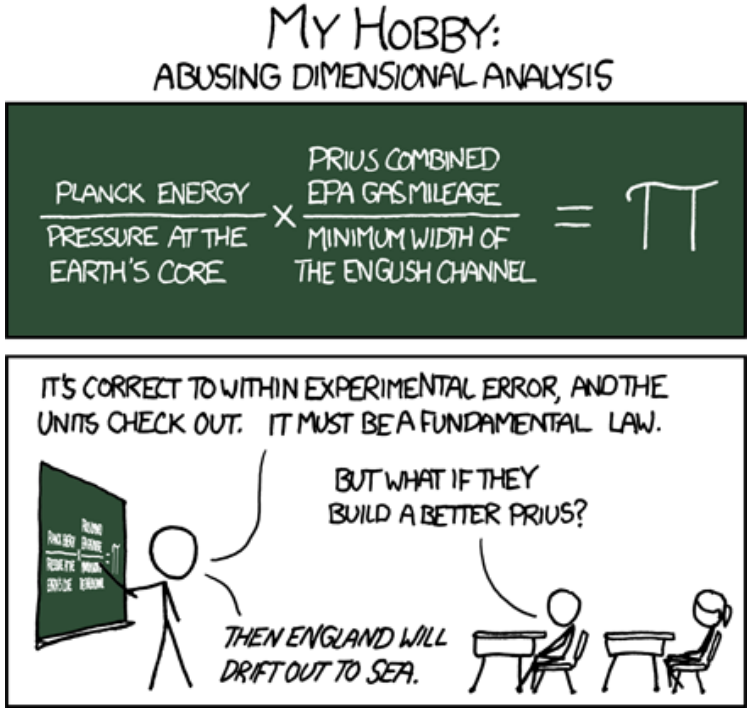
Graphs: Trees' Tangled Cousins

September 13, 2011

CS 42: Principles and Practice of Computer Science



The Game of Twenty Questions

- ✓ Extra credit in Assignment 3
- ✓ What's the tree?

The Game of Twenty Questions

- ✓ Extra credit in Assignment 3
- ✓ What's the tree?

Is it bigger than a breadbox?

no

yes

spam

Robin Burgener
gelconference.com/07/robin.html

a computer scientist

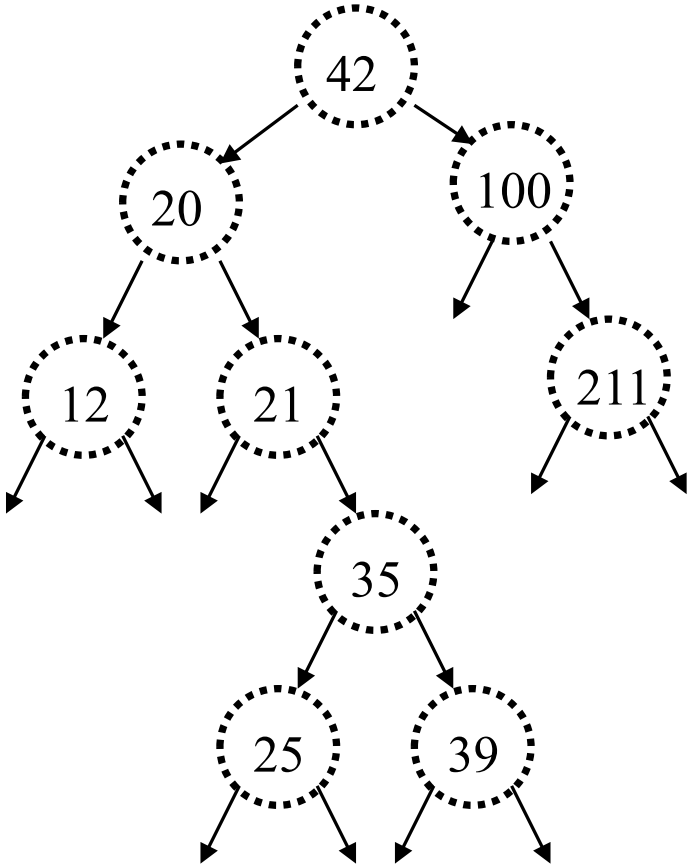
Side-Note: Big-O

Describes the scalability of an algorithm; an upper bound on number of steps for an input of size n, as n goes to infinity.

E.g., $O(n^2)$ *means "eventually bounded above by a multiple of* n^2*"*

- ✓ If an algorithm takes $n - 1$ steps for an input of size n, we can say it is $O(n)$.
- ✓ If an algorithm takes $3n^2 + 2n + 7$ steps for an input of size n, we can say it is $O(n^2)$.
- ✓ If an algorithm takes $3 \log_{12} n$ steps for an input of size n, we can say it is $O(???)$.
- ✓ If an algorithm takes $|10 \sin(n)|$ steps for an input of size n, we can say it is $O(?)$.

Not so fast?



N *nodes*

	Nice *case?*	Worst *case?*
`find`	`find`	`find`
`insert`	`insert`	`insert`
`delete`	`delete`	`delete`

Lopsided Trees

Suppose elements are being inserted into a binary tree.

Lopsided Trees

Suppose elements are being inserted into a binary tree.

At each decision point *99% of elements* are larger and sort to the right.

Lopsided Trees

Suppose elements are being inserted into a binary tree.

At each decision point *99% of elements* are larger and sort to the right.

How tall is such an N-node tree?

Trees are fast!

For any *constant* left/right split ratio of the data, `find`, `insert`, and `delete` are $O(\log N)$.

But what if the data conspires against you (*e.g.*, a sorted list)?

