Lecture 13: Special Applications of LP

Mircea Marin
West University of Timişoara
mircea.marin@e-uvt.ro

May 24, 2021

Requirements:

- Manipulation of a simple binary search tree whose nodes are indexed by integers (student ID), and satellite data is a string (student name):

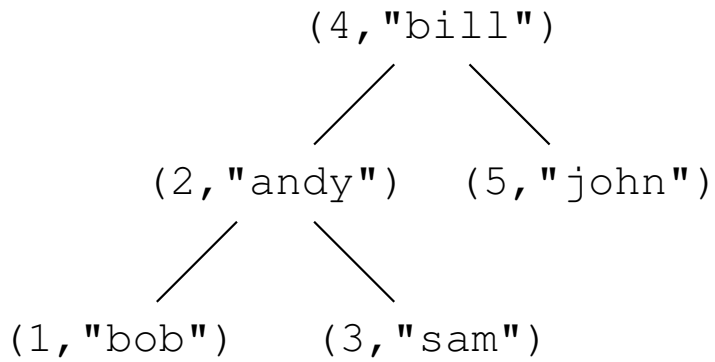
  bst `::=` `nil | bt(`*key*`,`*string*`,`*bst*`,`*bst*`)`
  key `::=` *integer*

- Provide the user with a menu to choose and perform one of the following possible actions:
  1. Create an empty tree (the current tree).
  2. Insert/modify a node in the current tree.
  3. Delete a node with a given key.
  4. Find a node with a given key and show its content (=satellite data).
  5. Show the whole content of tree, in increasing order of the key nodes.
  6. Stop the application.

1. Binary search trees

Data representation. Implementation of a menu

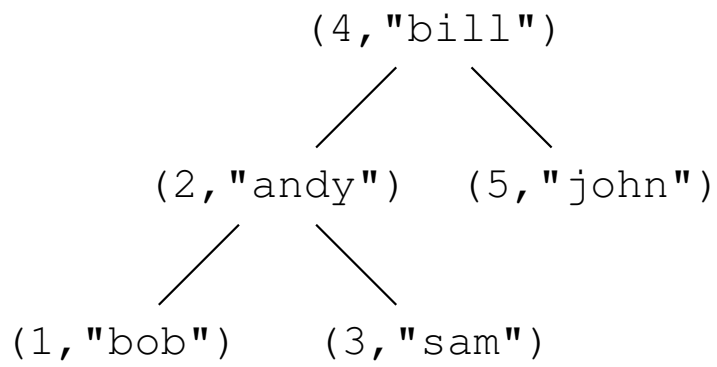
```
bt(4,"bill",
 bt(2,"andy",
  bt(1,"bob",nil,nil),
  bt(3,"sam",nil,nil)),
 bt(5,"john",nil,nil))
```



1. Binary search trees

Data representation. Implementation of a menu

```
bt(4,"bill",
 bt(2,"andy",
  bt(1,"bob",nil,nil),
  bt(3,"sam",nil,nil)),
 bt(5,"john",nil,nil))
```

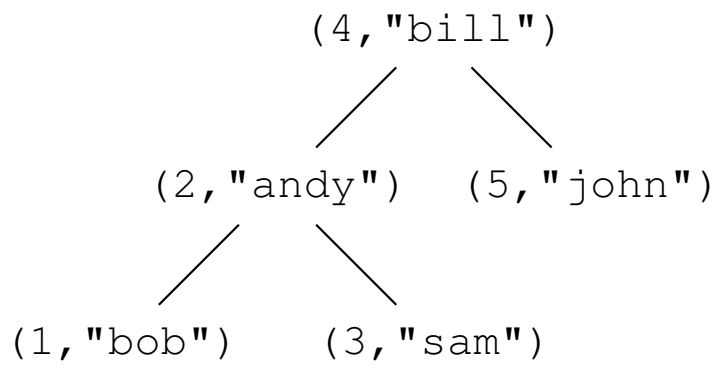



```
menu(Tree) :- nl,
   writeln('1. Create empty tree'),
   writeln('2. Insert or modify node'),
   writeln('3. Delete node'),
   writeln('4. Find node'),
   writeln('5. Show tree'),
   writeln('6. Stop'),
   write('Type option number followed by dot: '),
   read(Q),Q \= 6,!,
   action(Q,Tree,NewTree),
   menu(NewTree).
menu(_) :- writeln('Stopped.').
```

