Logic Programming

Rodica Potolea
Camelia Lemnaru

Lecture #7CS@TUCN

Agenda

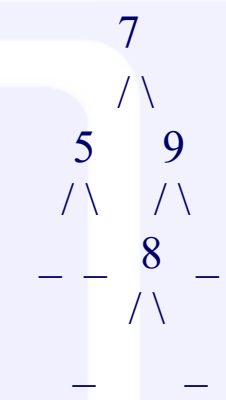
- Incomplete structures
  - Review – meaning, usage, behavior
  - Incomplete Trees
- Difference lists

Incomplete structures - review

- Incomplete structures (lists, trees)
  - Replace the empty structure at the end
  - with a variable
- Used as I/O structure (saves time and space)
- Change the behavior of regular predicates
- *Failure* conditions NEED to be:
  - ALL Explicit (no default failure in the absence of the clause)
  - Come FIRST (otherwise, is as if it does not exist)
- *Success* conditions should be:
  - All explicit (as before)
  - Start with the condition for the variable at the end which comes right after the failure condition(s)

Search in BST

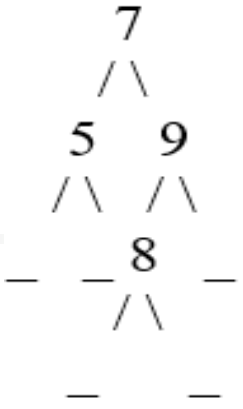
This is an Incomplete BST
(ended in variable)




```
tree(t(7, t(5,_,_),t(9,t(8,_,_),_))).
search(Key,t(Key,_,_)):-!,
       write("found").
search(Key, t(Key1,Left,_)):-
      Key<Key1,!,
      search(Key,Left).
search(Key,t(_,_,Right)):-
      search(Key,Right).
```

- The search predicate Works well for a **regular** BST
- What behavior would it have in case of an incomplete tree?

Search in BST



- It has **dual** significance in the search process:
  - if `Key` **present**, reports so
  - else **inserts** it as leaf

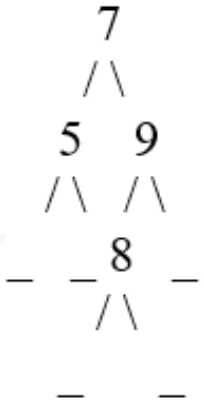
```
tree(t(7, t(5,_,_),t(9,t(8,_,_),_))).
search(Key,t(Key,_,_)):-!,
        write("found").
search(Key, t(Key1,Left,_)):-
       Key<Key1,!,
       search(Key,Left).
search(Key,t(_,_,Right)):-
       search(Key,Right).

q1: ?-tree(T),search_ins(9,T).
q2: ?-tree(T),search_ins(6,T).
```

q1 behaves as search (= **`search`**`_ins`)

q2 behaves as insert (= `search_`**`ins`**)

Search in BST




```
q1: ?-tree(T),search_ins(9,T).
search_ins(Key,t(Key,_,_)):-!,
        write("found OR inserted").
```

q1 behaves as search (= **search**_ins)

The meaning of the clause is:

```
search_ins(Key,T):-
                nonvar(T),T=t(Key1,_,_),Key1=Key,!,
               write("found OR inserted").
```

```
q2: ?-tree(T),search_ins(6,T).
search_ins(Key,t(Key,_,_)):-!,
        write("found OR inserted").
```

q2 behaves as search (= search_**ins**)

The meaning of the clause is:

```
search_ins(Key,T):-
               var(T),!,T=t(Key,_,_),
               write("found OR inserted").
```

PURE Search in BST (search based on Key)

```
search_IL(Key,T):-
      var(T),!,
      fail.
search_IL(Key,t(Key,_,_)):-!,
       write("found").
search_IL(Key, t(Key1,Left,_)):-
      Key<Key1,!,
      search(Key,Left).
search_IL(Key,t(_,_,Right)):-
      search(Key,Right).
```