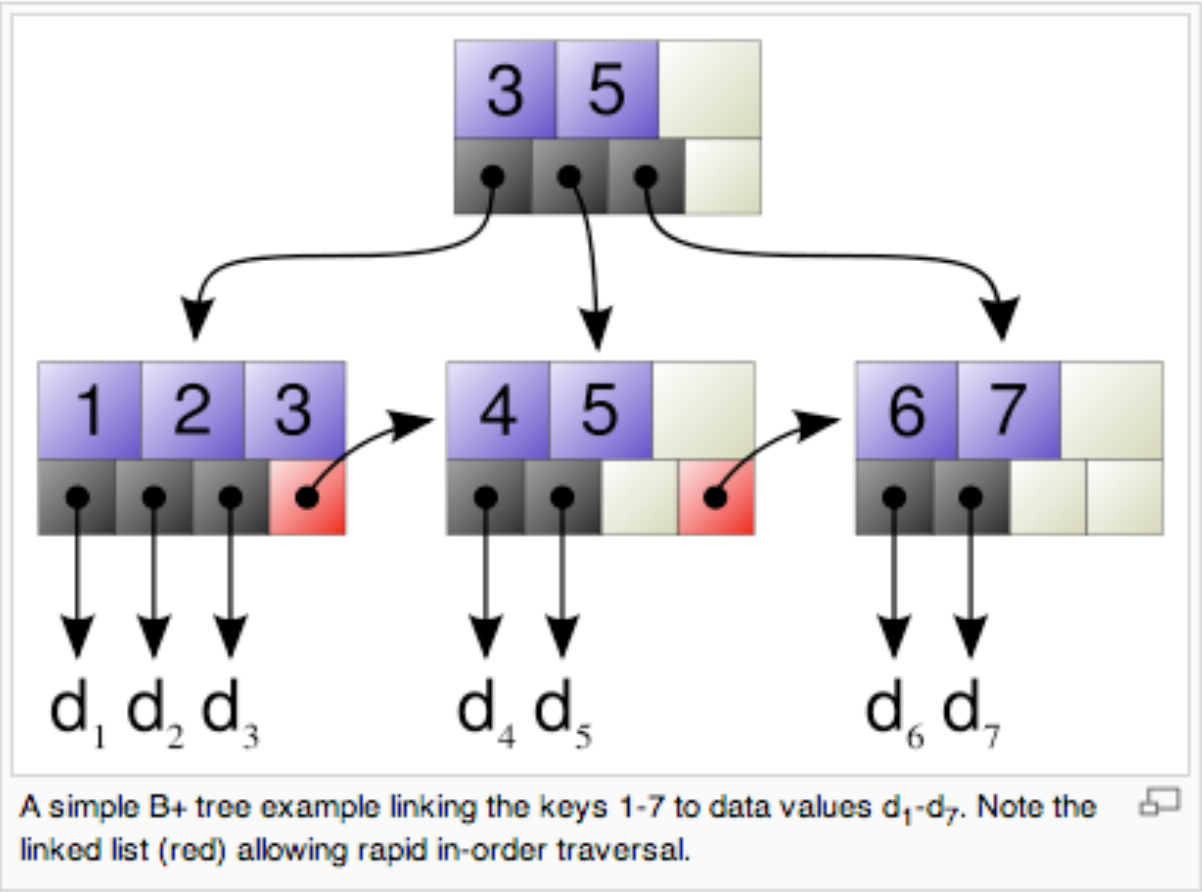
Trees are fast!

For any *constant* left/right split ratio of the data, `find`, `insert`, and `delete` are $O(\log N)$.

But what if the data conspires against you (*e.g.*, a sorted list)?

- ✓ Self-balancing binary trees (covered in CS 70)
- ✓ E.g., `http://www.cs.hmc.edu/~stone/Applets/avltree.html`

B-Trees (actually B+ Trees)



A simple B+ tree example linking the keys 1-7 to data values d_1-d_7. Note the linked list (red) allowing rapid in-order traversal.

wikipedia, B+ trees

The ReiserFS filesystem (for Unix and Linux), XFS filesystem (for IRIX and Linux), JFS2 filesystem (for AIX, OS/2 and Linux), and NTFS all use this type of tree for block indexing. Relational databases such as PostgreSQL and MySQL also often use this type of tree for table indices.

File systems and databases tend to use trees with higher branching factors. (Why?)

The Mac's File System

CHAPTER 12

The HFS Plus File System

Looking Back

Apple filed a patent for the Macintosh Hierarchical File System (U.S. Patent Number 4,945,475) in late 1989. The patent was granted in mid-1990. The original HFS was implemented using two B-Tree data structures: the Catalog B-Tree and the Extents B-Tree. As we will see in this chapter, HFS+ uses both of these B-Trees. Lisa OS—the operating system for Apple's Lisa computer (1983)—used a hierarchical file system before the Macintosh. Indeed, the HFS volume format benefited from work done on the Lisa's file system.

As we noted in Chapter 11, one hallmark of HFS was that it lent support to the graphical user interface by providing a separate data stream in a file—the *resource fork*—for storing application icons, resources, and other auxiliary data independently of the file's "main" data.

The HFS Plus file system (or simply HFS+) is the preferred and default volume format on Mac OS X. The term *HFS* stands for *Hierarchical File System*, which replaced the flat Macintosh File System (MFS) used in early Macintosh operating systems. HFS remained the primary volume format for Macintosh systems before Mac OS 8.1, which was the first Apple operating system to support HFS+. Also called the *Mac OS X Extended* volume format, HFS+ is architecturally similar to HFS but provides several important benefits over the latter.[1] Moreover, HFS+ itself has evolved greatly since its inception—not so much in fundamental architecture but in its implementation. In this chapter, we will discuss features and implementation details of HFS+ in Mac OS X.