1. Binary search trees

Data representation. Implementation of a menu

```
bt(4,"bill",
 bt(2,"andy",
  bt(1,"bob",nil,nil),
  bt(3,"sam",nil,nil)),
 bt(5,"john",nil,nil))
```




```
menu(Tree) :- nl,
   writeln('1. Create empty tree'),
   writeln('2. Insert or modify node'),
   writeln('3. Delete node'),
   writeln('4. Find node'),
   writeln('5. Show tree'),
   writeln('6. Stop'),
   write('Type option number followed by dot: '),
   read(Q),Q \= 6,!,
   action(Q,Tree,NewTree),
   menu(NewTree).
menu(_) :- writeln('Stopped.').
```

Remark: The program is started with `?-menu(nil).`

The `menu` predicate: Implementation details

```
menu(+Tree)
```

1) takes as input the current binary search tree
2) displays a menu and asks the user to choose what to do:

```
read(-Q)
```

instantiates `Q` with the action number chosen by the user.

4) If `Q` $\neq$ `6`, answer the sub-query

```
action(+Q,+Tree,-NewTree)
```

by performing action `Q` on the current tree `Tree`

$\Rightarrow$ the current tree becomes `NewTree`, and computation continues with `menu(NewTree)`.

5) If `Q` $=$ `6`, computation stops after printing the message `'Stopped.'`.

```
% tree creation
action(1,_,nil).
```

Actions: tree creation; node insertion and modification

```
% tree creation
action(1,_,nil).
% node insertion or modification
action(2,Tree,NewTree) :-
    write('Enter key: '),read(K),
    write('Enter data: '),read(D),
    insert(K,D,Tree,NewTree).
...
```

```
% tree creation
action(1,_,nil).
% node insertion or modification
action(2,Tree,NewTree) :-
    write('Enter key: '),read(K),
    write('Enter data: '),read(D),
    insert(K,D,Tree,NewTree).
...
% insert(+K,+D,+Tree,-NewTree)
```

```
% tree creation
action(1,_,nil).
% node insertion or modification
action(2,Tree,NewTree) :-
    write('Enter key: '),read(K),
    write('Enter data: '),read(D),
    insert(K,D,Tree,NewTree).
...
% insert(+K,+D,+Tree,-NewTree)
% base case 1
insert(K,D,nil,bt(K,D,nil,nil)).
```

```
% tree creation
action(1,_,nil).
% node insertion or modification
action(2,Tree,NewTree) :-
    write('Enter key: '),read(K),
    write('Enter data: '),read(D),
    insert(K,D,Tree,NewTree).
...
% insert(+K,+D,+Tree,-NewTree)
% base case 1
insert(K,D,nil,bt(K,D,nil,nil)).
% base case 2: node found ⇒ update satellite data
insert(K,D,bt(K,_,T1,T2),bt(K,D,T1,T2)) :- !.
```

```
% tree creation
action(1,_,nil).
% node insertion or modification
action(2,Tree,NewTree) :-
   write('Enter key: '),read(K),
   write('Enter data: '),read(D),
   insert(K,D,Tree,NewTree).
...
% insert(+K,+D,+Tree,-NewTree)
% base case 1
insert(K,D,nil,bt(K,D,nil,nil)).
% base case 2: node found ⇒ update satellite data
insert(K,D,bt(K,_,T1,T2),bt(K,D,T1,T2)) :- !.
% recursive case 1
insert(K,D,bt(K1,D1,T1,T2),bt(K1,D1,NewT1,T2)) :-
   K<K1,!,insert(K,D,T1,NewT1).
% recursive case 2
insert(K,D,bt(K1,D1,T1,T2),bt(K1,D1,T1,NewT2)) :-
   insert(K,D,T2,NewT2).
```

