UCS1524 – Logic Programming

Tree

Session Meta Data

Author	**Dr. D. Thenmozhi**
Reviewer	
Version Number	1.2
Release Date	10 September 2022

Session Objectives

- Understanding tree representation and operations on trees in Prolog.
- Learn about binary tree, binary dictionary with insert, delete and display operations.

Session Outcomes

- At the end of this session, participants will be able to
  - Explain the representation and operations on tree in Prolog.

Agenda

- Representation of tree
  - Binary tree
  - Binary dictionary
- Operations
  - Create
  - Search
  - Insert
  - Delete
  - Display

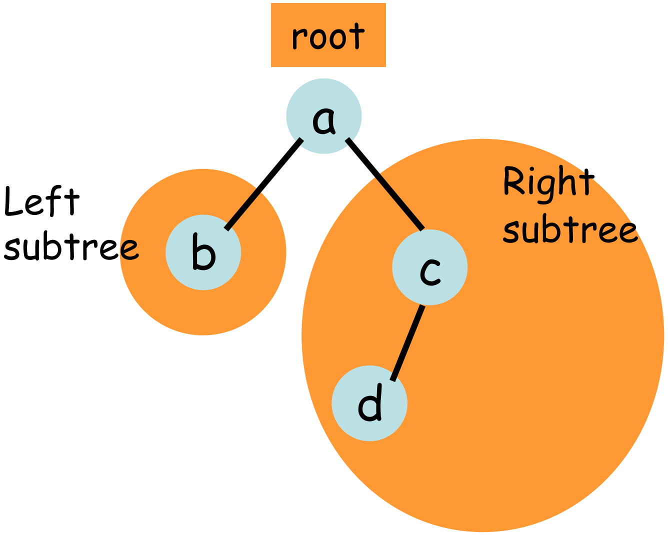
Representing sets by binary trees

- A disadvantage of using a list for representing a set is that the set membership testing is relatively inefficient.
- Using the predicate **member(X, L)** to find **X** in a list **L** is very inefficient because this procedure scans the list element by element until **X** is found or the end of the list is encountered.
- For representing sets, there are various tree structures that facilitate more efficient implementation of the set membership relation.
- We will here consider binary trees.

Representing sets by binary trees

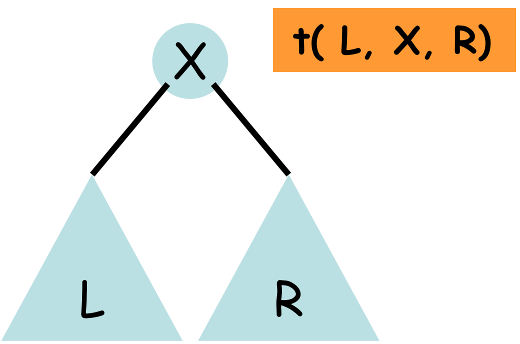
- A binary tree is either empty or it consists of three things:
  - A root;
  - A left subtree;
  - A right subtree.

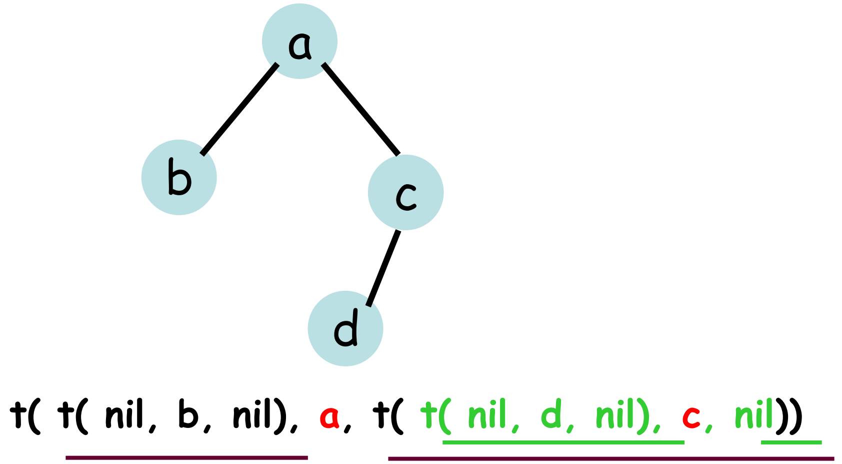
  The root can be anything, but the subtrees have to be binary tree again.



