IKI30820 – Logic Programming
Operations on Data Structures
Slide 10

Ari Saptawijaya (slides) Adila A. Krisnadhi (LaTeXadaptation)

Fakultas Ilmu Komputer
Universitas Indonesia

2009/2010 • Semester Gasal

Outline

Sorting lists

- A list can be sorted if there is an ordering relation between items in the list.
- Assume that there is an ordering relation:

  ```
  gt(X,Y)
  ```

  meaning that `X` is “greater” than `Y`
- If the objects are numbers, then the `gt` relation can be defined as:

  ```
  gt(X,Y) :- X>Y.
  ```

- If the objects are atoms, then the `gt` relation corresponds to the alphabetical order:

  ```
  gt(X,Y) :- X @> Y.
  ```

Sort relation

- Given the order relation `gt`, we can define a sort relation between two lists as:

  ```
  sort(List, SortedList)
  ```

  which holds when `List` is a list of items and `SortedList` is a list of the same items but sorted in ascending order according to the `gt/2` relation.

Insertion sort

To sort a nonempty list `L = [X|T]`

1. Sort the tail `T` of `L`
2. Insert the head `X` of `L` into the sorted tail at such position that the resulting list is sorted.

```
insertionSort([],[]).
insertionSort([X|T],Sorted) :-
  insertionSort(T,SortedTail),
  insert(X,SortedTail,Sorted).

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

gt(X,Y) :- X@>Y.
```

Quicksort

To sort a nonempty list `L = [X| T]`

1. Delete some element `X` (called the pivot) from `L` and split the rest of `L` into two lists:
   - `Small`: all elements in `L` that are smaller than `X`
   - `Big`: all other elements in `L`
2. Sort `Small` to obtain `SortedSmall`
3. Sort `Big` to obtain `SortedBig`
4. The whole sorted list is the concatenation of `SortedSmall` and `[X | SortedBig]`

```
quicksort( [], []).
quicksort( [X|Tail], Sorted) :-
   split( X, Tail, Small, Big),
   quicksort( Small, SortedSmall),
   quicksort( Big, SortedBig),
   append( 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).
```

```
% A more efficient implementation of quicksort using
% difference-list
quicksort( List, Sorted) :-
 quicksort2( List, Sorted - [] ).

% quicksort2( List, SortedDiffList): sort List,
% result is represented as difference list
quicksort2( [], Z - Z).
quicksort2( [X | Tail], A1 - Z2) :-
  split( X, Tail, Small, Big),
  quicksort2( Small, A1 - [X | A2] ),
  quicksort2( Big, A2 - Z2).
```

Mergesort

To sort a nonempty list `L`:

1. Divide `L` into two lists, `L1` and `L2`, of approximately equal length
2. Sort `L1` giving `S1`
3. Sort `L2` giving `S2`
4. Merge `S1` and `S2` to obtain the whole sorted list

```
% mergesort( List, SortedList ): sort List by the
% MergeSort algorithm.
mergesort([], []).
mergesort([X], [X]) :- !.
mergesort(List, SortedList) :-
   divide(List, List1, List2),
   mergesort(List1, Sorted1),
   mergesort(List2, Sorted2),
   merge(Sorted1, Sorted2, SortedList).

% divide a list into two lists of approximately
% equal length.
divide([], [], []).
divide([X], [X], []).
divide([X,Y|L], [X|L1], [Y|L2]) :-
   divide(L, L1, L2).
```

```
% merge( List1, List2, List3 ) is true if List1 and
% List2 are ordered lists that merge into the
% ordered list List3.
% Example:
% ?- merge([1, 3, 4], [0, 2, 5, 6], L).
% L = [0, 1, 2, 3, 4, 5, 6]

merge([], List, List) :- !.
merge(List, [], List).
merge([X|Tail1], [Y|Tail2], [X|Tail3]) :-
  X @< Y, !, merge(Tail1, [Y|Tail2], Tail3).
merge(List1, [Y|Tail2], [Y|Tail3]) :-
  merge(List1, Tail2, Tail3).
```

Outline

Binary trees

- A disadvantage of using a list for representing a collection of objects is that the membership testing or searching is relatively inefficient
- Binary tree can be used to facilitate more efficient implementation of membership relation
- A binary tree is either empty or it consists of three things:
  - a root
  - a left subtree
  - a right subtree

Binary search tree (binary dictionary)