Node deletion (1)

```
% node deletion
action(3,Tree,NewTree) :-
  write('Enter key of node to delete: '),
  read(Key),elim(Key,Tree,NewTree).
...
% elim(+K,+Tree,-NewTree)
% base case
elim(_,nil,nil).
elim(K,bt(K1,D,T1,T2),bt(K1,D,NewT1,T2)) :-
  K<K1,!,elim(K,T1,NewT1).
elim(K,bt(K1,D,T1,T2),bt(K1,D,T1,NewT2)) :-
  K>K1,!,elim(K,T2,NewT2).
elim(K,bt(K,_,nil,T2),T2):-!.
elim(K,bt(K,_,T1,nil),T1):-!.
% recursive case: T1 and T2 are not nil
elim(K,bt(K,D,T1,T2),NewTree):- ... % see next slide
```

How to delete the root node of `bt(K,D,T1,T2)` when both `T1,T2` are not `nil`?

Main idea: insert subtree `T2` in `T1`, as right subtree of node `P` with largest key in `T1`.
Note: `P` is the root of a binary search tree `bt(K1,D1,T1,nil)`

```
elim(K,bt(K,D,T1,T2),NewTree):-
  insertTree(T1,T2,NewTree).

% insertTree(+Tree,+T,-NewTree)
% binds NewTree to the result of inserting T in Tree,
% as right subtree of the node with largest key in Tree
insertTree(bt(K,D,T1,nil),T,bt(K,D,T1,T)):-!.
insertTree(bt(K,D,T1,T2),T,bt(K,D,T1,NewT2)):-
  insertTree(T2,T,NewT2).
```

```
% find a node
action(4,Tree,Tree) :-
    write('Enter key of node to find: '),
    read(Key),findNode(Key,Tree).
...
% findNode(+K,+Tree)
findNode(_,nil) :- writeln("Node not found").
findNode(K,bt(K,D,_,_)) :- !,
  writeln(D).
findNode(K,bt(K1,_,T1,_)) :- K<K1,!,
  findNode(K,T1).
findNode(K,bt(_,_,_,T2)) :-
  findNode(K,T2).
```

1. Binary search trees

Action: display tree content (in inorder)

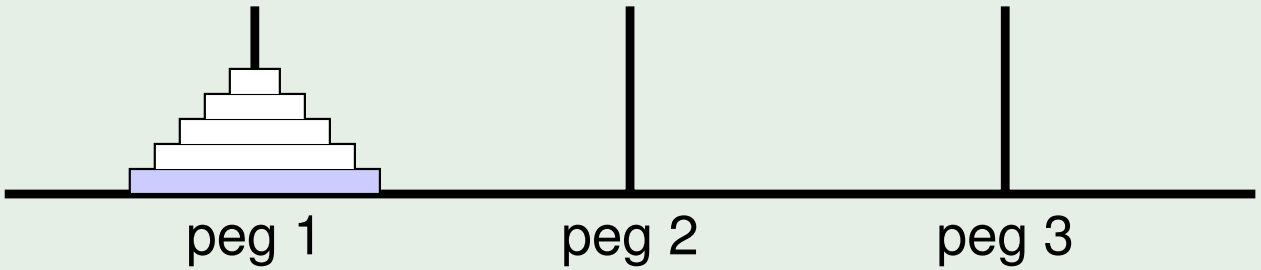
```
action(5,Tree,Tree) :-
    showTree(Tree).

% showTree(+Tree)
showTree(nil).
showTree(bt(K,D,T1,T2)):-
    showTree(T1),
    write('Key: '),write(K),
    write(', data: '), writeln(D),
    showTree(T2).
```

2. Hanoi towers

Goal: Move n disks from peg 1 to peg 3
Rules of the game: move repeatedly only one disk from one peg on another peg. A disk must always be taken from top, and placed on the floor or on top of a larger disks.
Initial configuration: All disks are on peg 1, in decreasing order of size.

