Programming Paradigms CSI2120 – Winter 2018

Jochen Lang

EECS, University of Ottawa

Canada

uOttawa

L'Université canadienne
Canada's university

Université d'Ottawa | University of Ottawa

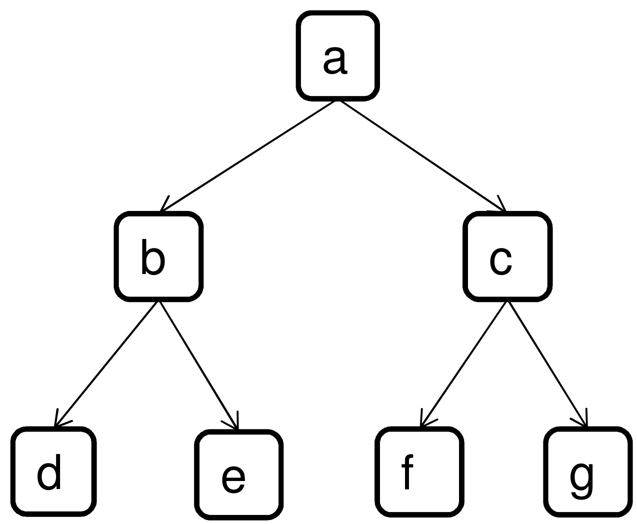
uOttawa.ca

Logic Programming in Prolog

- **Data structures**
- **Trees**
  - Representation
  - Examples
  - Binary search tree
- **Graphs**
  - Representation
  - Graph problems

Binary Trees

- **Tree where each element has one parent and up to two children**
  - Common data structure



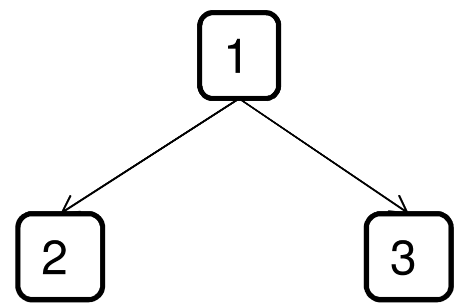
Binary Trees in Prolog

- **Define a fact for a node in the data structure**

  `t(element, left, right)`

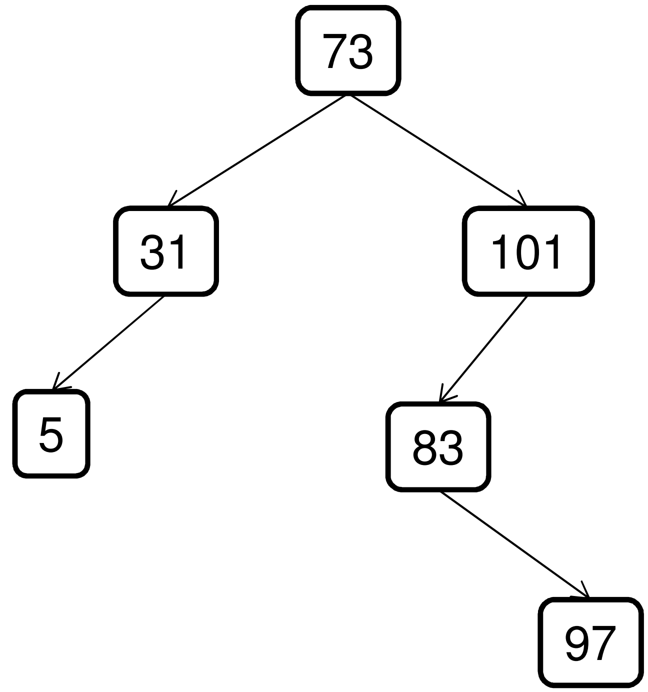
  - `element` is the value stored at the node
  - `left` is the left subtree
  - `right` is the right subtree
  - an empty subtree can be marked with a '`nil`'
- **A tree with only the root node is `t(1,nil,nil)`**
- **A balanced binary tree with three nodes**

  `t(1,t(2,nil,nil),t(3,nil,nil)).`



A Binary Tree

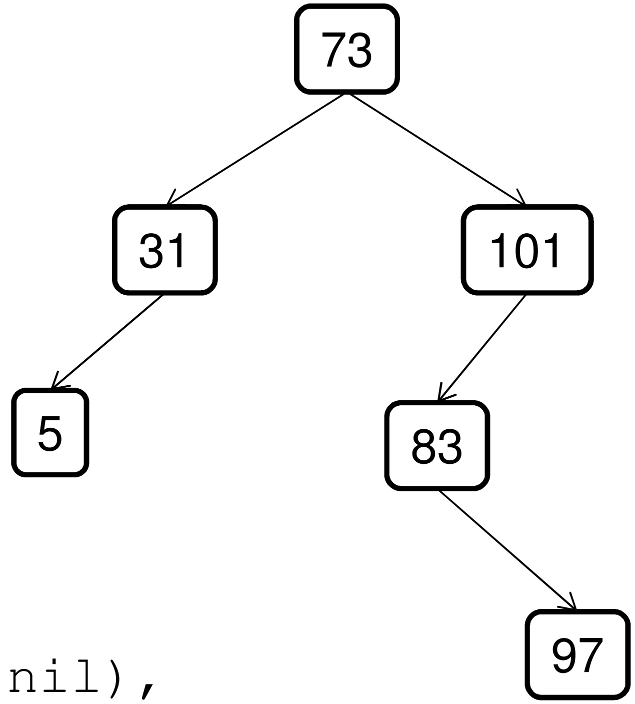
```
treeA(X) :- X=
t(73,
  t(31,
    t(5,nil,nil),
    nil),
  t(101,
    t(83,nil
      t(97,nil,nil)),
    nil)).
```



