Logic Programming

Rodica Potolea
Camelia Lemnaru

Lecture #6, 2021

Cluj-Napoca

Agenda

- Trees (BST and regular trees)
  - insert
  - search
  - delete
- Incomplete structures
  - meaning/utility
  - lists
  - Trees – next lecture

Insert in BST

```
%insert_bst/3
%insert_bst(in_tree, key_2_ins,out_tree).
```

Q: is the order of arguments relevant ? Why/why not?

```
insert_bst(nil, Key,t(nil,Key,nil)):-!.
% insert_bst(Tin, Key,Rez):-Tin=nil,!,Rez= t(nil,Key,nil).
insert_bst(t(Left,Key,Right), Key, t(Left,Key,Right)):-!,
      write("already in tree").
insert_bst(t(Left,Key1,Right),Key,t(NewLeft,Key1,Right)):-
      Key<Key1,!,
      insert_bst(Left,Key,NewLeft).
insert_bst(t(Left,Key1,Right),Key,t(Left,Key1,NewRight)):-
      insert_bst(Right,Key,NewRigth).
```

Qs: **1.** Where is inserted (position)? ALWAYS LEAF! NO EXCEPTION!

2. What does ! in clause 1 negate? In clause 2?

Time estimate: a=1, b=2 (if balanced), c=0 =>O(lgn)

Q: Estimate best/worst case (situation) and time

Insert in regular (not search) tree

```
%insert/3
%insert(in_tree, key_2_ins,out_tree).

insert(nil, Key,t(nil,Key,nil)):-!.
%insert(Tin, Key,Tout):-Tin=nil,!, Tout=t(nil,Key,nil).
insert(t(Left,Key,Right),Key,t(Left,Key,Right)):-!,
      write("already in tree").
%insert(Tin,Key,Tout):-Tin=t(Left,Key,Right),!,
      %write("already in tree"),Tout= t(Left,Key,Right)).
insert(t(Left,Key1,Right),Key,t(NewLeft,Key,Right)):-
      insert(Left, Key,NewLeft).
insert (t(Left,Key1,Right), Key,t(Left,Key,NewRight)):-
      insert(Right,Key,NewRigth).
```

Qs: 1. Where is inserted (position)? Specifically!

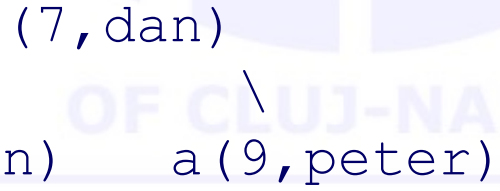
Oral discussion on OR nondeterminism for parallel/concurrent processing.

Search in BST

Assume the node contains a structure of type a(key,value)
The tree is BST based on key

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

Represents a tree with 7 and 2 leaves (7 and 9).



If stored in knowledge base, processing is with:

```
?-tree(T), process(T,…).
```

Search in BST (search based on Key)

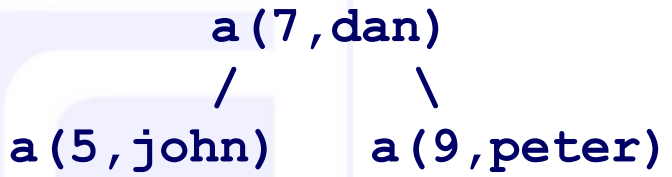
```
%is_in_stree/2
%is_in_stree(key_searched,tree)

is_in_stree(SN,n(_,Node,_):-
        eqch(SN,Node),!
        SN=Node.
is_in_stree(SN,n(Left,Node,_):-
        ord(SN,Node),!
        is_in_stree(SN,Left).
is_in_stree(SN,n(_,_,Right):-
        is_in_stree(SN,Right).

eqch(a(SK,_),(a(Key,_)):-
        nonvar(SK),
        SK=Key,!.
ord(a(SK,_),a(Key,_)):-
        nonvar(SK),
        SK<Key,!.
```

Search in BST (search by Key) – contd.

```
eqch(a(SK,_),(a(Key,_)):-
        nonvar(SK),
        SK=Key,!.
```