Representing sets by binary trees

- The representation of a binary tree:
  - Let the atom **nil** represent the empty tree.
  - Let the functor be **t** so the tree that has a root **X**, a left subtree **L**, and a right subtree **R** is represented by the term **t(L, X, R).**





Representing sets by binary trees

- Let use consider the set membership relation **in**. A goal **in(X, T)**
  is true if **X** is a node in a tree **T**.

- **X** is in tree **T** if
  - The root of T is X, or
  - X is in the left subtree of T, or
  - X is in the right subtree of T.

```
in( X, t( _, X, _)).
in( X, t( L, _, _)) :- in( X, L).
in( X, t( _, _, R)) :- in( X, R).
```

  - The goal **in(X, nil)** will fail for any X.

Representing sets by binary trees

```
| ?- T = t( t( nil, b, nil), a, t( t( nil, d, nil), c, nil)), in( X, T).
T = t(t(nil,b,nil),a,t(t(nil,d,nil),c,nil))
X = a ? ;
T = t(t(nil,b,nil),a,t(t(nil,d,nil),c,nil))
X = b ? ;
T = t(t(nil,b,nil),a,t(t(nil,d,nil),c,nil))
X = c ? ;
T = t(t(nil,b,nil),a,t(t(nil,d,nil),c,nil))
X = d ? ;
(15 ms) no
```

To print the elements of a tree

```
| ?- T = t( t( nil, b, nil), a, t( t( nil, d, nil), c, nil)), in(a, T).
T = t(t(nil,b,nil),a,t(t(nil,d,nil),c,nil)) ?
(16 ms) yes
```

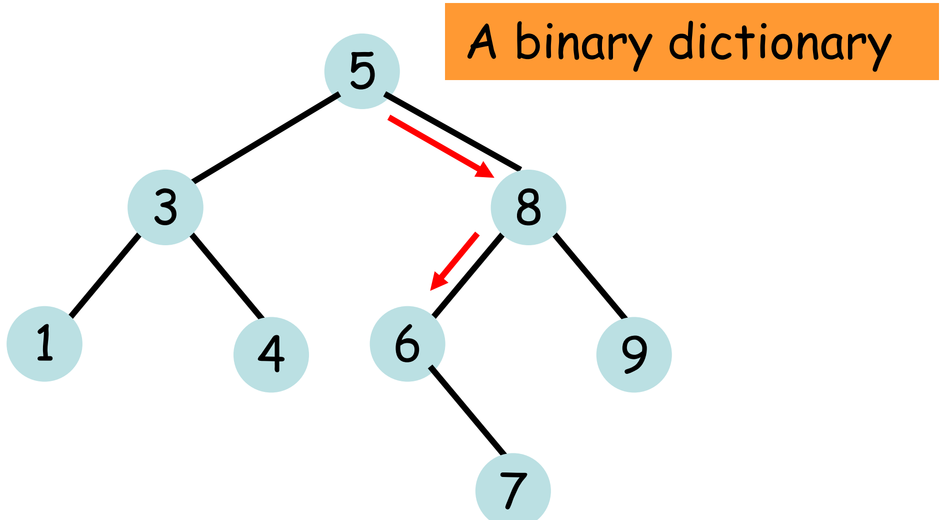
To search for an elements in a tree

```
| ?- T = t( t( nil, b, nil), a, t( t( nil, d, nil), c, nil)), in(e, T).
no
```

Representing sets by binary trees

- The above representation is also inefficient.
- After several recursive calls with fails and backtracking the element is found in the tree
- We can improve it by using a binary dictionary. (a binary search tree)
- In binary dictionary, the data in the tree can be ordered from left to right according to an ordering relation.
- A non-empty tree **t(Left, X, Right)** is ordered from left to right if:
  - all the nodes in the left subtree, Left, are less than X, and
  - all the nodes in the right subtree, Right, are greater than X; and
  - both subtrees are also ordered.
- The advantage of ordering: to search for an object in a binary dictionary, it is always sufficient to search at most one subtree.