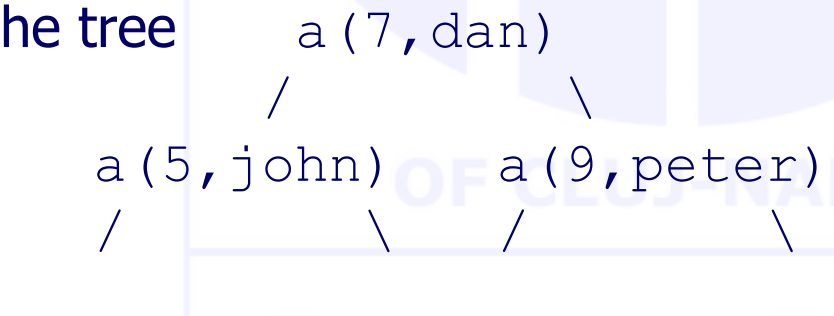
Search in BST

Assume the composite structure of type `a(key,value)`

The incomplete BST

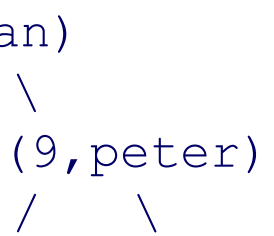
```
Ex: tree(n(
              (n(_,a(5,john),_)
              a(7,dan),
              n(_,a(9,peter),_)
              )
            ).
```

Represents the tree



Search in BST (search based on Key)

```
search_ins(Key,n(_,a(Key,Info),_),Info):-!.
search_ins(Key,n(Left,a(Key1,_),_),Info):-
      Key<Key1,!
      search_ins(Key,Left,Info).
search_ins(Key,n(_,_,Right),Info):-
      search_ins (Key,Right).
q1: ?- tree(T),search_ins(9,T,R).
Yes, R=peter
```

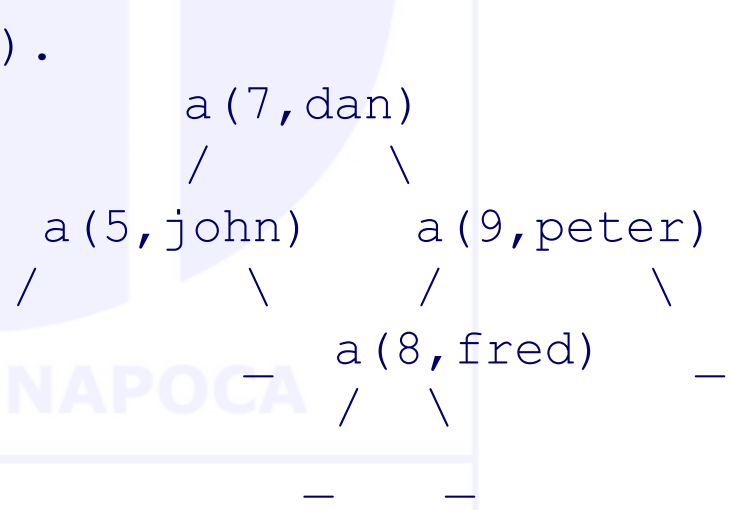


Search in BST (search based on Key)

```
search_ins(Key,n(_,a(Key,Info),_),Info):-!.
search_ins(Key,n(Left,a(Key1,_),_),Info):-
       Key<Key1,!
       search_ins(Key,Left,Info).
search_ins(Key,n(_,_,Right),Info):-
       search_ins (Key,Right).
```

q2: ?- tree(T),search_**ins**(8,T,fred).

Yes, T increases



q3: ?- tree(T),search_ins(5,T,maria).

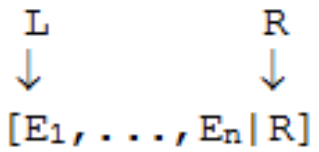
Does it fail?

If Yes, How?

If Not, Why? How is the tree affected?

Difference Lists

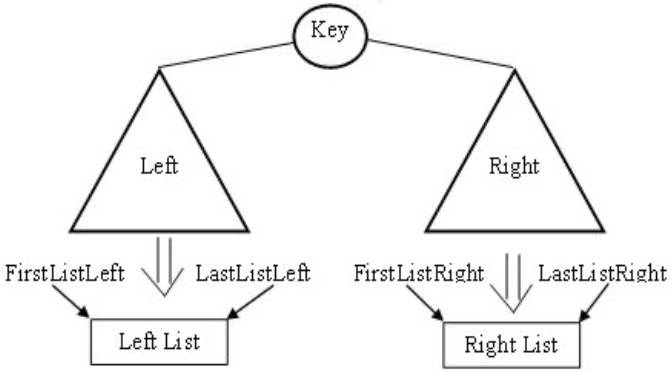
- Another type of "irregular" structures
- Allow access to both beginning and END of lists ☺
- Named as lists with First and Last element (that is, besides the variable ="pointer" at the beginning of the structure, we have a variable to "point" at the end of the list). Similar to imperative languages where we have lists with pointers in the beginning and end of the list
- Allow for increased performance
- Difference list:

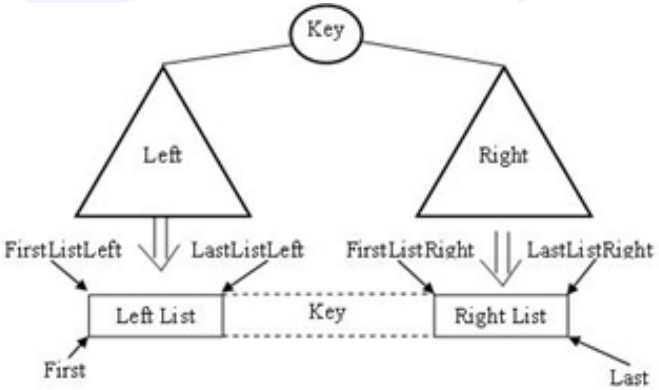


- D, denoted as L\R (sometimes, in code, it is preferred to be represented as 2 arguments, L and R) is:
  - `D=L\R=[E1,..,En]`
  - L "points" at the beginning, R at the "end"

Tree to list (remember solutions in Lecture 5)

```
%generate_ordered_list/3
%generate_ordered_list(tree, begin_diff_list, end_diff_list)
generate_ordered_list(nil,First,Last):-First=Last.
generate_ordered_list (t(Left,Key,Right),First,Last):-
        generate_ordered_list(Left,FirstListLeft,LastListLeft),
        generate_ordered_list(Right,FirstListRight,LastListRight),
        First= FirstListLeft,
        LastListLeft=[Key|FirstListRight],
        Last=  LastListRight.
```





Tree 2 list forward – code (Lecture #5)

% sort_1/3
% sort_1(in_tree,partial_result_list,final_result_list)

```
sort_1(nil,L,L).
sort_1(n(Left,Key,Right),Lin,Lout):-
     sort_1(Left,Lin,Llout),
     append(Llout,[Key],Lrin),
     sort_1(Right,Lrin,Lout).

sort_1(Tree,Result):-
     sort_1(Tree,[],Result).
```

Tree to list (same solution – check Lecture #5)

- Needs specialized call

```
generate_ordered_list(ITree,OList):-
    generate_ordered_list(ITree,OList,[]).
```

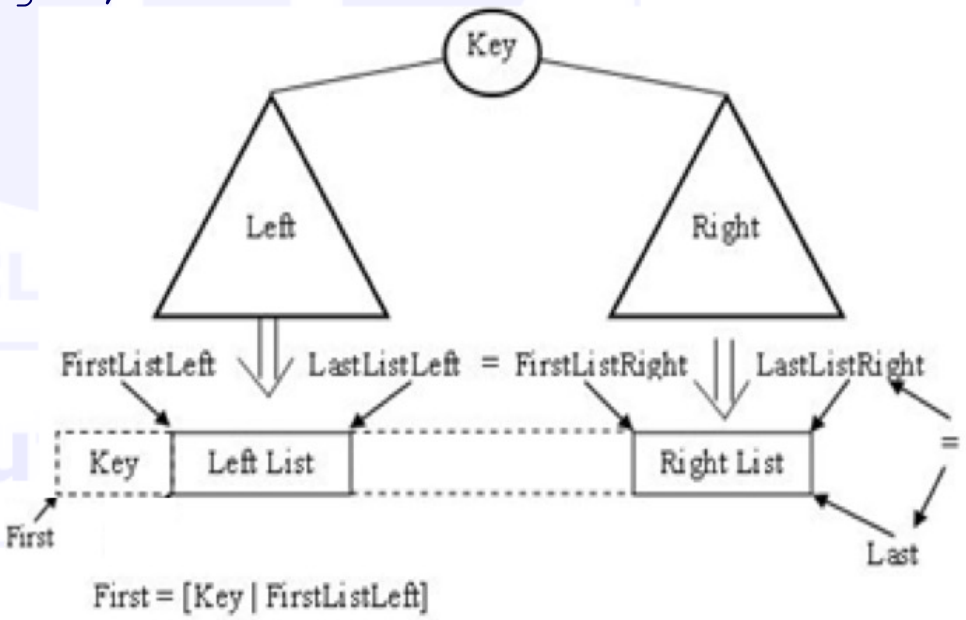
- Replace explicit unifications with implicit ones.

```
generate_ordered_list(nil,L,L).
generate_ordered_list(t(Left,Key,Right),FirstListLeft,LastListRight):-
      generate_ordered_list(Left,FirstListLeft,[Key|Intermediate]),
      generate_ordered_list(Right,Intermediate,LastListRight).
```

- Same as before, yet the utilization of the same name for variables (`FirstListLeft`) instead of making explicit unifications (`First=FirstListLeft` and the other 2).