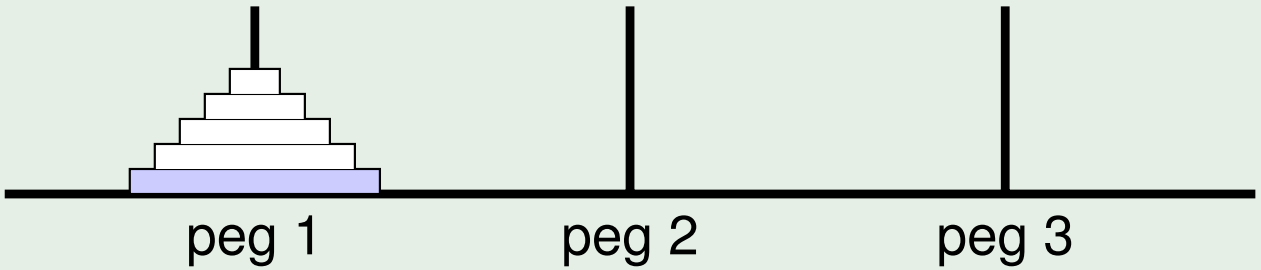
Example (Goal: move $n = 5$ disks from peg 1 on peg 2)



2. Hanoi towers

Goal: Move n disks from peg 1 to peg 3
Rules of the game: move repeatedly only one disk from one peg on another peg. A disk must always be taken from top, and placed on the floor or on top of a larger disks.
Initial configuration: All disks are on peg 1, in decreasing order of size.

Example (Goal: move $n = 5$ disks from peg 1 on peg 2)

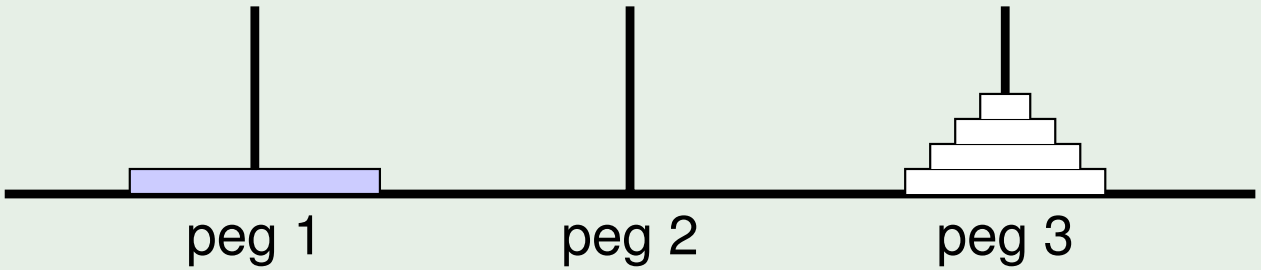


We can decompose this goal into 3 subgoals:

2. Hanoi towers

Goal: Move n disks from peg 1 to peg 3
Rules of the game: move repeatedly only one disk from one peg on another peg. A disk must always be taken from top, and placed on the floor or on top of a larger disks.
Initial configuration: All disks are on peg 1, in decreasing order of size.

Example (Goal: move $n = 5$ disks from peg 1 on peg 2)



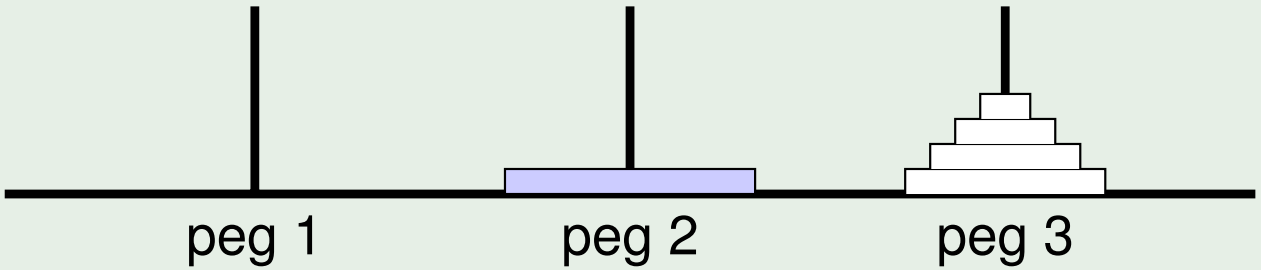
We can decompose this goal into 3 subgoals:

1. Move all disks, except the largest one, from peg 1 to peg 3.

2. Hanoi towers

Goal: Move n disks from peg 1 to peg 3
Rules of the game: move repeatedly only one disk from one peg on another peg. A disk must always be taken from top, and placed on the floor or on top of a larger disks.
Initial configuration: All disks are on peg 1, in decreasing order of size.

