CS370

# Symbolic Programming
## Declarative Programming

LECTURE 11: Operations on Data Structures


**Jong C. Park**

park@cs.kaist.ac.kr

**Computer Science Department**

**Korea Advanced Institute of Science and Technology**


**http://nlp.kaist.ac.kr/~cs370**

# Operations on Data Structures

- Sorting lists
- Representing sets by binary trees
- Insertion and deletion in a binary dictionary
- Displaying trees
- Graphs

# Sorting lists

- **Order: gt(X,Y)**
  - gt(X,Y) :- X > Y.
  - gt(X,Y) :- X @> Y.
- **Sorting**
  - sort(List, Sorted)

# Sorting lists

- Bubble sort
- Insertion sort
- Quicksort

# Sorting lists

- **Bubble sort**

```
bubble(List,Sorted) :-
  swap(List,List1), !,
  bubble(List1,Sorted).
bubble(Sorted,Sorted).

swap([X,Y|Rest],[Y,X|Rest]) :-
  gt(X,Y).
swap([Z|Rest],[Z|Rest1]) :-
  swap(Rest,Rest1).
```

# Sorting lists

- **Insertion sort**

```
insertsort([ ],[ ]).
insertsort([X|Tail],Sorted) :-
  insertsort(Tail,SortedTail),
  insert(X,SortedTail,Sorted).

insert(X,[Y|Sorted],[Y|Sorted1]) :-
  gt(X,Y), !,
  insert(X,Sorted,Sorted1).
insert(X,Sorted,[X|Sorted]).
```

# Sorting lists

- **Quicksort**

```
quicksort([ ],[ ]).
quicksort([X|Tail],Sorted) :-
  split(X,Tail,Small,Big),
  quicksort(Small,SortedSmall),
  quicksort(Big,SortedBig),
  conc(SortedSmall, [X|SortedBig], Sorted).
split(X,[ ],[ ],[ ]).
split(X,[Y|Tail],[Y|Small],Big) :-
  gt(X,Y), !, split(X,Tail,Small,Big).
split(X,[Y|Tail],Small,[Y|Big]) :-
  split(X,Tail,Small,Big).
```

# Sorting lists

- **Quicksort with difference lists**

```
quicksort(List,Sorted) :-
  quicksort2(List,Sorted-[ ]).

quicksort2([ ],Z-Z).
quicksort2([X|Tail], A1-Z2) :-
  split(X, Tail, Small, Big),
  quicksort2(Small, A1-[X|A2]),
  quicksort2(Big, A2-Z2).
```

# Representing sets by binary trees

## Use Prolog terms

- root -> principal functor of the term
- subtrees -> arguments
- Example
  - a(b,c(d))

## Pros and cons

-
-

# Representing sets by binary trees

## Use fixed functor names

- empty tree -> nil
- tree with root X -> t(L,X,R)
- Example

```
t(t(nil,b,nil),a,t(t(nil,d,nil),c,nil))
in(X,t(_,X,_)).
in(X,t(L,_,_)) :- in(X,L).
in(X,t(_,_,R)) :- in(X,R).
```

## Pros and cons

-
-

# Representing sets by binary trees

## ⊙ Use a binary dictionary

- tree with root X -> t(Left,X,Right)
  - node Y in Left -> gt(X,Y)
  - node Z in Right -> gt(Z,X)
  - Left and Right are binary trees

```
in(X,t(_,X,_)).
in(X,t(Left,Root,Right)) :-
  gt(Root,X), in(X,Left).
in(X,t(Left,Root,Right)) :-
  gt(X,Root), in(X,Right).
```

# Insertion and deletion in binary dictionary

- **Add X to D, resulting in D1**
  - addleaf(D,X,D1)

```
addleaf(nil, X, t(nil,X,nil)).
addleaf(t(Left,X,Right), X, t(Left,X,Right)).
addleaf(t(Left,Root,Right), X,
  t(Left1,Root,Right)) :-
  gt(Root,X),
  addleaf(Left, X, Left1).
addleaf(t(Left,Root,Right), X,
  t(Left,Root,Right1)) :-
  gt(X,Root),
  addleaf(Right, X, Right1).
```

# Insertion and deletion in binary dictionary

- **Delete X from D1, with the result in D2**
  - Method1: delleaf(D1,X,D2)

```
delleaf(D1,X,D2) :-
        addleaf(D2,X,D1).
```

# Insertion and deletion in binary dictionary

## ⊙ Delete X from D1, with the result in D2

- Method 2: del(D1,X,D2) using a binary dictionary

```
del(t(nil,X,Right), X, Right).
del(t(Left,X,nil), X, Left).
del(t(Left,X,Right), X, t(Left,Y,Right1)) :-
  delmin(Right, Y, Right1).
del(t(Left, Root, Right), X, t(Left1, Root, Right)) :-
  gt(Root,X), del(Left, X, Left1).
del(t(Left,Root,Right), X, t(Left,Root,Right1)) :-
  gt(X,Root), del(Right, X, Right1).
delmin(t(nil,Y,Right), Y, Right).
delmin(t(Left,Root,Right), Y, t(Left1,Root,Right)) :-
  delmin(Left, Y, Left1).
```