Noteworthy features of HFS+ include the following:

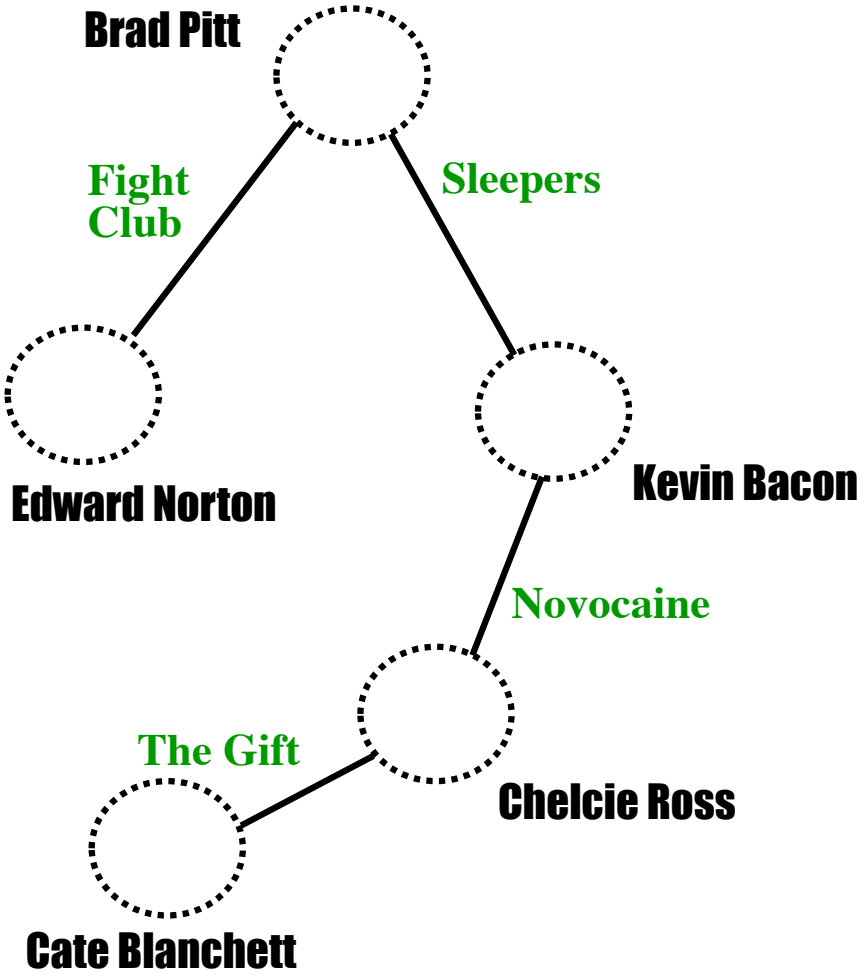
- Support for files up to 2^{63} bytes in size
- Unicode-based file/directory name encoding, with support for names containing up to 255 16-bit Unicode characters[2]
- A B+ Tree (the *Catalog B-Tree*) for storing the file system's hierarchical structure, allowing tree-based indexing
- Extent-based allocation of storage space using 32-bit allocation block numbers, with delayed allocation of physical blocks
- A B+ Tree (the *Extents Overflow B-Tree*) for recording files' "overflow" extents (the ninth and subsequent—for files with more than eight extents)
- Multiple byte-streams (or forks) per file, with two predefined forks and an arbitrary number of other, named forks that are stored in a separate B-Tree (see next item).
- A B+ Tree (the *Attributes B-Tree*) for storing arbitrary metadata[3] per file, thus providing native support for extended file system attributes (the names of which are Unicode strings up to 128 16-bit Unicode characters in length)
- Metadata journaling through the kernel's VFS-level journaling mechanism
- Multiple mechanisms to allow one file system object to refer to another: aliases, hard links, and symbolic links

1. Two of the major limitations in HFS were that it was largely single threaded and that it supported only 16-bit allocation blocks.

2. HFS+ stores Unicode characters in canonical, fully decomposed form.

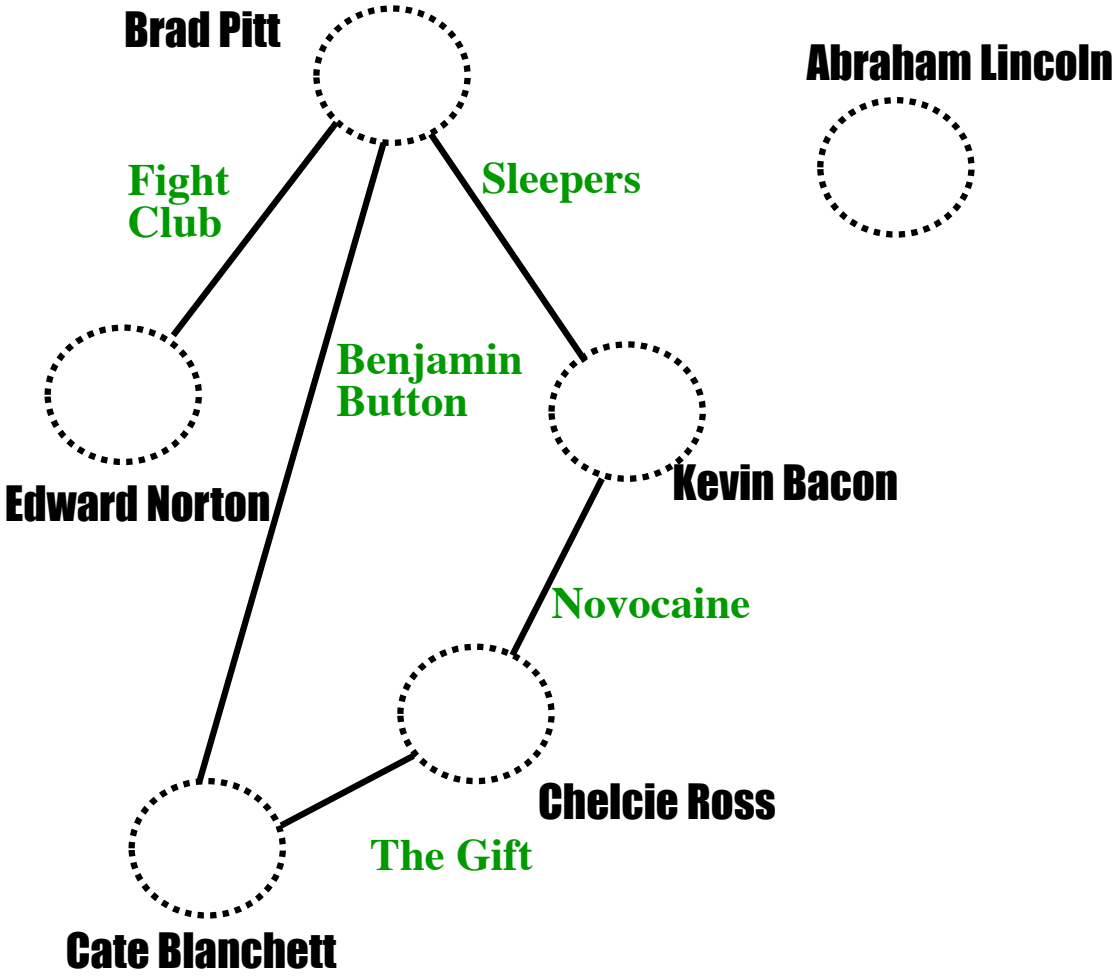
3. The size of the data associated with a single extended attribute is limited to slightly less than 4KB in Mac OS X 10.4.