Example (Goal: move $n = 5$ disks from peg 1 on peg 2)



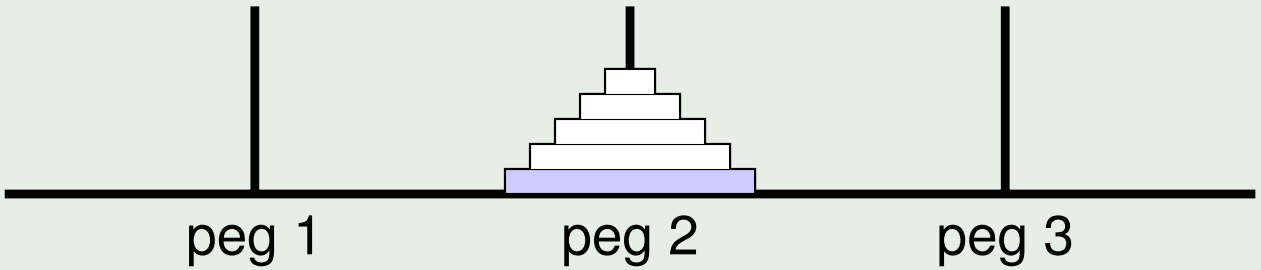
We can decompose this goal into 3 subgoals:

1. Move all disks, except the largest one, from peg 1 to peg 3.
2. Move the largest disk from peg 1 on peg 2.

2. Hanoi towers

Goal: Move n disks from peg 1 to peg 3
Rules of the game: move repeatedly only one disk from one peg on another peg. A disk must always be taken from top, and placed on the floor or on top of a larger disks.
Initial configuration: All disks are on peg 1, in decreasing order of size.

Example (Goal: move $n = 5$ disks from peg 1 on peg 2)



We can decompose this goal into 3 subgoals:

1. Move all disks, except the largest one, from peg 1 to peg 3.
2. Move the largest disk from peg 1 on peg 2.
3. Move all disks from peg 3 on peg 2.

2. Hanoi towers

Auxiliary predicates

- `hanoi(N)` moves `N` disks from peg `1` to peg `2` using peg `3` as intermediary. It is assumed that, initially, the disks are placed in decreasing order of their size.
- `move(+N,+A,+B,+C)` moves the top `N` disks from peg `A` to peg `B`, using peg `C` as an intermediary peg.
- A movement from peg `A` to peg `B` is signaled by writing the message `move from A to B`

```
hanoi(N):-move(N,1,2,3).
move(0,_,_,_) :- !.
move(N,A,B,C) :-
    M is N-1,
    move(M,A,C,B),
    write('move from '),write(A),
    write(' to '),writeln(B),
    move(M,C,B,A).
```

- Assume a weighted digraph whose arcs are represented by facts

  `arc(X,Y,W)`

  where `W` is a numeric value for the weight of the arc from `X` to `Y`.

Example

```
arc(newcastle,carlisle,58).
arc(carlisle,penrith,23).
arc(smallville,metropolis,15).
arc(penrith,darlington,52).
arc(smallville,ambridge,10).
arc(workington,carlisle,33).
arc(workington,ambridge,5).
arc(workington,penrith,39).
arc(darlington,metropolis,25).
```

Best-first search strategy = adjustment of breadth-first search strategy (for shortest paths, as number of edges), which finds a path with minimum weight from a given source node to a given destination node.

- For every shortest path found so far, it retains a record of its length
  - `r(`W`,[`$X_n, \ldots, X_2, X_1$`])` is a weight-annotated path.
    It records the fact that `[`$X_1, \ldots, X_n$`]` is a shortest path from X_1 to X_n, with weight W. Note: W is the sum of weights of arcs $X_1 \to X_2, \ldots, X_{n-1} \to X_n$ of path `[`$X_1, X_2, \ldots, X_n$`]`.
- Each time, best-first search selects and extends the path in the record `r(`W`,[`$X_n, \ldots, X_2, X_1$`])` for which W has the smallest value.