Predicate to check if the *key* we are *searching* for has the **same** value as the *key* in the *current node* in the tree.
Qs: what is the meaning of `nonvar(SK)` and why is needed? The ! Is not mandatory! Why?

```
ord(a(SK,_),a(Key,_)):-
        nonvar(SK),
        SK<Key,!.
```

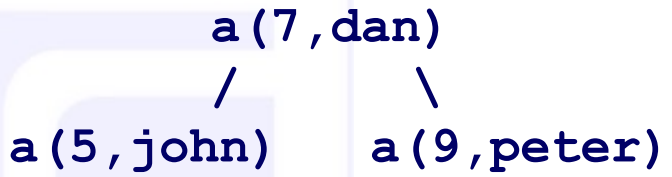
Predicate to check if the *key* we are *searching* for has a **smaller** value than the *key* in the *current node* in the tree.
Q: same questions.

```
?-tree(T),is_in_tree(a(9,X),T).
Yes,T=…, X=…?
```

Search in regular tree (search by Value)

```
%is_in_tree/2
%is_in_tree(key_searched,tree)

is_in_tree(SN,n(_,SN,_)).
is_in_tree(SN,n(Left,_,_):-
       is_in_tree(SN,Left).
is_in_tree(SN,n(_,_,Right):-
       is_in_tree(SN,Right).
```




```
?-tree(T),is_in_tree(a(R,john),T).
Yes, T=…,R=5.
```

%T would be the tree read, R the key assoc. to the value (name) provided

```
?-tree(T),is_in_tree(a(9,X),T).
X=…?
```

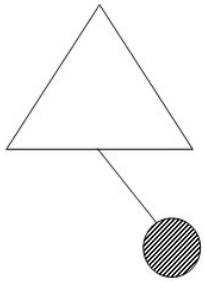
% can we ask this way? What's the answer? What's the meaning?

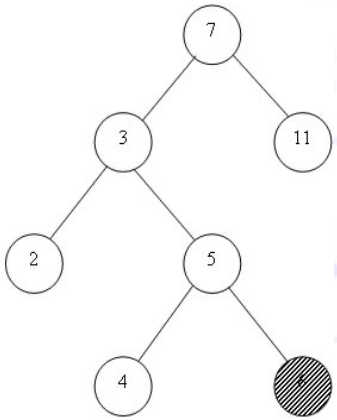
Delete from BST

- Search for the node where the key resides + remove the node
- Delete a key = remove the node containing the key
- Cases to consider:
  - Leaf – just remove the node
  - One child node – shortcut the node (link the child to its grandparent)
  - Two children (⇔ delete the root after the search stage) different story