Another Tree?



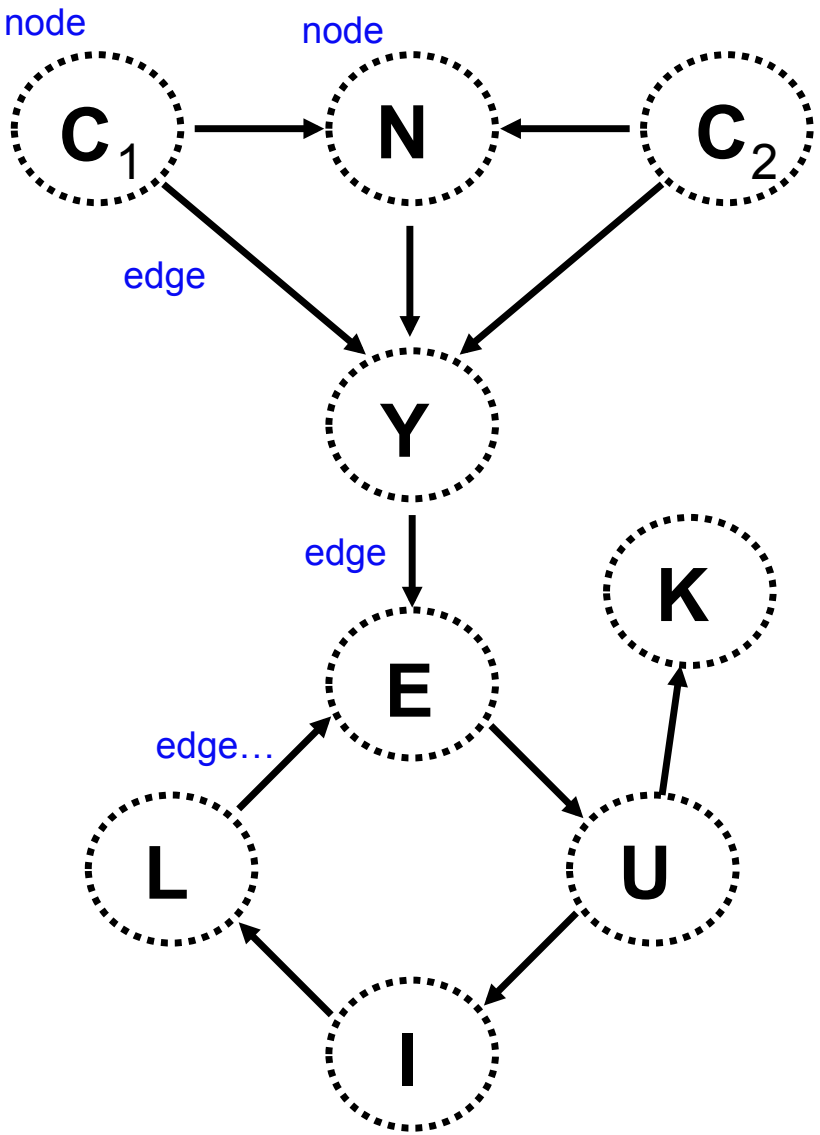
The "Oracle of Bacon" at **oracleofbacon.org/**

Undirected Graphs



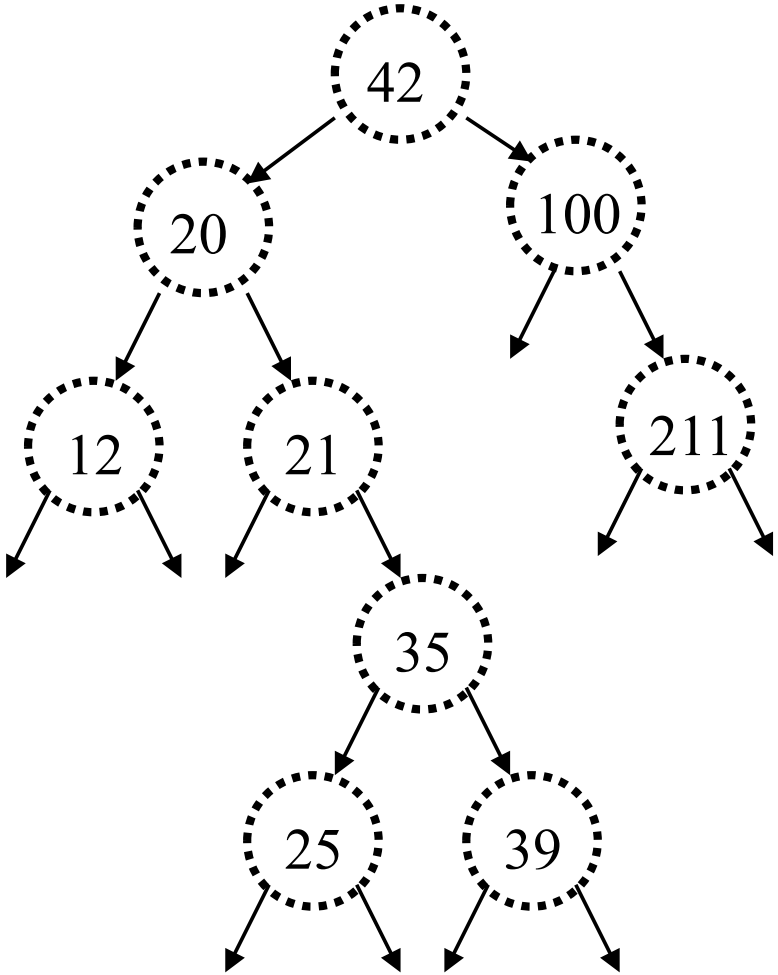
The "Oracle of Bacon" at **oracleofbacon.org/**