Representing sets by binary trees



- To find an item X in a dictionary D:
  - if X is the root of D then X has been found, otherwise
  - if X is less than the root of D then search for X in the left subtree of D, otherwise
  - search for X in the right subtree of D;
  - if D is empty the search fails.

Representing sets by binary trees

% Figure 9.7 Finding an item X in a binary dictionary.

```
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).
```

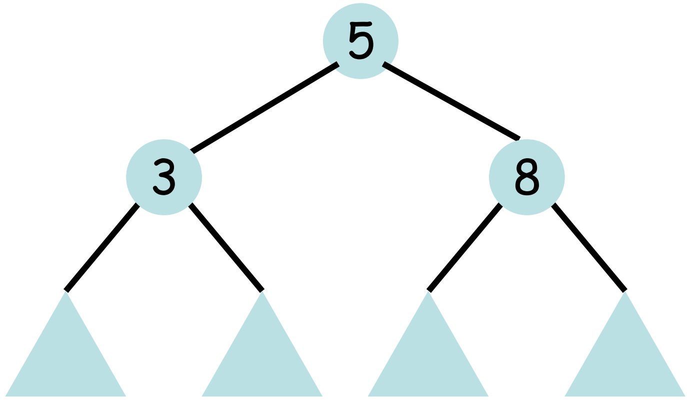
- The relation **gt(X, Y)** means X is greater than Y.
- The **in** procedure itself can be also used for constructing a binary dictionary. For example:

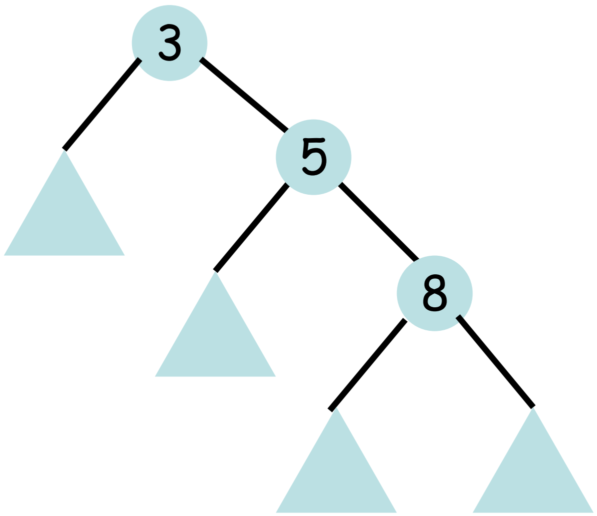
```
| ?- in( 5, D), in( 3, D), in( 8, D).
D = t(t(_,3,_),5,t(_,8,_)) ?
(16 ms) yes
```

Representing sets by binary trees

```
| ?- in( 5, D), in(3, D), in(8, D).
D = t(t(_,3,_),5,t(_,8,_)) ?
(16 ms) yes
```

```
| ?- in( 3, D), in(5, D), in(8, D).
D = t(_,3,t(_,5,t(_,8,_))) ?
yes
```





Representing sets by binary trees

- A tree is (approximately) balanced if, for each node in the tree, its two subtrees accommodate an approximately equal number of items.
- If a dictionary with n nodes is nicely balanced then its height is proportional to $\log n$.
- If the tree gets out of balance its performance will degrade.
- In extreme cases of totally unbalanced trees, a tree is reduced to a list. In such a case the tree's height is n, and the tree's performance is equally poor as that of a list.