# Insertion and deletion in binary dictionary

## ⊙ add and delete (cf. Figure 9.14)

- The add/3 relation is defined non-deterministically.

```
add(Tree,X,NewTree) :- addroot(Tree,X,NewTree).
add(t(L,Y,R),X,t(L1,Y,R)) :- gt(Y,X), add(L,X,L1).
add(t(L,Y,R),X,t(L,Y,R1)) :- gt(X,Y), add(R,X,R1).
addroot(nil,X,t(nil,X,nil)).
addroot(t(L,Y,R),X,t(L1,X,t(L2,Y,R))) :-
   gt(Y,X), addroot(L,X,t(L1,X,L2)).
addroot(t(L,Y,R),X,t(t(L,Y,R1),X,R2)) :-
   gt(X,Y), addroot(R,X,t(R1,X,R2)).
?- add(nil,3,D1), add(D1,5,D2), add(D2,1,D3),
   add(D3,6,D), add(DD,5,D).
```

# Displaying trees

- **show(T)**

```
show(Tree) :- show2(Tree,0).

show2(nil,_).
show2(t(Left,X,Right),Indent) :-
  Ind2 is Indent + 2,
  show2(Right,Ind2),
  tab(Indent), write(X), nil,
  show2(Left,Ind2).
```

# Graphs

## ⊙ Representing graphs

- Method 1:

```
connected(a,b).
arc(s,t,3).
```

- Method 2:

```
graph([a,b,c,d],[e(a,b),e(b,d),e(b,c),e(c,d)]).
digraph([s,t,u,v],[a(s,t,3),a(t,v,1),a(t,u,5),
                   a(u,t,2),a(v,u,2)]).
```

# Graphs

## Representing graphs

- Method 3:

  [a->[b],b->[a,c,d],c->[b,d],d->[b,c]]

  [s->[t/3],t->[u/5,v/1],u->[t/2],v->[u/2]]

- Pros and cons
  -
  -

# Graphs

## Finding a path

- path(A,Z,G,P)
  - P is an acyclic path between A and Z in G.

path(a,d,G,[a,b,d])

```
path(A,Z,Graph,Path) :- path1(A,[Z],Graph,Path).
path1(A,[A|Path1],_,[A|Path1]).
path1(A,[Y|Path1],Graph,Path) :-
  adjacent(X,Y,Graph),
  not member(X,Path1),
  path1(A,[X,Y|Path1],Graph,Path).
adjacent(X,Y,graph(Nodes,Edges)) :-
  member(e(X,Y),Edges); member(e(Y,X),Edges).
```

# Graphs

## Hamiltonian path

- an acyclic path comprising all the nodes in the graph

```
hamiltonian(Graph,Path) :-
  path(_,_,Graph,Path),
  covers(Path,Graph).

covers(Path,Graph) :-
  not(node(N,Graph), not member(N,Path)).
```

# Graphs

## ⊙ Path with a cost

```
path(A,Z,Graph,Path,Cost) :-
  path1(A,[Z],0,Graph,Path,Cost).
path1(A,[A|Path1],Cost1,Graph,[A|Path1],Cost1).
path1(A,[Y|Path1],Cost1,Graph,Path,Cost) :-
  adjacent(X,Y,CostXY,Graph),
  not member(X,Path1),
  Cost2 is Cost1+CostXY,
  path1(A,[X,Y|Path1],Cost2,Graph,Path,Cost).
```

# Graphs

## Spanning tree

- G = (V,E) is a connected graph if there is a path from any node to any other node for every pair of nodes in G.
- T = (V, E') is a spanning tree of G where E' is a subset of E if
  - T is connected and
  - There is no cycle in T
- Example spanning trees
  - Tree1 = [a-b, b-c, c-d]
  - Tree2 = [a-b, b-d, d-c]
  - Tree3 = [a-b, b-d, b-c]

# Graphs

## ⊙ Finding a spanning tree of a graph

```
stree(Graph,Tree) :- member(Edge,Graph),
                     spread([Edge],Tree,Graph).
spread(Tree1,Tree,Graph) :-
  addedge(Tree1,Tree2,Graph), spread(Tree2,Tree,Graph).
spread(Tree,Tree,Graph) :-
  not addedge(Tree,_,Graph).
addedge(Tree,[A-B|Tree],Graph) :-
  adjacent(A,B,Graph), node(A,Tree), not node(B,Tree).
adjacent(Node1,Node2,Graph) :-
  member(Node1-Node2,Graph);
  member(Node2-Node1,Graph).
node(Node,Graph) :- adjacent(Node,_,Graph).
```

# Summary

- Sorting lists
- Representing sets by binary trees
- Insertion and deletion in a binary dictionary
- Displaying trees
- Graphs