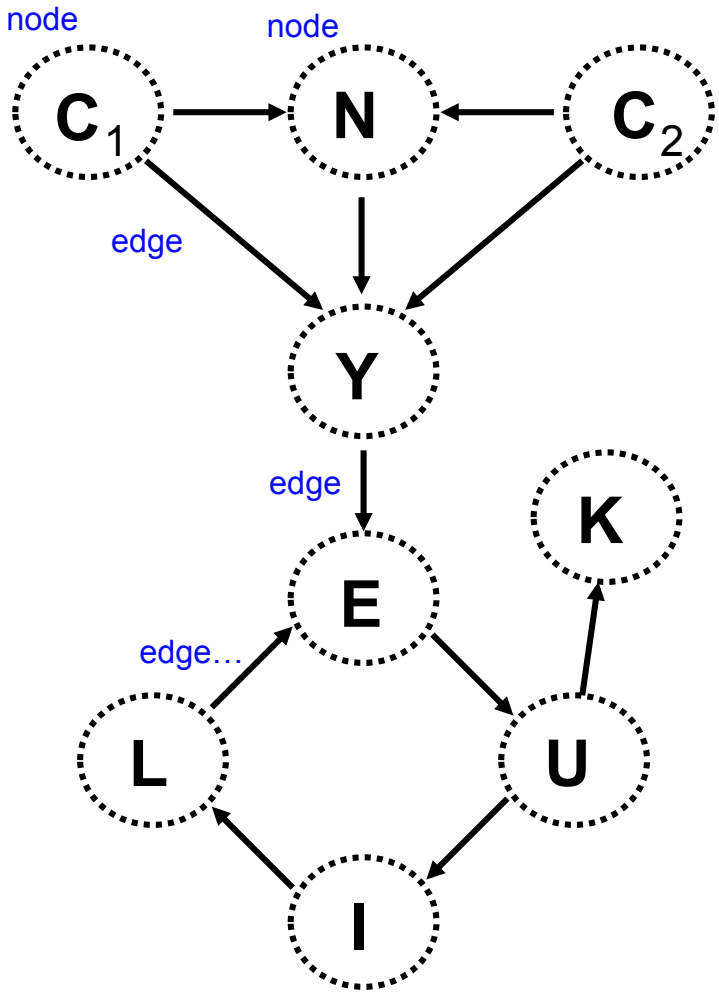
Directed Graphs



Trees vs. Directed Graphs

How are they different?



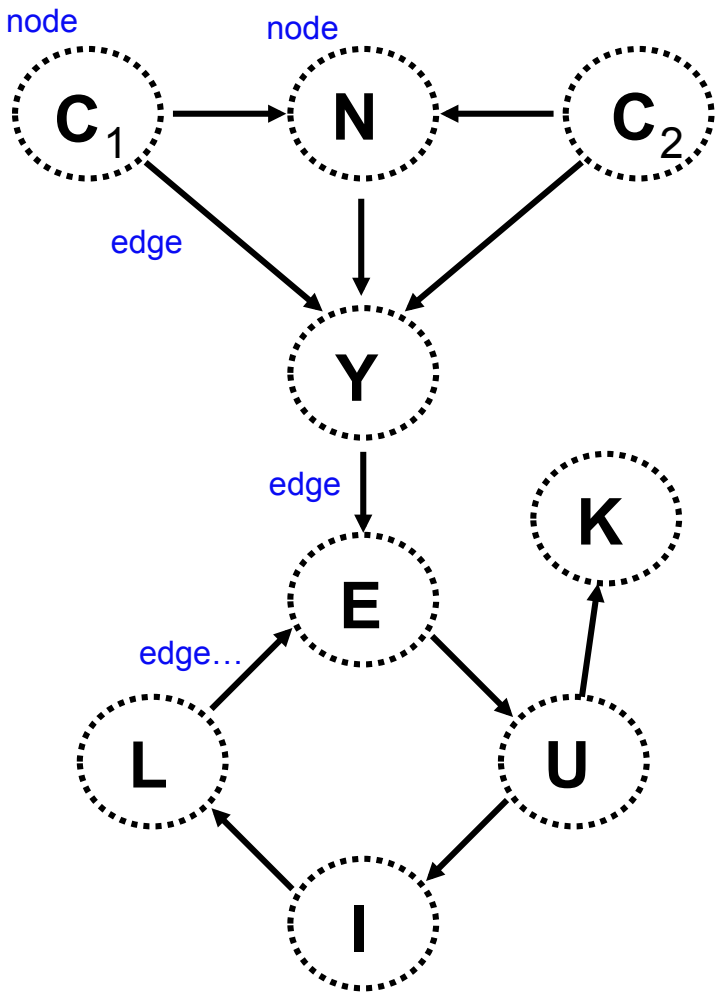


Graphical Programming?