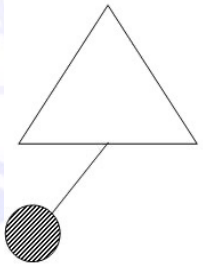
Figs. For leaves removals

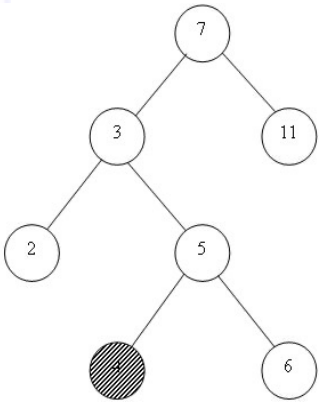
Case 1





Case 2

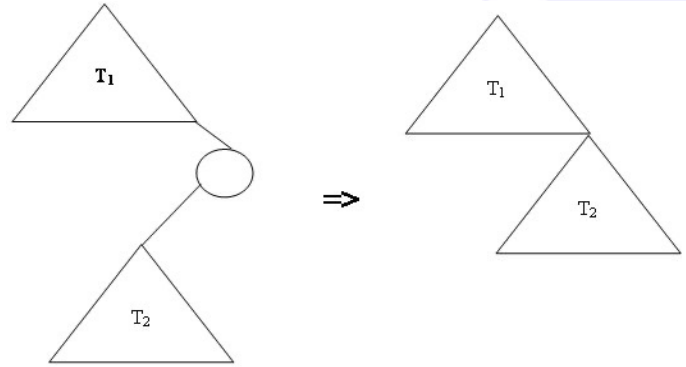


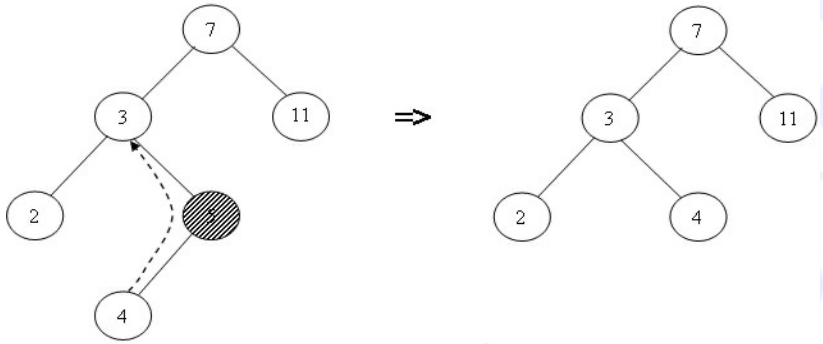


Delete from BST – contd.

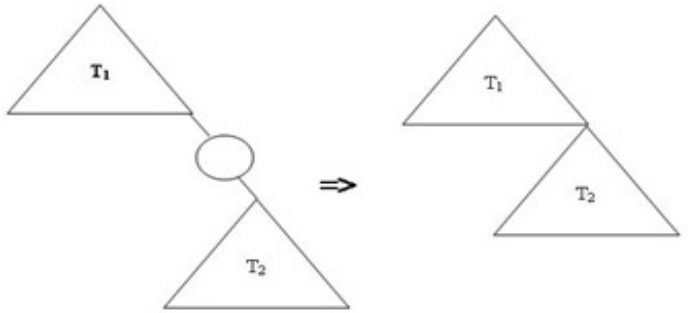
Removal of a right child node with just one subtree

