((c)))
    result:  key = b  val = (0.575 (c))
  reducing:  key = c  val = ((0.5) (1.0) ((a)))
    result:  key = c  val = (1.425 (a))

Starting MapReduce on:
  ((a (1.0 (b c))) (b (0.575 (c))) (c (1.425 (a))))
  reducing:  key = a  val = (((b c)) (1.425))
  reducing:  key = b  val = ((0.5) ((c)))
  reducing:  key = c  val = ((0.5) (0.575) ((a)))


Starting MapReduce on:
  ((a (1.36125 (b c))) (b (0.575 (c))) (c (1.06375 (a))))
  reducing:  key = a  val = (((b c)) (1.06375))
  reducing:  key = b  val = ((0.680625) ((c)))
  reducing:  key = c  val = ((0.680625) (0.575) ((a)))

  ... after 10 steps:

Result = ((a (1.16673918 (b c)))
          (b (0.64324941 (c)))
          (c (1.19001141 (a))))
```

Advanced Performance

The notions of Big O and single-algorithm performance on a single CPU must be extended in order to understand performance of programs on more complex computer architectures. We need to also account for:

- Disk access time
- Network bandwidth and data communication time
- Coordination of processes on separate machines
- Congestion and bottlenecks as many computers or many users want the same resource.

Performance Techniques in MapReduce

- The Google File System (GFS) stores multiple copies (typically 3) of data files on different computers for redundancy and availability.
- Master assigns workers to process data such that the data is on the worker's disk, or near the worker within the same rack. This reduces network communication; network bandwidth is scarce.
- Combiner functions can perform partial reductions (adding `"1"` values) before data are written out to disk, reducing both I/O and network traffic.
- Master can start redundant workers to process the same data as a dead or ”slacker” worker; whichever worker finishes first wins.
- Reduce workers can start work as soon as some Map workers have finished their data.

Buffering

Buffering is a technique used to match a small-but-steady process (e.g. a program that reads or writes one line at a time) to a large-block process (e.g. disk I/O).

Disk I/O has two problematic features:

- A whole disk block (e.g. 4096 bytes) must be read or written at a time.
- Disk access is slow (e.g. 8 milliseconds).

An *I/O buffer* is an array, the same size as a disk block, that is used to collect data. The application program removes data from the block (or adds data to it) until the block is empty (full), at which time a new block is read from disk (written to disk).

If there are R Reduce tasks, each Map task will have R output buffers, one for each Reduce task. When an output buffer becomes full, it is written to disk. When the Map task is finished, it sends the file names of its R files to the Master.

Load Balancing

Some data values are much more popular than others. For example, there are 13 people on the class roster whose names start with **S**, but only one **K**, and no **Q** or **X**.

If MapReduce assigned Reduce tasks based on `key` values, some Reduce tasks might have large inputs and be too slow, while other Reduce tasks might have too little work.

MapReduce performs *load balancing* by having a large number R of Reduce tasks and using hashing to assign data to Reduce tasks:

$$task \;=\; Hash(key)\; mod\; R$$

This assigns many keys to the same Reduce task. The Reduce task reads the files produced by all Map tasks for its hash value (remote read over the network), sorts the combined input by `key` value, and appends the `value` lists before calling the application's Reduce function.

Algorithm Failure

If MapReduce detects that a worker has failed or is slow on a Map task, it will restart redundant Map tasks to process the same data.

If the redundant Map tasks also fail, maybe the problem is that the data causes the algorithm to fail, rather than hardware failure.

MapReduce can restart the Map task without the last unprocessed data. This causes the output to be not quite right, but for some tasks (e.g. average movie rating) it could be acceptable.

Atomic Commit

If multiple worker machines are working on the same data, it is necessary to ensure that only one set of result data is actually used.

An *atomic commit* is provided by the operating system (and, ultimately, CPU hardware) that allows exactly one result to be committed or accepted for use. If other workers produce the same result, those results will be discarded.

In MapReduce, atomicity is provided by the file system. When a Map worker finishes, it renames its temporary file to the final name; if a file by that name already exists, the renaming will fail.