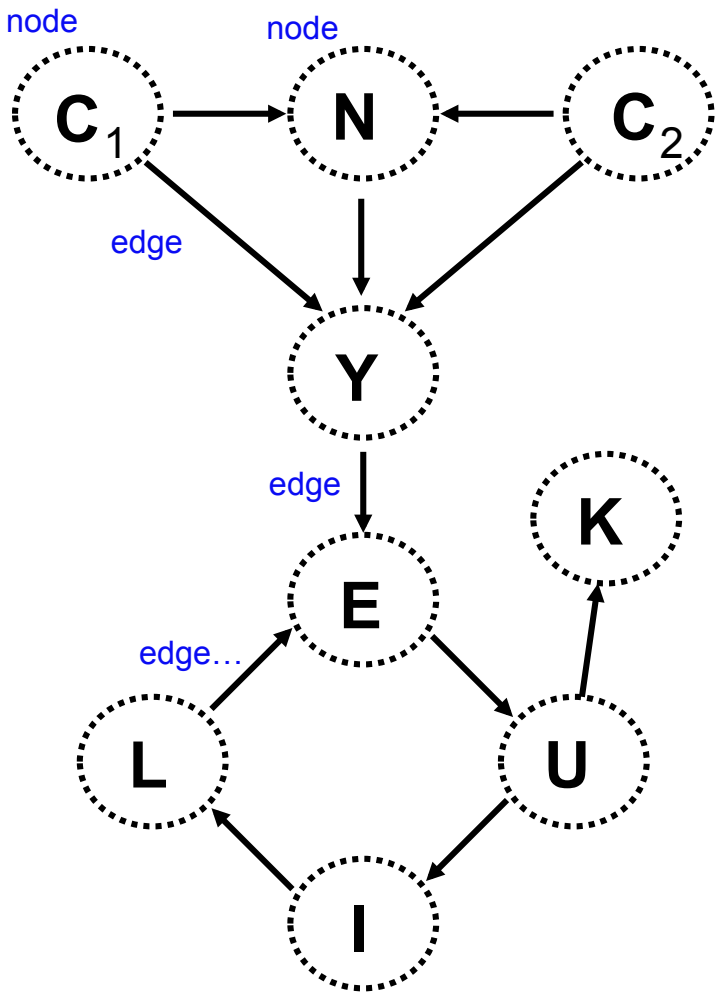
Lists are our only building block in Racket.

- ✓ Unicalc Quantities: `'(14.0 (day) ())`
- ✓ BSTs: `'(42 (2 () ()) (50 (45 () ()) ()))`
- ✓ Graphs: ???