Pre-order

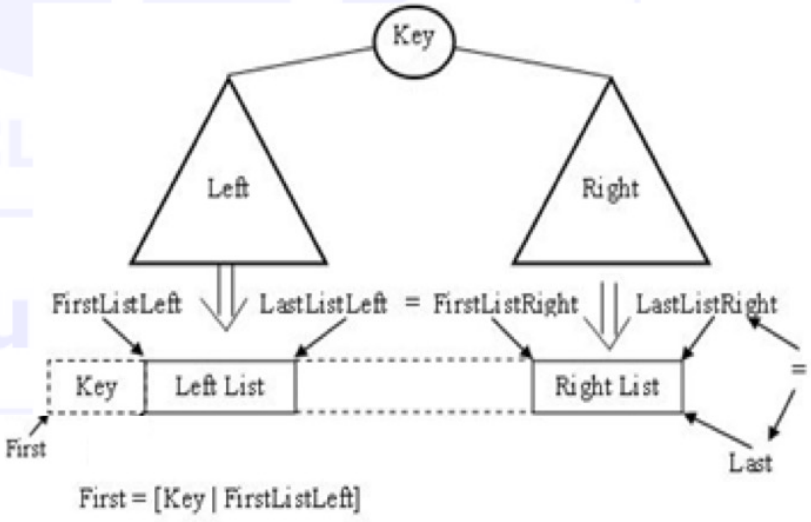
```
generate_preorder_list(nil, First,Last):-First=Last.
generate_preorder_list(t(Left,Key,Right),First,Last):-
        generate_preorder_list(Left,FirstListLeft,LastListLeft),
        generate_preorder_list(Right,FirstListRight,LastListRight),
        First= [Key|FirstListLeft],
        LastListLeft=FirstListRight,
        Last=  LastListRight.
```



Pre-order

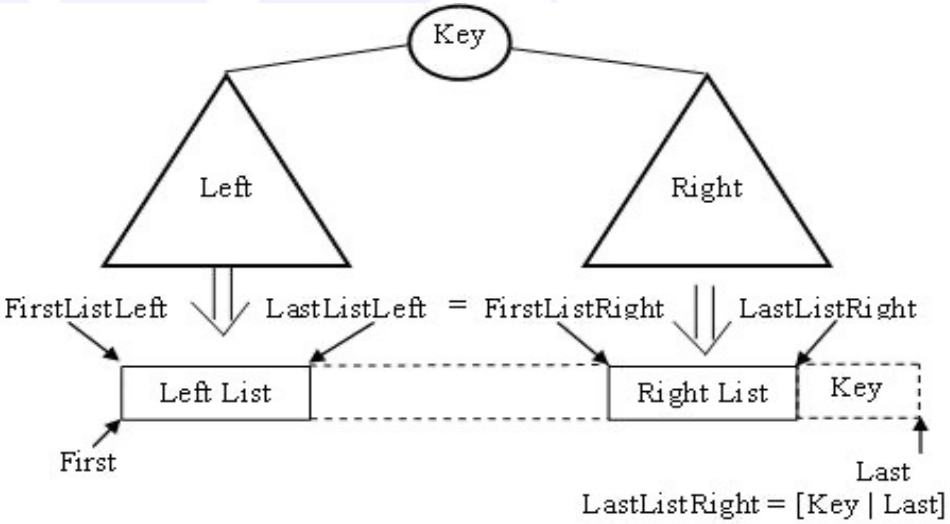
```
generate_preorder_list(nil,L,L).
generate_preorder_list(t(Left,Key,Right),[Key|FirstListLeft],LastListRight):
        generate_preorder_list(Left,FirstListLeft,Interm),
        generate_preorder_list(Right,Interm,LastListRight).

generate_preorder_list(ITree,OList):-
        generate_preorder_list(ITree,OList,[]).
```



Post-order

```
generate_postorder_list(nil, First,Last):-First=Last.
generate_postorder_list(t(Left,Key,Right),First,Last):-
        generate_postorder_list(Left,FirstListLeft,LastListLeft),
        generate_postorder_list(Right,FirstListRight,LastListRight),
        First= FirstListLeft,
        LastListLeft=FirstListRight,
        LastListRight=[Key|Last].
```



Post-order

```
generate_postorder_list(nil,L,L).
generate_postorder_list(t(Left,Key,Right),FirstListLeft,LastListRight):-
        generate_postorder_list(Left,FirstListLeft,Interm),
        generate_postorder_list(Right,Interm,[Key|LastListRight]).

generate_postorder_list(ITree,OList):-
        generate_postorder_list(ITree,OList,[]).
```