`propagate_min(+L,-R)` takes as input a nonempty list `L` of shortest-path records, and binds `R` to a rearrangement of list `L` where a record with minimum weight occurs first.

```
propagate_min([Rec],[Rec]):-!.
propagate_min([Rec|L],[A,B|T]):-
    propagate_min(L,[Rec1|T]),
    rearrange(Rec,Rec1,A,B).
% rearrange(+R1,+R2,-A,-B)  -- what is this doing?
rearrange(r(W1,Xs),r(W2,Ys),
          r(W2,Ys),r(W1,Xs)) :- W1>W2,!.
rearrange(R1,R2,R1,R2).
```

`propagate_min(+L,-R)` takes as input a nonempty list `L` of shortest-path records, and binds `R` to a rearrangement of list `L` where a record with minimum weight occurs first.

```
propagate_min([Rec],[Rec]):-!.
propagate_min([Rec|L],[A,B|T]):-
    propagate_min(L,[Rec1|T]),
    rearrange(Rec,Rec1,A,B).
% rearrange(+R1,+R2,-A,-B)  -- what is this doing?
rearrange(r(W1,Xs),r(W2,Ys),
          r(W2,Ys),r(W1,Xs)) :- W1>W2,!.
rearrange(R1,R2,R1,R2).
```

Example

```
?-propagate_min([r(3,[a,x]),r(1,[b,x]),r(0,[x])],R).
R = [r(0,[x]),r(3,[a,x]),r(1,[b,x])].
```

`extension_ok(`$Y, [X_n, \ldots, X_1], W, NewW$`)` holds if

- $[X_1, \ldots, X_n]$ is a path from X_1 to X_n with weight W
- `arc(`X_n, Y, D`)` is a fact
- $Y \notin [X_1, \ldots, X_n]$, thus it is ok to build the extended path $[X_1, \ldots, X_n, Y]$ with weight $NewW = W + D$

```
extension_ok(Y,[X|Xs],W,NewW) :- arc(X,Y,D),
  not(member(Y,[X|Xs])),NewW is W+D.
```

`extend(+WAPaths,+Y,-WAPath)` binds `WAPath` to a weight-annotated path `WAPath` with minimum weight to destination `Y`, by extending with best-first strategy the weight-annotated paths from the list `WAPaths`:

```
extend([r(W,[Y|Xs])|_],Y,r(W,[Y|Xs])) :- !.
extend([r(W,Xs)|Rs],Y,WAPath) :-
  findall(r(NewW,[Z|Xs]),
          extension_ok(Z,Xs,W,NewW),
          Rs1),
  append(Rs,Rs1,LsNew),
  propagate_min(LsNew,RsNew),
  extend(RsNew,Y,WAPath).
```

`best_path(+X,+Y,r(-W,-Path))` binds `Path` to a path with minimum weight, if there is one, and `W` to its weight. The path is found with best-first search strategy.

```
best_path(X,Y,r(W,Path)) :-
  extend([r(0,[X])],Y,r(W,T)),
  reverse(T,Path).
```

`best_path(+X,+Y,r(-W,-Path))` binds `Path` to a path with minimum weight, if there is one, and `W` to its weight. The path is found with best-first search strategy.

```
best_path(X,Y,r(W,Path)) :-
  extend([r(0,[X])],Y,r(W,T)),
  reverse(T,Path).
```

```
?- best_path(workington,darlington,WAPath).
WAPath = r(91,[workington,penrith,darlington]).
```

References

1. Chapter 7 from
   - W.F. Clocksin, C.S. Mellish. *Programming in Prolog*, Fifth Edition. Springer 2003.
2. Section 9.4 from
   - A. M. Florea, B. Dorohonceanu, C. Frâncu. *Programare în Prolog pentru Inteligenţă Artificială*. Universitatea “Politehnica” Bucureşti. 1997.