- A non-empty binary tree `t(Left,X,Right)` is ordered from left to right if:
  1. all the nodes in the left subtree, `Left`, are less than `X`
  2. all the nodes in the right subtree, `Right`, are greater than `X`
  3. both subtrees are also ordered
- Such an ordered binary tree is called a binary search tree (binary dictionary)

Finding an item X in a binary dictionary D

- If X is the root of D, then X has been found,
- if X is less than the root of D, then search for X in the left subtree of D, otherwise
- search X in the right subtree of D;
- if D is empty, the search fails

```
% Finding an item X in a binary dictionary
% (binary search tree).
% in( X, Tree): X in binary dictionary Tree

in( X, t( _, X, _) ).
in( X, t( Left, Root, Right) ) :-
    gt( Root, X), % Root greater than X
    in( X, Left). % Search left subtree
in( X, t( Left, Root, Right) ) :-
    gt( X, Root), % X greater than Root
    in( X, Right). % Search right subtree

gt( X, Y ) :- X @> Y.
```

- The predicate `in` can be used for constructing a binary search tree.
- Draw the resulted trees.

```
?- in(d, BST), in(b, BST), in(f, BST), in(e, BST).
BST = t(t(_G256, b, _G258), d, t(t(_G264, e, _G266), f, _G262))

?- in(b, BST), in(d, BST), in(e, BST), in(f, BST).
BST = t(_G252, b, t(_G256, d, t(_G260, e, t(_G264, f, _G266))))
```

Insertion and deletion in binary dictionary: adding a leaf

- `addleaf( D, X, D1 )`: add `X` to `D` giving `D1`. Rules for adding at the leaf level:
  1. The result of adding `X` to the empty tree is the tree `t(nil, X, nil)`.
  2. If `X` is the root of `D` then `D1 = D` (no duplicate item is inserted).
  3. If the root of `D` is greater than `X` then insert `X` into the left subtree of `D`; if the root of `D` is less than `X` then insert `X` into the right subtree.

```
% addleaf( Tree, X, NewTree):
% inserting X as a leaf into binary dictionary Tree gives
% NewTree

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 ).
```

Deleting a leaf

- It is easy to delete a leaf. The deletion of a leaf can be defined as the inverse operation of inserting at the leaf level:

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

- Deleting an internal node X is more complicated.
  - Suppose that `X` has two subtrees, `Left` and `Right`.
  - After `X` is deleted, we have a hole in the tree and the subtrees `Left` and `Right` are no longer connected to the rest of the tree.
  - If `Left` or `Right` is empty then the non-empty subtree is connected to the parent of `X`.
  - If `Left` and `Right` are both non-empty then the leftmost node of `Right`, `Y`, is transferred from its current position upwards to fill the position of `X`.

```
% del( Tree, X, NewTree): deleting X from
% binary dictionary Tree gives NewTree
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( Tree, Y, NewTree):
% delete minimal item Y in binary dictionary
% Tree producing NewTree

delmin( t( nil, Y, R), Y, R).

delmin( t( Left, Root, Right), Y,
        t( Left1, Root, Right)) :-
  delmin( Left, Y, Left1).
```

Insertion into BST at any level of the tree I

```
% add( Tree, X, NewTree): inserting X at any level
% of binary dictionary Tree gives NewTree

% Add X as new root
add( Tree, X, NewTree) :-
    addroot( Tree, X, NewTree).

% Insert X into left subtree
add( t( L, Y, R), X, t( L1, Y, R)) :-
    gt( Y, X), add( L, X, L1).

% Insert X into right subtree
add( t( L, Y, R), X, t( L, Y, R1)) :-
    gt( X, Y), add( R, X, R1).
```

Insertion into BST at any level of the tree II

```
% addroot( Tree, X, NewTree): inserting X as
% the root into Tree gives NewTree

% Insert into empty tree
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)).
```

Displaying a binary tree

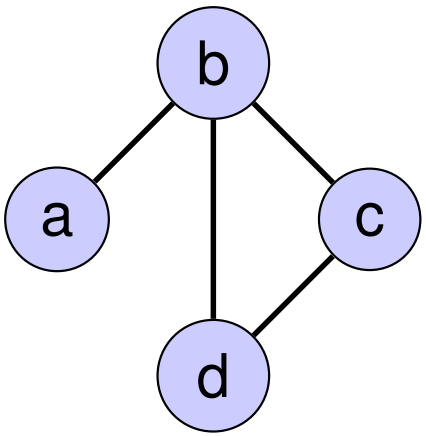
```
% show( Tree): display binary tree
show( Tree) :-
   show2( Tree, 0).

% show2( Tree, Indent): display Tree indented by Indent
show2( nil, _).

show2( t( Left, X, Right), Indent) :-
   Ind2 is Indent + 2,                % Indentation of subtrees
   show2( Right, Ind2),               % Display right subtree
   tab( Indent), write( X), nl,       % Write root
   show2( Left, Ind2).                % Display left subtree
```