Graph's Anatomy: Edge Lists

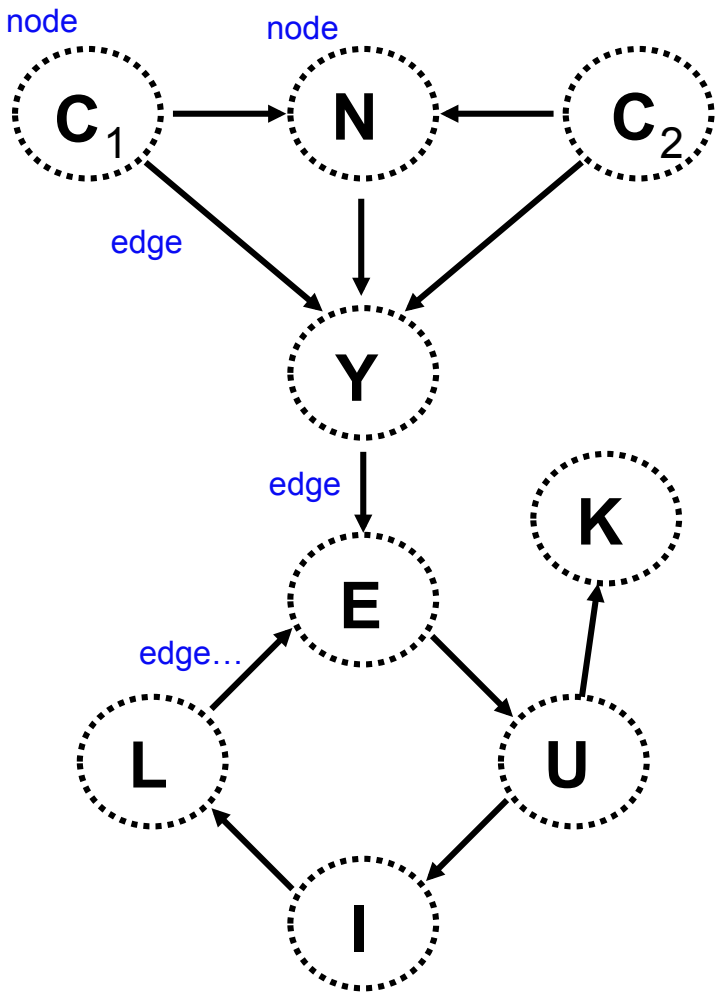


Graph's Anatomy: Edge Lists

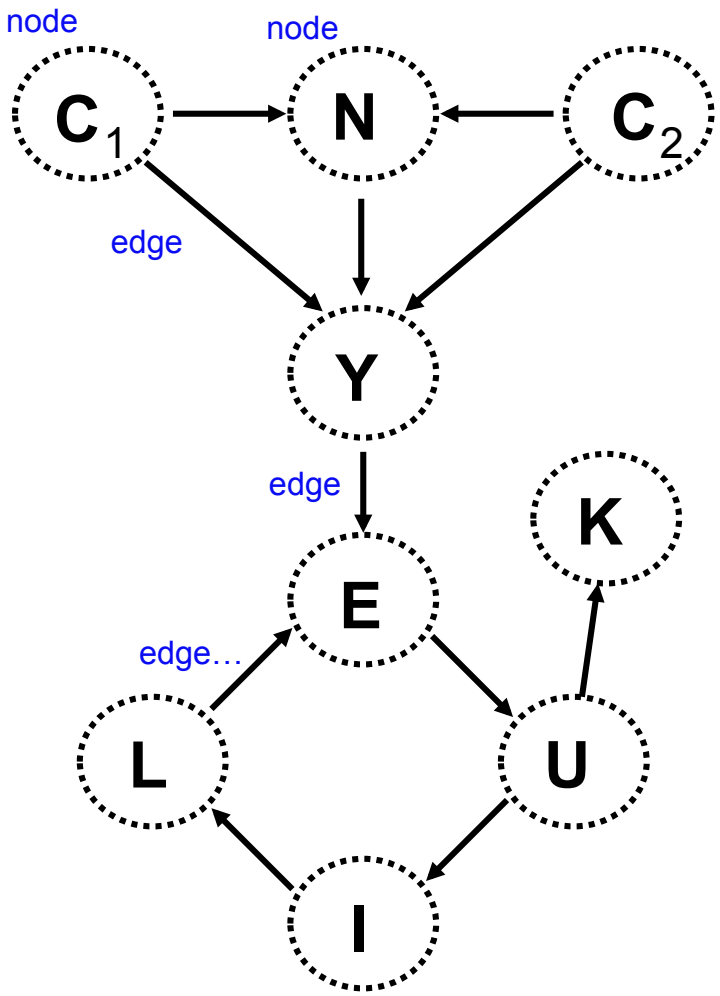



```
(define G '( (c1 n) (c1 y)
             (n  y) (c2 n)
             (c2 y) (y  e)
             (e  u) (u  i)
             (i  l) (l  e)
             (u k)
           )
)
```

Graph's Anatomy: Adjacency Lists

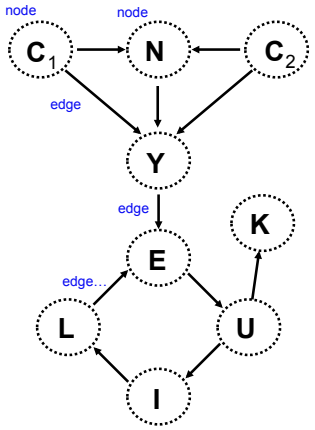


Graph's Anatomy: Adjacency Lists




```
(define G '( (c1 (n y))
             (n  (y))
             (c2 (n y))
             (k  ())
             (y  (e))
             (e  (u))
             (l  (e))
             (u  (i k))
             (i  (l))
           )
)
```

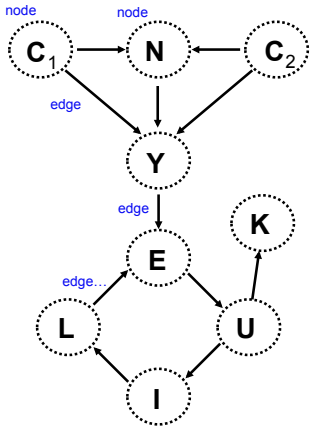
Find all the nodes in an edge list G




```
'( (c1 n) (c1 y) (n  y) (c2 n)
   (c2 y) (y  e) (e  u) (u  i)
   (i  l) (l  e) (u  k)       ))
```

```
(define (nodes G)
```

Find all the nodes in an edge list G



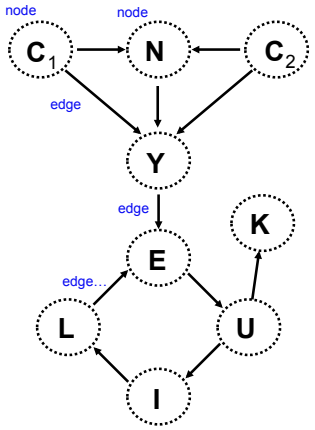

```
'( (c1 n) (c1 y) (n  y) (c2 n)
   (c2 y) (y  e) (e  u) (u  i)
   (i  l) (l  e) (u  k)       ))
```

```
(define (nodes G)

  (remove-dups (foldr append '() G)))

(define (remove-dups L)
  (cond
     [ (null? L) '() ]
     [ (member (first L) (rest L))  (remove-dups (rest L)) ]
     [ else  (cons (first L) (remove-dups (rest L))) ]))
```

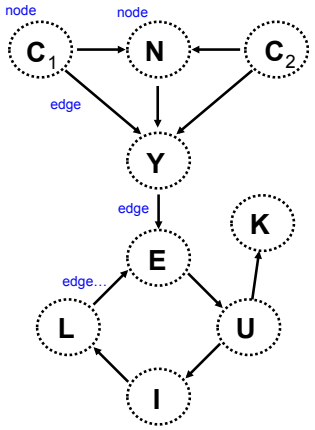
Find the chidren of a node n in edge list G




```
'( (c1 n) (c1 y) (n  y) (c2 n)
   (c2 y) (y  e) (e  u) (u  i)
   (i  l) (l  e) (u  k)       ))
```

```
(define (kids n G)
```

Find the chidren of a node n in edge list G

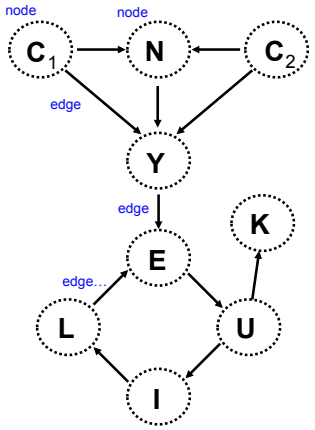



```
'( (c1 n) (c1 y) (n  y) (c2 n)
   (c2 y) (y  e) (e  u) (u  i)
   (i  l) (l  e) (u  k)       ))
```

```
(define (kids n G)

   (map second
        (filter (lambda (x) (equal? (first x) n)) G)))
```

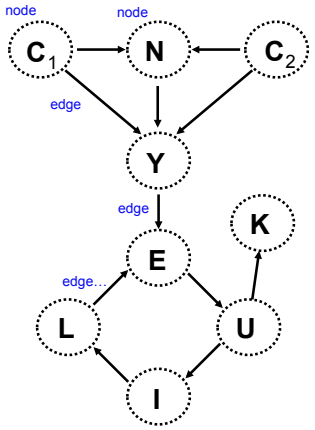
How about the parents?




```
'( (c1 n) (c1 y) (n  y) (c2 n)
   (c2 y) (y  e) (e  u) (u  i)
   (i  l) (l  e) (u  k)       ))
```

```
(define (parents n G)
```

How about the parents?