Insertion and deletion in a binary trees

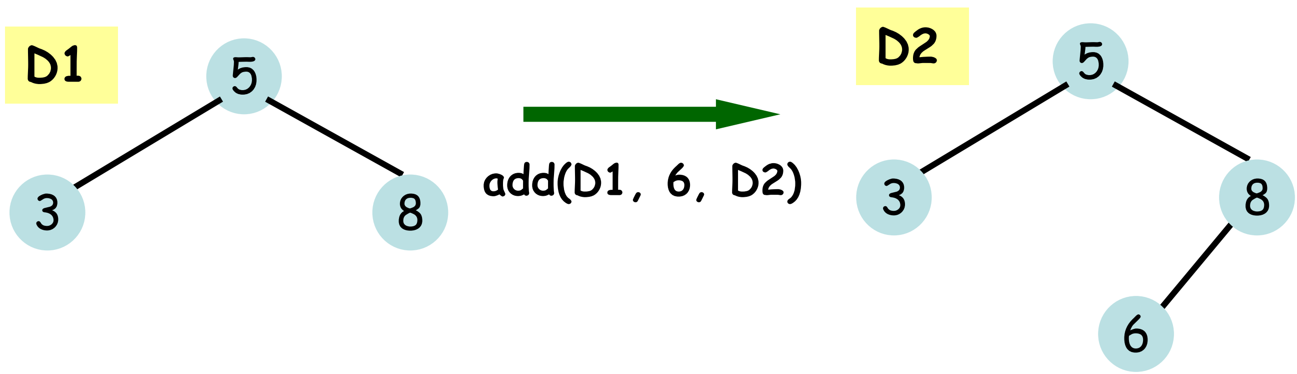
- When maintaining a dynamic set of data we may insert new items into the set and delete some old items from the set.

in(X, S) **X is a member of S**

add(S, X, S1) **Add X to S giving S1**

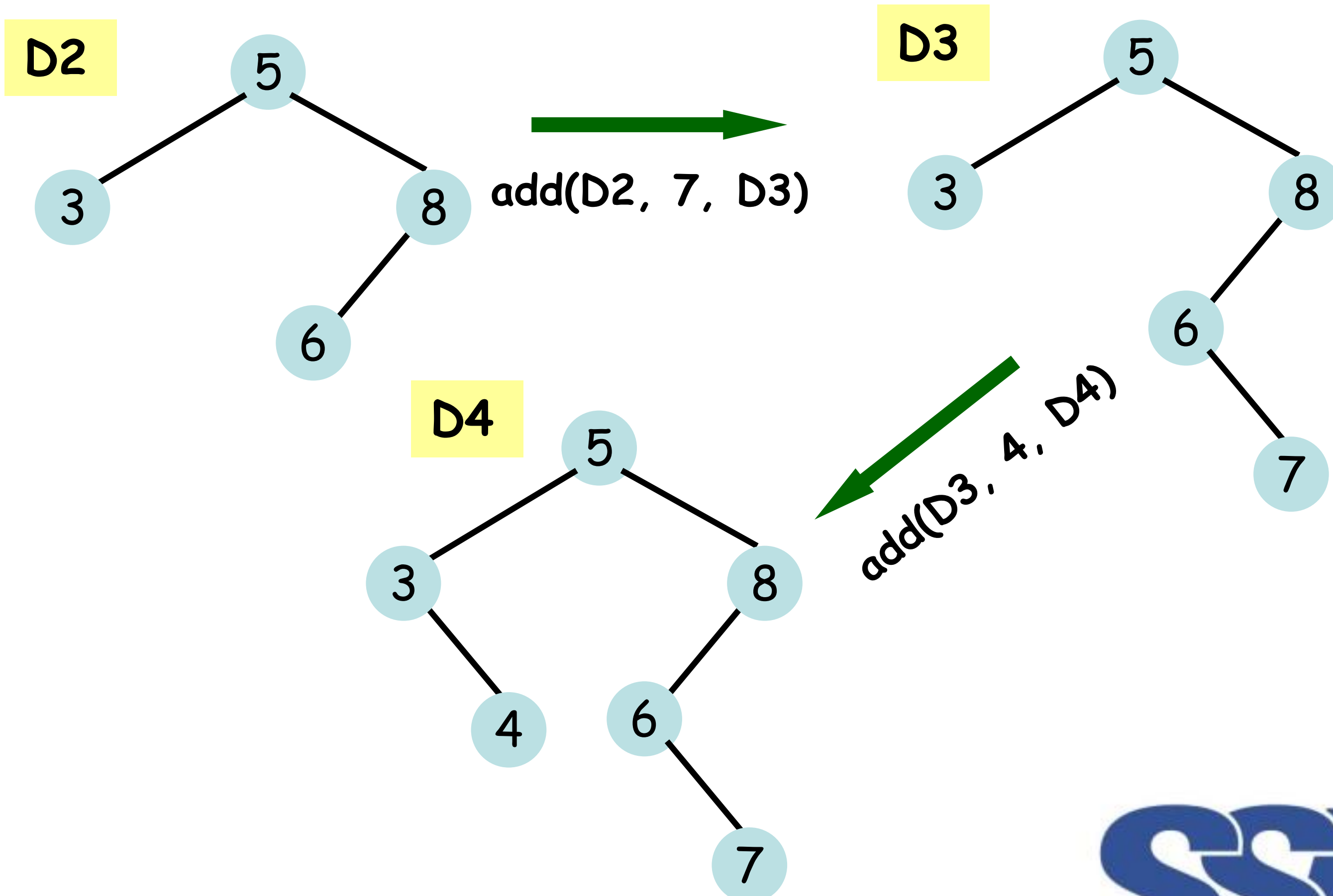
del(S, X, S1) **Delete X from S giving S1**