Inorder Traversal

```
inorder(nil).
inorder(t(Root,Left,Right)) :-
      inorder(Left),
      write(Root),
      write(' '),
      inorder(Right).

?-  treeB(X),  inorder(X).
5 31 73 83 97 101
X = t(73, t(31, t(5, nil, nil), nil),
t(101, t(83, nil, t(97, nil, nil)),
nil)).
```



Binary Search Tree

- **Sort predicate (assuming no duplicates)**

```
precedes(Key1, Key2) :- Key1 < Key2.
```

- **Boundary case: Searched for node found**

```
binarySearch(Key, t(Key, _, _)).
```

- **Search in left subtree**

```
binarySearch(Key, t(Root, Left, _)) :-
    precedes(Key, Root),
    binarySearch(Key, Left).
```

- **Search in right subtree**

```
binarySearch(Key, t(Root, _, Right)) :-
    precedes(Root, Key),
    binarySearch(Key, Right).
```

Element Insertion in a BST

- **Boundary case insert new leaf node**

```
insert(Key, nil, t(Key, nil, nil)).
```

- **Insert new node on the left**

```
insert(Key, t(Root, Left, Right),
       t(Root, LeftPlus, Right)) :-
    precedes(Key, Root),
    insert(Key, Left, LeftPlus).
```

- **Insert new node on the right**

```
insert(Key, t(Root, Left, Right),
       t(Root, Left, RightPlus)) :-
    precedes(Root, Key),
    insert(Key, Right, RightPlus).
```

Deleting a Key at the Root

- **Boundary case replace key with the right subtree**

```
deleteBST(Key, t(Key, nil, Right), Right).
```

- **Boundary case replace key with the left subtree**

```
deleteBST(Key, t(Key, Left, nil), Left).
```

- **Delete root and replace with maximum left key**

```
deleteBST(Key, t(Key, Left, Right),
            t(NewRoot, NewLeft, Right)) :-
     removeMax(Left, NewLeft, NewRoot).
```

  – arguments of removeMax

```
% removeMax(Tree,NewTree,Max)
```

Deleting any Key

- **Search on the left subtree for key to delete**

```
deleteBST(Key, t(Root, Left, Right),
             t(Root, LeftSmaller, Right)) :-
      precedes(Key, Root),
      deleteBST(Key, Left, LeftSmaller).
```

- **Search on the right subtree for key to delete**

```
deleteBST(Key, t(Root, Left, Right),
             t(Root, Left, RightSmaller)) :-
      precedes(Root, Key),
      deleteBST(Key, Right, RightSmaller).
```

Deleting the Maximum Element

- **boundary case right-most node is maximum**

```
removeMax(t(Max, Left, nil), Left, Max).
```

- **recursion on the right of the root node (for tree nodes sorted with less than).**

```
removeMax(t(Root, Left, Right),
          t(Root, Left, RightSmaller), Max) :-
    removeMax(Right, RightSmaller, Max).
```

General Graphs

- **A binary tree is a tree, and a tree is a (restricted) graph**
- **Graph representation**

```
g([Node,…],[edge(Node1,Node2,Weight),…]).
```

  - directed edge

```
edge(g(Ns,Edges),N1,N2,Weight):-
        member(edge(N1,N2,Weight),Edges).
```

  - undirected edge

```
edge(g(Ns,Edges),N1,N2,Weight):-
        member(edge(N1,N2,Weight),Edges);
        member(edge(N2,N1,Weight),Edges).
```

Neighbors of a Node

- **Find all neighboring nodes and the connecting edge (use with `edge/4` predicate).**

```
neighbors(Graph,Node,Neighbors):-
     setof((N,Edge),edge(Graph,Node,N,Edge),Neighbors).
```

