

Summary: computing list results

- Many programs require list results to be computed, built and returned
- Ways to build lists...
- (1) append intermediate answer to an overall result list

```
make_intlist(M, N, [ ]) :- M > N.
```

```
make_intlist(M, N, L) :-
```

```
  M =< N,
```

```
  M2 is M + 1,
```

```
  make_intlist(M2, N, L2),
```

```
  append([M], L2, L).
```

must pass L back to get final answer
in calling predicate

L2 is intermediate answer

Summary: building lists

- (2) prepend intermediate solution in predicate head

`make_intlist(M, N, []) :- M > N.`

`make_intlist(M, N, [M|L2]) :-`

`M =< N,`

`M2 is M + 1,`

`make_intlist(M2, N, L2).`

concatenate M to start of L2



L2 is intermediate answer



Summary: building lists

- (3) create a new argument with intermediate list; return it back when recursion has terminated

?- make_intlist(N, [], L).

make_intlist(N, L, L) :- N < 0.

make_intlist(N, L2, L) :-

N >= 0,

N2 is N - 1,

make_intlist(N2, [N | L2], L).

% runs down from N to 0

when N is negative, then return this intermediate built list, which is soln

L2 starts as empty list;
successive N's are added to it
(must count downward, or they'll be in
inverse order!)

L will be a variable term
UNTIL first clause activates,
at which time final solution
passed all the way to top

A common technique...

- **Note: this is a common Prolog programming technique, where the final result is unified when the recursion terminates...**

```
my_predicate( ... vars..., Final, Final) :- <terminate recursion>.
```

```
my_predicate( ... vars..., Intermed, Final) :- % recursive case(s)  
    <some processing>,  
    <add to Intermed to get Intermed2>,  
    my_predicate(...vars..., Intermed2, Final).
```

A Common list building error

- Building up list during calls, but not returning as final result:

```
make_intlist(N, L) :- N < 0.
```

```
make_intlist(N, L) :-
```

```
    N >= 0,
```

```
    N2 is N - 1,           % runs from N down to 0
```

```
    make_intlist(N2, [N | L]).
```

- Problem: the term “[N | L]” in the recursive call builds up an answer
- however, looking at the head of the rule, only L is available to the calling routine --> the computed “N” result is not passed back!

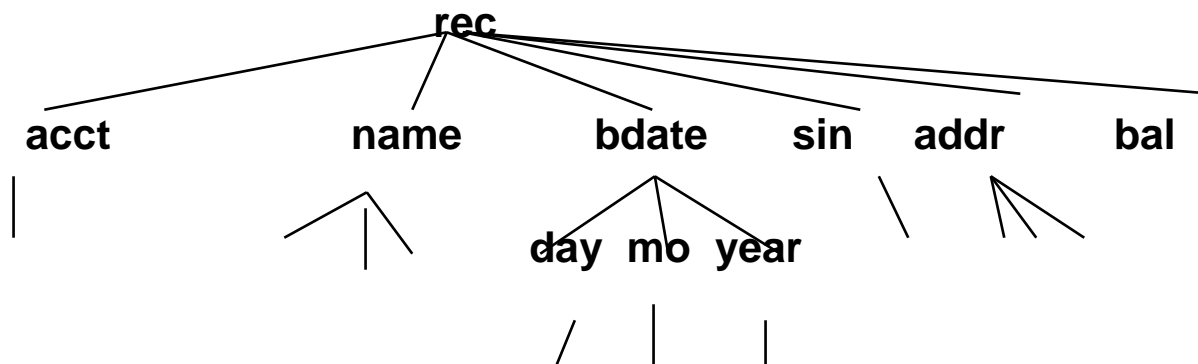
Other data structures

- Prolog's symbolic denotation of data means that complex data structures are easily derived

1. database records: invent a convention for representing data

- eg. define a structure 'rec' with 6 args, each argument = DB field
- first arg is key
- other args are interpreted according to their position in record, so programs must know this conventional structure

`rec(acct(Acct),name(First,MI,Last), bdate(day(D),mo(M),year(Y)),
sin(SIN,) addr(Street,City,Prov), bal(Balance))`