- “**add**” relation: Insert nodes into a binary dictionary at the leaf level



Insertion and deletion in a binary trees

- “**add**” relation: Insert nodes into a binary dictionary at the leaf level



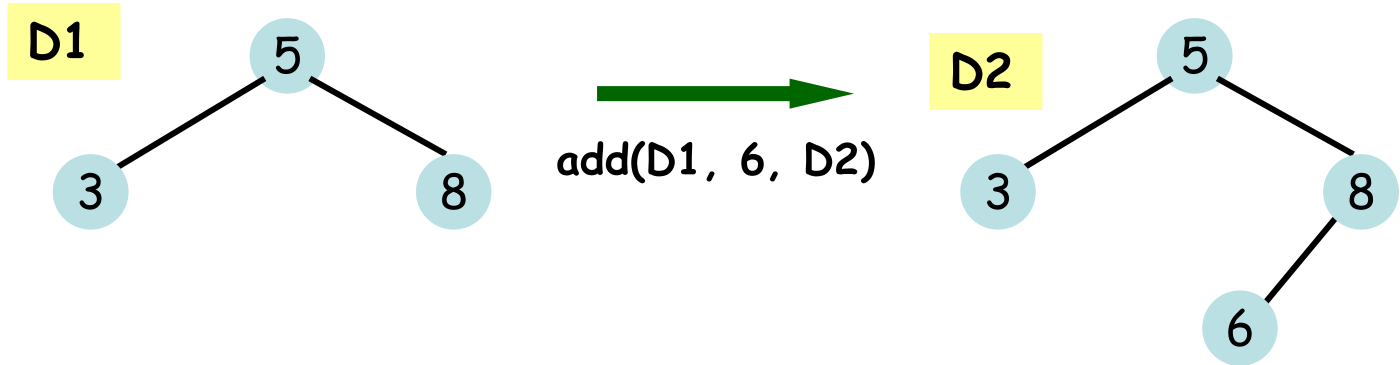
Insertion and deletion in a binary trees

- Let us call this kind of insertion **addleaf(D, X, D1).**
- Rules for adding at the leaf level are:
  - The result of adding **X** to the empty tree is the tree **t(nil, X, nil).**
  - If **X** is the root of **D** then **D1** = **D** (no duplicate item gets inserted).
  - 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.