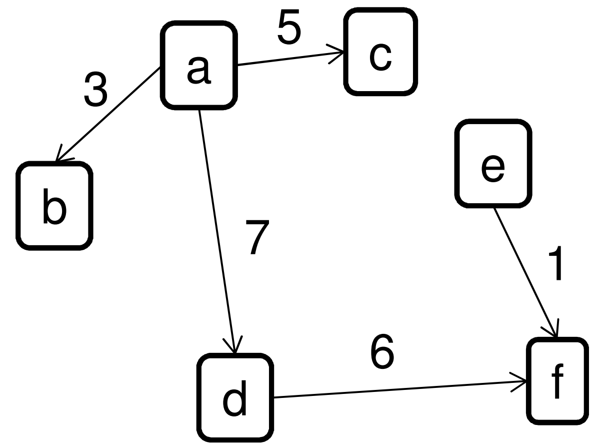
  – Define a graph

```
graphA(X) :- X=g([a,b,c,d,e,f],
                 [edge(a,b,3), edge(a,c,5), edge(a,d,7),
                  edge(e,f,1), edge(d,f,6)]).
```

  – Example queries

```
?- graphA(X), neighbors(X,c,V).
V = [ (a, 5)].

?- graphA(X), neighbors(X,a,V).
V = [ (b, 3), (c, 5), (d, 7)].
```



Graph Coloring

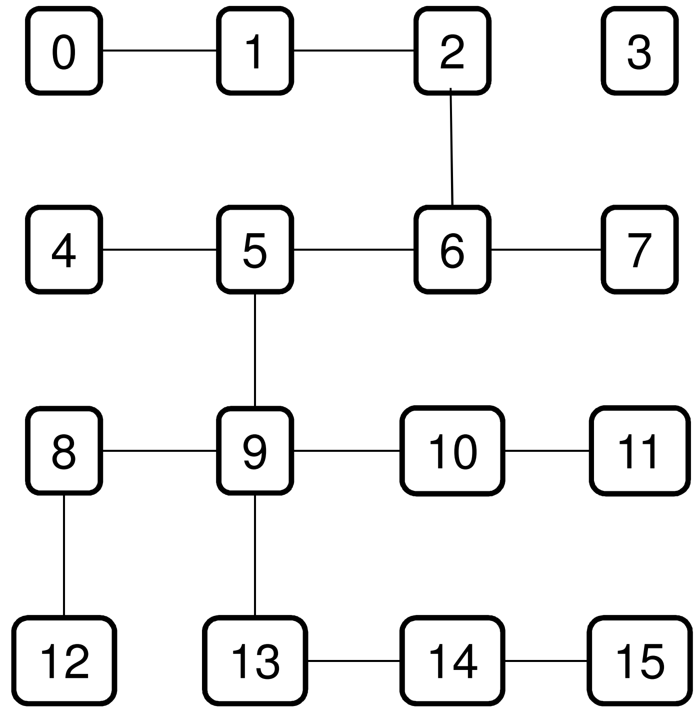
```
color(g(Ns,Edges),Colors,GC):-
    generate(Ns,Colors,GC),
    test(Edges,GC).
generate([],_,[]).
generate([N|Ns],Colors,[(N,C)|Q]):-
    member(C,Colors),
    generate(Ns,Colors,Q).
test([],_).
test([edge(N1,N2,_)|Ns],GC):-
    member((N1,C1),GC),
    member((N2,C2),GC),
    C1\=C2,
    test(Ns,GC).
```

Graph Coloring Queries

```
?- graphA(X), color(X,[red,blue,white,green],V).
X = g([a, b, c, d, e, f], [edge(a, b, 3), edge(a,
c, 5), edge(a, d, 7), edge(e, f, 1), edge(d, f,
6)]),
V = [ (a, red), (b, blue), (c, blue), (d, blue),
(e, red), (f, white)] ;
X = …,
V = [ (a, red), (b, blue), (c, blue), (d, blue),
(e, red), (f, green)] ;
X = …,
V = [ (a, red), (b, blue), (c, blue), (d, blue),
(e, blue), (f, red)] ;
…
```

Graph Problem: Labyrinth

```
link(0,1).  % start = 0
link(1,2).
link(2,6).
link(6,5).
link(6,7).
link(5,4).
link(5,9).
link(9,8).
link(8,12).
link(9,10).
link(10,11).
link(9,13).
link(13,14).
link(14,15).  % finish = 15
```



Labyrinth Solution

- **Predicate generating undirected edges**

```
successor(A,B) :- link(A,B).
successor(A,B) :- link(B,A).
```

- **Define the finish node**

```
finish(15).
```

- **Boundary case if finish is reached**

```
pathFinder([Last|Path],[Last|Path]) :-
        finish(Last).
```

- **Go to the next node in a depth first manner unless it is a loop**

```
pathFinder([Curr|Path],Solution) :-
        successor(Curr,Next),
        \+member(Next,Path),write(Next),nl,
        pathFinder([Next,Curr|Path],Solution).
```

Example: Labyrinth

```
?- pathFinder([0],S).
1
2
6
5
4
9
8
12
10
11
13
14
15
S = [15, 14, 13, 9, 5, 6, 2, 1, 0] ;
7
false.
```

Summary

- **Binary tree**
  - tree representation
  - binary search tree
  - insert an element
  - delete an element
- **Graphs**
  - graph representation
  - graph search
  - graph coloring
  - labyrinth