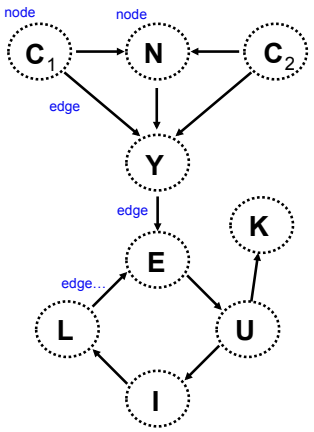

```
'( (c1 n) (c1 y) (n  y) (c2 n)
   (c2 y) (y  e) (e  u) (u  i)
   (i  l) (l  e) (u  k)       ))
```

```
(define (parents n G)

   (kids n (map reverse G))
```

Name:

"Quiz:" Functions on Graphs



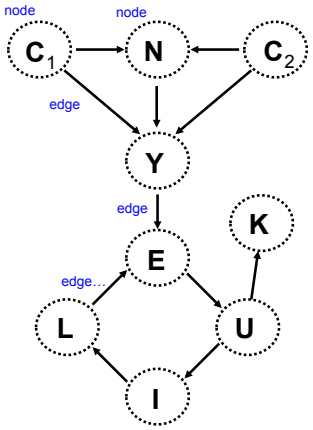

```
'( (c1 n) (c1 y)
   (n  y) (c2 n)
   (c2 y) (y  e)
   (e  u) (u  i)
   (i  l) (l  e)
   (u  k)       ))
```

Find the "roots" of a graph **G**.

```
;; Given a directed graph (edge list) G,
;; find the nodes with no incoming edge.
(define (roots G)
```

```
;; Given a directed graph (edge list) G,
;;  return #t if gp is a "grandparent"
;;  of gk, otherwise #f.
(define (gkid? gp gk G)
```

Reachability?




```
'( (c1 n) (c1 y) (n  y) (c2 n)
   (c2 y) (y  e) (e  u) (u  i)
   (i  l) (l  e) (u  k)      ))
```

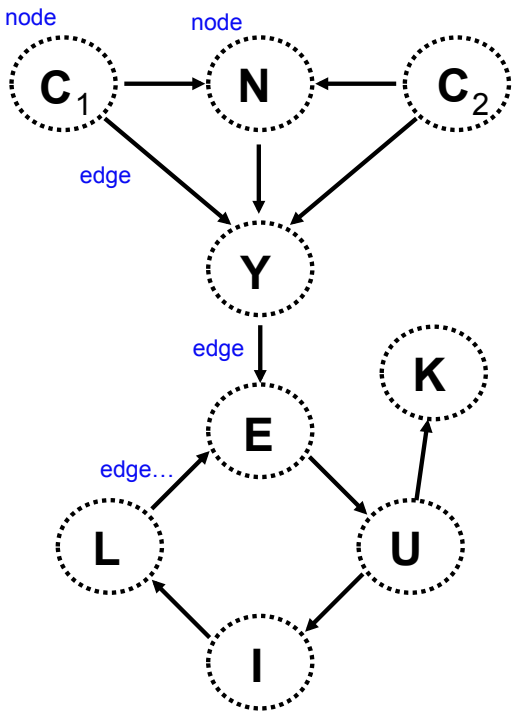
```
;; Can node b be reached by a directed path
;;   from node a?
(define (reach a b G)
   ;; What's our strategy?
```

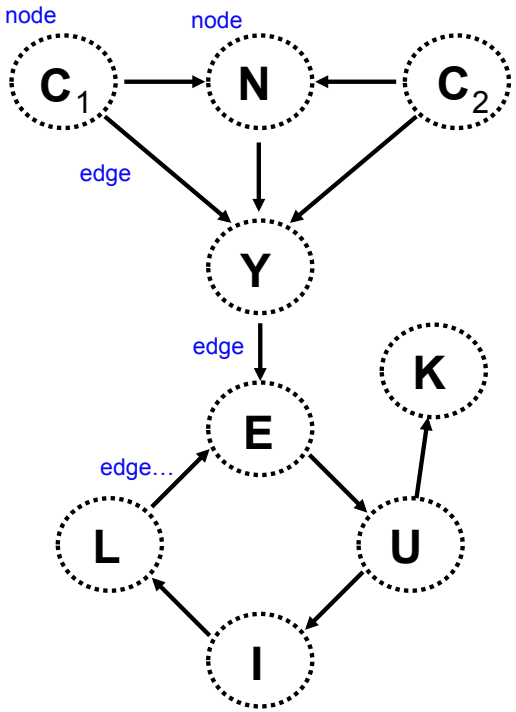
One approach

- ✓ Keep a collection of "partial" paths starting at **a**
- ✓ Repeatedly take out a path
  - If it contains the destination, done!
  - If not, find all ways to make it one step longer (without creating a cycle!), add these to our collection.

```
(define (reach a b G)
  (reach-help b (list (list a)) G))

(define (reach-help dst path-list G)
  (if (null? path-list)
      #f
      (let ((active-path (first path-list))
            (remaining-paths (rest path-list)))
        ; If active path contains the destination, return #t
        ; Otherwise, expand the active path by one node for
        ; each kid, ignoring any paths with cycles, creating
        ; a list of 0 or more new paths.
        ; append this new list of paths onto the remaining
        ; paths and recurse.
        ; At least one helper function (to build the list
        ; of new paths) is recommended.
        ...
```


node
node
C1
N
C2
edge
Y
edge
K
E
edge…
L
U
I




```
(define G '( (c1 n) (c1 y)
             (n  y) (c2 n)
             (c2 y) (y  e)
             (e  u) (u  i)
             (i  l) (l  e)
             (u k)
           )
)
```

```
(reach 'c1 'e G)
```

Weighted Graphs

Find the shortest path from Bucharest to Arad.