Insertion and deletion in a binary trees

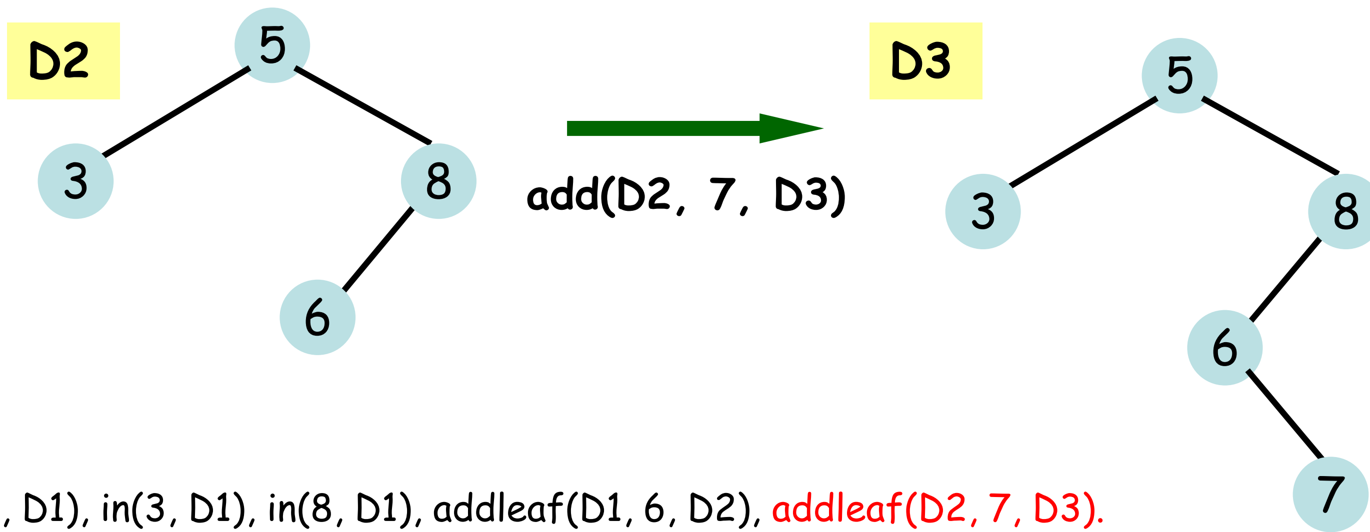
```
% Figure 9.10  Inserting an item as a leaf into the binary dictionary.

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 a binary trees



| ?- in(5, D1), in(3, D1), in(8, D1), addleaf(D1, 6, D2).
D1 = t(t(A,3,B),5,t(nil,8,C))
D2 = t(t(A,3,B),5,t(t(nil,6,nil),8,C)) ?
yes

Insertion and deletion in a binary trees

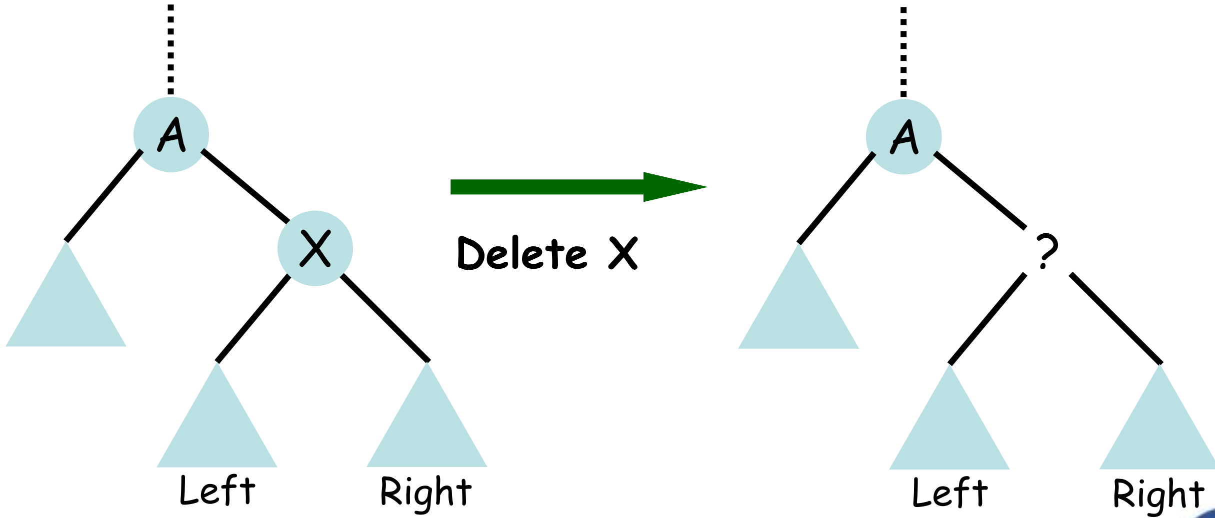