Data Structures: Records

- There are a number of ways of storing a database with this information

(a) List: [Rec1, Rec2, Rec3, , RecK]

```
add_rec(Rec, OldDB, [Rec|OldDB]) :-  
    get_key(Rec, Key),  
    \+ member(rec(Key,_,_,_,_), OldDB).
```

```
get_rec(Key, DB, rec(Key,A,B,C,D,E)) :-  
    member(rec(Key,A,B,C,D,E), DB).
```

```
del_rec(Key,DB, NewDB) :-  
    delete(rec(acct(Key),_,_,_,_), DB, NewDB).
```

```
get_key(rec(Key,_,_,_,_), Key).
```

```
delete(X,[X|R], R).  
delete(X, [Y|R], [Y|R2]) :- delete(X, R, R2).
```

Data Structures

2. Records: another approach is to label each field, and place fields in a list structure themselves

[name(F,MI,L), acct(Acct), sin(SIN), address(S,C,P),...]

Database: [[rec1], [rec2], ..., [recK]] (ie. a list of lists)

advantage: 1. fields can be in any order

2. easy to modify records: just add another labelled element

disadvantage: 1. To find a field, must search the record list

get_key(Rec, Key) :- member(acct(Key), Rec). % searches rec list for Key

get_rec(Key, DB, Rec) :-

member(Rec, DB), % backtracking selects each element of list DB

get_key(Key, Rec). % does this record match?

- **Records & databases: note there is another way to represent them - as Prolog facts!**

`rec(acct(38349), name(john,j,smith),).`

or

`rec([acct(38349), sin(483-234-113), name(john,j,smith),...]).`

- **We will discuss how to add or delete such records later**
 - hint: use Prolog's builtin "assert" and "retract" predicates
 - (Prolog also has efficient means for accessing such records)

Data Structures: stacks

3. Stack: use the list

% push(Item, Stack, NewStack) succeeds if, by pushing Item on Stack, we get

% NewStack

push(Item, Stack, [Item | Stack]).

% pop(Item, Stack, NewStack) succeeds if Item is popped from Stack, resulting

% in NewStack

pop(Item, [Item| Rest], Rest).

% top and is_empty are self-explanatory...

top(Top, [Top | _]).

is_empty([]).

- **Note that most programs don't use these. Instead, stacks are implemented directly as lists in arguments**

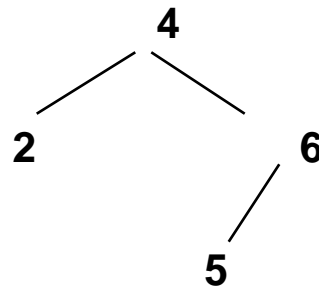
4. (sorted) Binary Tree

- list is too inefficient for large databases; if database has K records, a get or delete requires an average of $K/2$ comparisons during list search
- binary trees permit DB to be kept in order all the time
- two types of nodes:

nil : empty branch/tree

tree(Left_branch, Key, Right_branch) : tree node with data

- hence **tree(tree(nil,2,nil), 4, tree(tree(nil,5,nil), 6, nil))** represents:

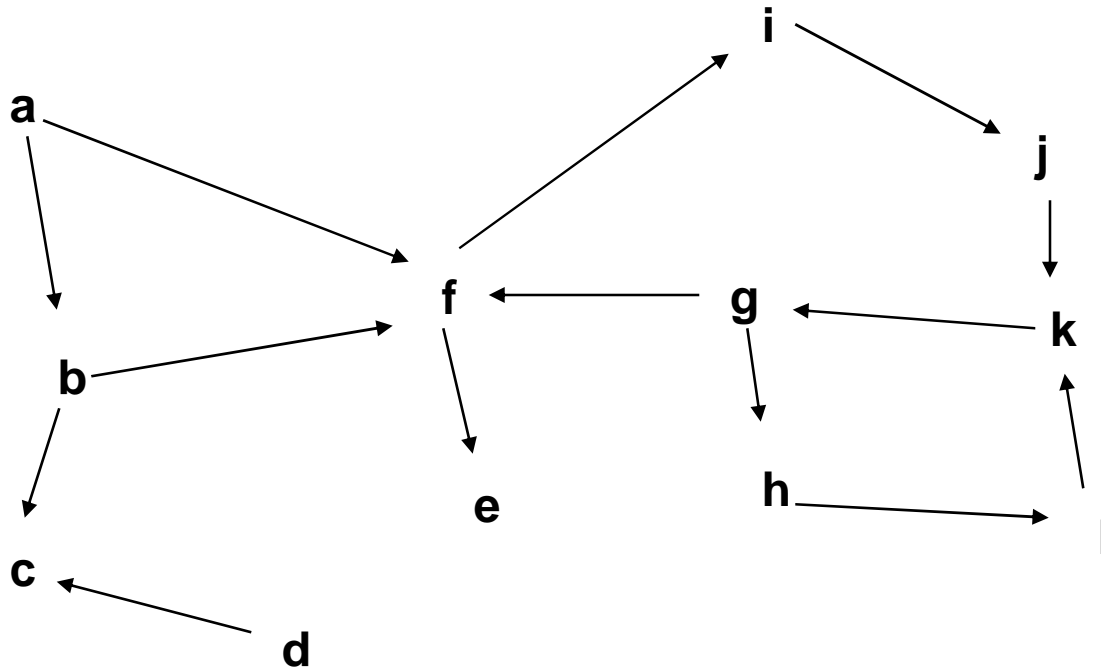


Data structures: binary trees

- **binary trees are very useful when they are sorted: keys on the left branch are less than the node, and keys on right are greater**

```
% add_bintree(Key, OldTree, NewTree)...  
add_bintree(Key, nil, tree(nil, Key, nil)).  
add_bintree(Key, tree(L, Key, R), tree(L, Key, R)).  
add_bintree(Key, tree(L, K, R), tree(L2, K,R)) :-  
    Key < K,  
    add_bintree(Key, L, L2).  
add_bintree(Key, tree(L, K, R), tree(L, K, R2)) :-  
    Key > K,  
    add_bintree(Key, R, R2).
```

Data Structures: graphs



5. Representing directed graphs as program clauses

edge(d, c).	edge(f, e).	edge(h, l).
edge(b, c).	edge(f, i).	edge(l, k).
edge(b, f).	edge(i, j).	edge(g, f).
edge(a, b).	edge(j, k).	edge(g, h).
edge(k, g).	edge(a, f).	

- **5. Graphs (cont)**

- Can then search for connections within the graph:

`path(A, B) :- edge(A,B).`

`path(A,B) :- edge(A,C), path(C,B).`

note similarity with 'ancestor'

`ancestor(A,B) :- parent(A,B).`

`ancestor(A,B) :- parent(A,C), ancestor(C,B).`

- However, unlike family relationships, graphs can have loops
- using Prolog's backtracking, it can easily get sidetracked into a loop
- eg. `?- path(j,a)` will go thru: `j - k - g - f - i - j - etc`

- **6. Graphs: another approach: use a list structure**

[v(Node1, Nodelist1), v(Node2, Nodelist2), ...]

then

**graph([v(a, [b, f]), v(b, [c,f]), v(d, [f]), v(f, [e,i]), v(g, [f,h]),
v(i, [j]), v(j, [k]), v(k, [g]), v(l, [k]), v(h, [l])]).**

- **Can write a 'path' routine which keeps track of places it has already been to; if it has been to a place already, then don't go there again!
(can also do this with the other representation)**

Data Structures: graphs

path(Start, Finish) :- graph(G), smart_path(Start, Finish, G, []).

**smart_path(A, B, G, _) :-
 member(v(A,L), G),
 member(B, L).**

**smart_path(A, B, G, Previous) :-
 member(v(A,L), G),
 member(C, L),
 \+ member(C, Previous),
 smart_path(C, B, G, [A | Previous]).**

Data structures

- Of course, structures permit arbitrarily complex data structures to be represented

`foo(List, cons1, Stack_of_bintrees, Bintree_of_stacks, etc...)`

- The power of symbolic execution is that you can define these structures symbolically, without regards to implementation details (memory allocation, pointers, ...)