Outline

Graphs

Various representations of graphs

- Each edge or arc is represented as a clause.
- A whole graph as structure.




```
connected(a,b).
connected(b,d).
connected(b,c).
connected(c,d).
```

Or

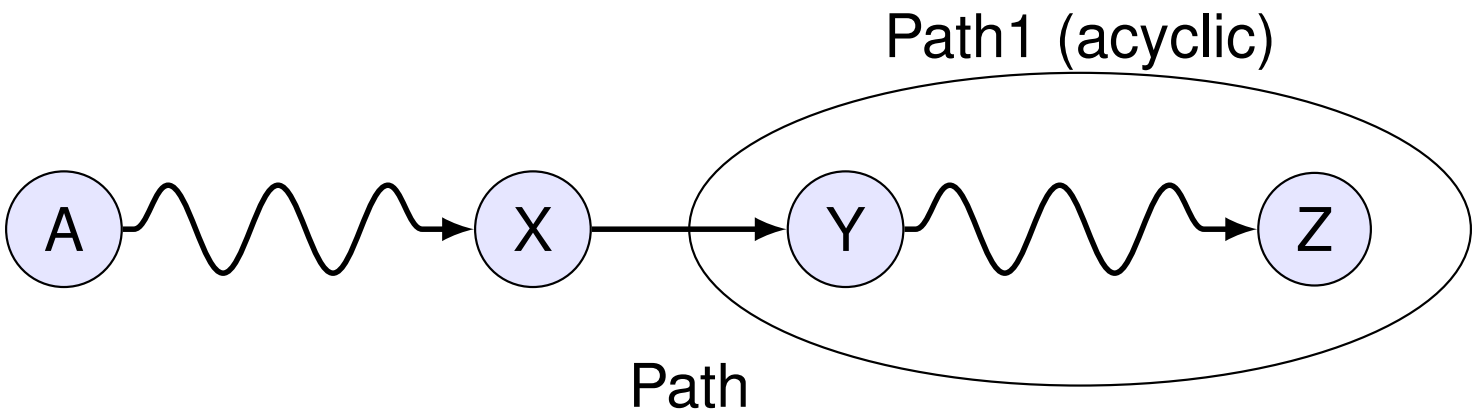
```
G = graph([a,b,c,d],
          [e(a,b),e(b,d),e(b,c),e(c,d)]).
```

Finding an acyclic path

- P is acyclic path between A and Z in graph G:

  `path(A,Z,G,P)`

  e.g. `path(a,d,G,[a,b,d])`, `path(a,d,G,[a,b,c,d])`.
- Since acyclic (i.e. contains no cycle), a node can appear in the path at most once
- A possible solution:
  - if `A=Z`, then `P=[A]`, otherwise:
  - find an acyclic path `P1` from some node `Y` to `Z`, and find a path from `A` to `Y` avoiding the nodes in `P1`



Define an auxiliary predicate: `path1(A,Path1,G,Path)`

- Boundary case: start node of `Path1` coincides with start node of `Path`, i.e. `A`
- General case: if it does not coincide, extend `Path1` by finding a node `X`, s.t.
  - `X` is adjacent to `Y`
  - `X` is not yet in `Path1`

  until start node of `Path` (i.e. `A`) is reached.

```
% path( A, Z, Graph, Path)
% Path is an acyclic path from A to Z in Graph

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),
 \+ member( X, Path1),     % No-cycle condition
 path1( A, [X, Y | Path1], Graph, Path).
```

- If graph is represented as structure, e.g.

```
G = graph([a,b,c,d],
          [e(a,b),e(b,d),e(b,c),e(c,d)])
```

then:

```
adjacent(X,Y,graph(Nodes,Edges)) :-
  member(e(X,Y),Edges).
adjacent(X,Y,graph(Nodes,Edges)) :-
  member(e(Y,X),Edges).
```

- Find Hamiltonian path!