```
| ?- in(5, D1), in(3, D1), in(8, D1), addleaf(D1, 6, D2), addleaf(D2, 7, D3).
D1 = t(t(A,3,B),5,t(nil,8,C))
D2 = t(t(A,3,B),5,t(t(nil,6,nil),8,C))
D3 = t(t(A,3,B),5,t(t(nil,6,t(nil,7,nil)),8,C)) ?
yes
| ?- in(5, D1), in(3, D1), in(8, D1), addleaf(D1, 6, D2), addleaf(D2, 7, D3),
     addleaf(D3, 4, D4).
D1 = t(t(A,3,nil),5,t(nil,8,B))
D2 = t(t(A,3,nil),5,t(t(nil,6,nil),8,B))
D3 = t(t(A,3,nil),5,t(t(nil,6,t(nil,7,nil)),8,B))
D4 = t(t(A,3,t(nil,4,nil)),5,t(t(nil,6,t(nil,7,nil)),8,B)) ?
yes
```

Insertion and deletion in a binary trees

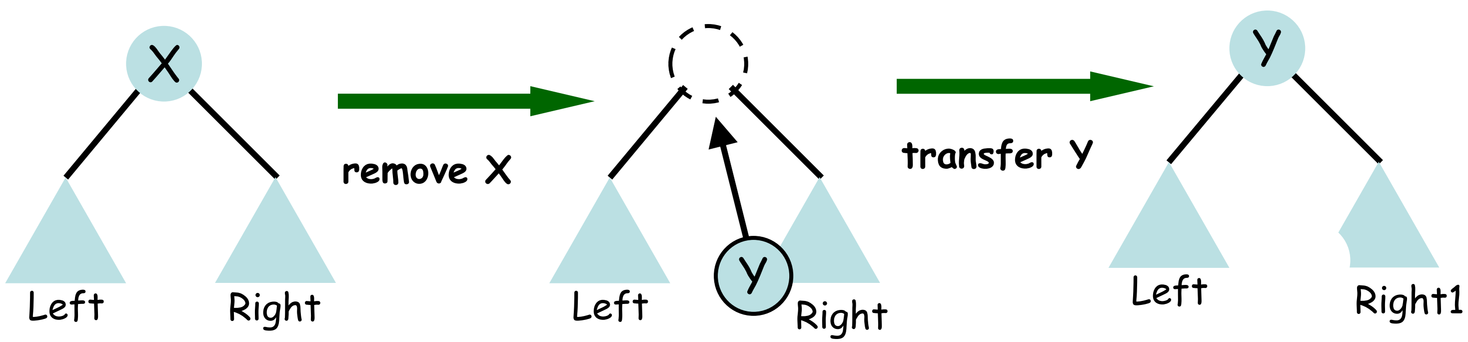
- Consider "**delete**" operation:
  - It is easy to delete a leaf, but deleting an internal node is more complicated.
  - The deletion of a leaf can be in fact defined as the inverse operation of inserting at the leaf level:

    delleaf(D1, X, D2) :- addleaf(D2, X, D1)
  - What happens if X is an internal node?



Insertion and deletion in a binary trees

- Solutions:
  - If one of the subtrees **Left** and **Right** is empty then the solution is simple: the non-empty subtree is connected to A.
  - If they are both non-empty then the left-most node of **Right**, Y, is transferred from its current position upwards to fill the gap after X.



Insertion and deletion in a binary trees

```
% Figure 9.13  Deleting from the 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, R), Y, R).
delmin( t( Left, Root, Right), Y, t( Left1, Root, Right))  :-
      delmin( Left, Y, Left1).
```

X

L R

Insertion and deletion in a binary trees

- Another elegant solution of add:
  - The add relation can be defined non-deterministically so that a new item is inserted at any level of the tree, not just at the leaf level.

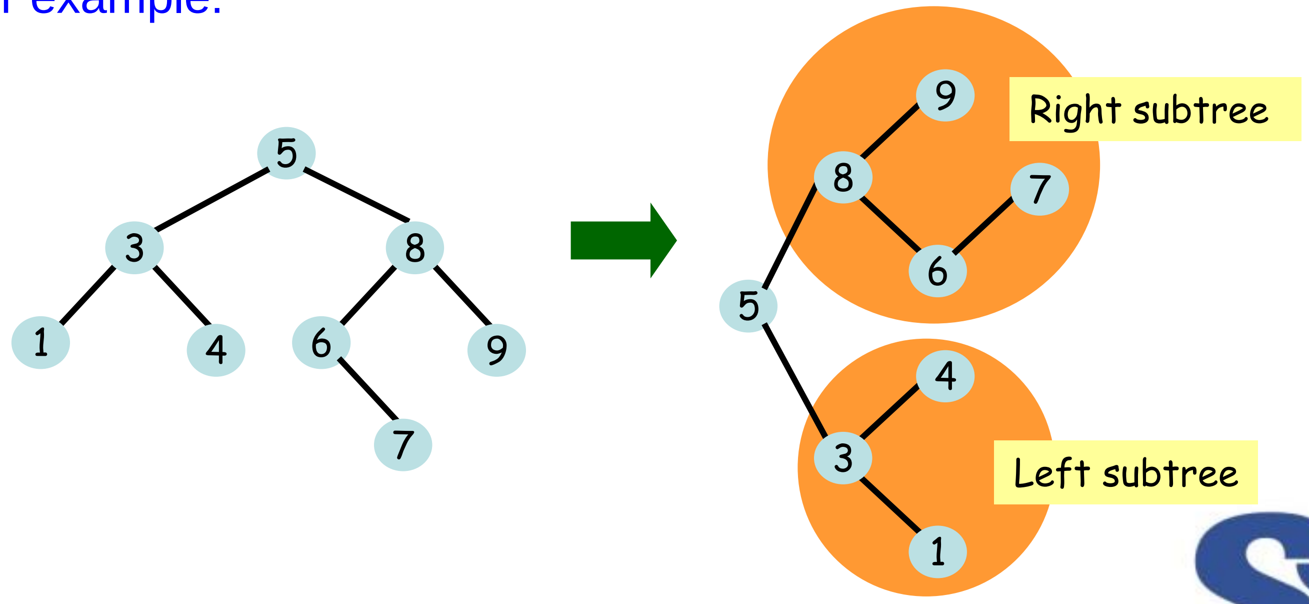
  - To add **X** to a binary dictionary **D** either:
    - Add **X** at the root of **D** (so that **X** becomes the new root), or
    - If the root of **D** is greater than **X** than insert **X** into the left subtree of **D**, otherwise insert **X** into the right subtree of **D**.

Display trees

- The goal

  write(T)

  will output all the information of a binary tree T, but will not graphically indicate the actual tree structure.
- There is a simple method for displaying trees in graphical forms. The trick is to display a tree growing from left to right, and not from top to bottom as trees are usually pictured.
- For example:



Display trees

- The procedure

  show(T)

  will display a tree T in the graphical form.
- To show a non-empty tree, T:

  (1) show the right subtree of T, indicated by some distance, H, to the right;

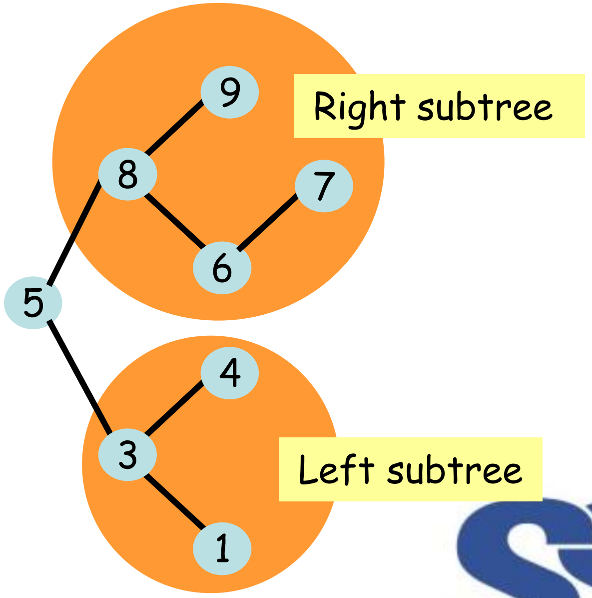
  (2) write the root of T;

  (3) show the left subtree of T indented by distance H to the right.

  – The indentation distance H is an additional parameter for displaying trees.

  show2(T, H)

  displays T indented H spaces from the left margin.



Display trees

```
% Displaying a binary tree.

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

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

Summary

- Representation of tree
  - Binary tree
  - Binary dictionary - BST
- Operations
  - Create
  - Search
  - Insert - add at the leaf
  - Delete
  - Display