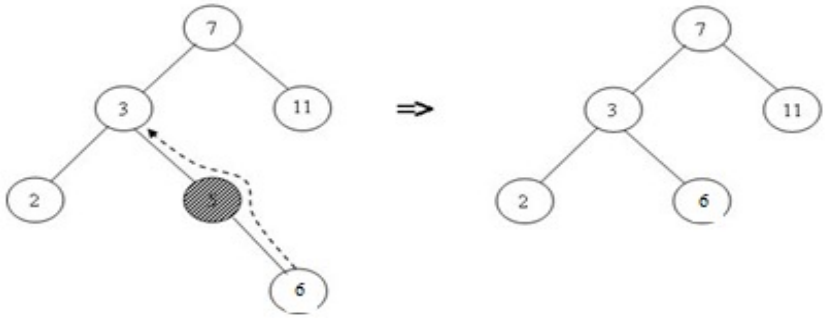
Case 3





Case 4

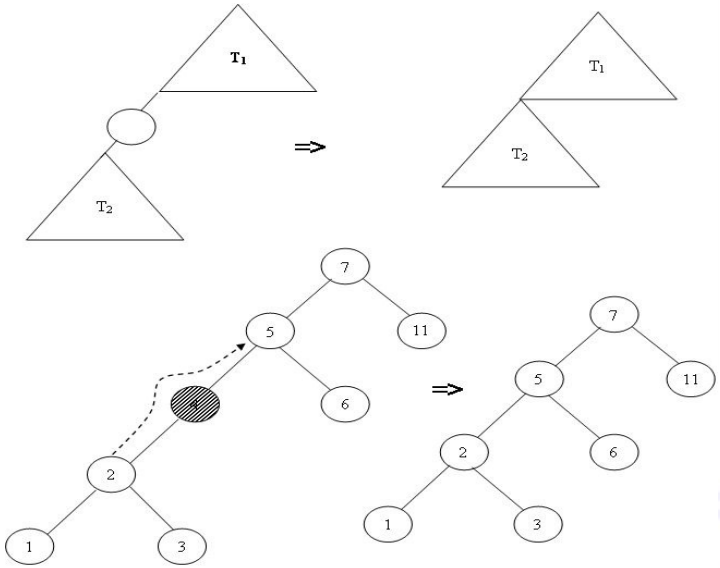




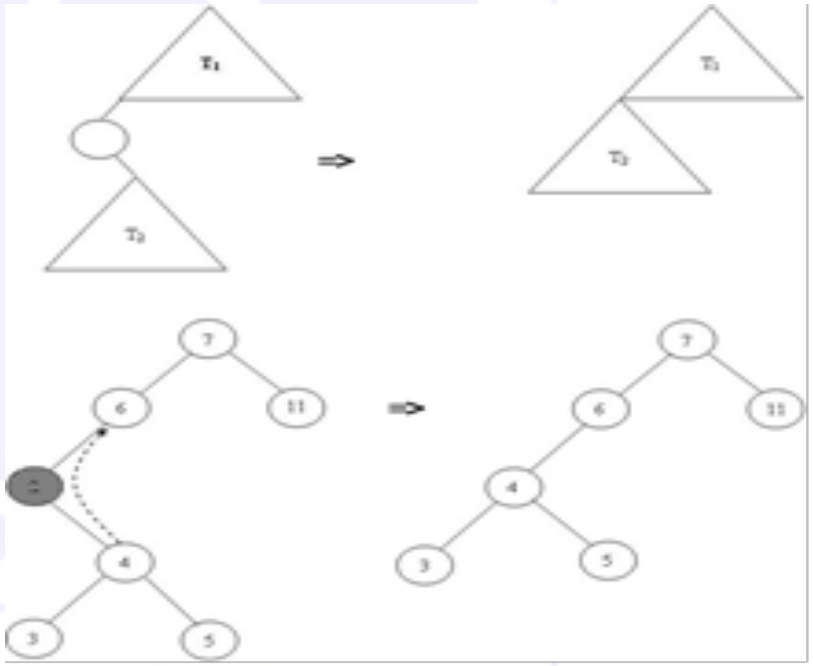
Delete from BST – contd.

Removal of a left child node with just one subtree

Case 5



Case 6



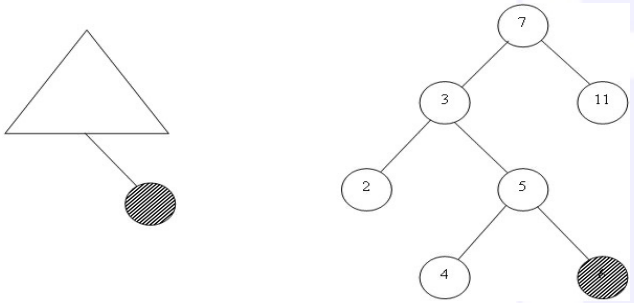
- How about root? Postpone for the moment.
- Let's reach this point with code.

Delete from BST - code

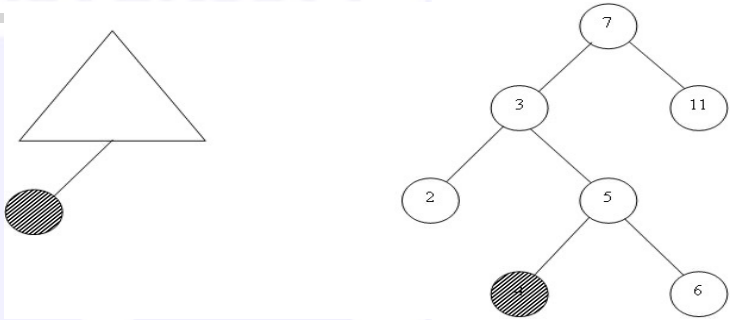
```
% delete_bst/3
% delete_bst(in_tree, key_to_del,out_tree)
% search key; find the node to contain the key
delete_bst(t(Left,Key1,Right),Key,t(NewLeft,Key1,Right)):-
      Key<Key1,!,
      delete_bst(Left,Key,NewLeft).
delete_bst(t(Left,Key1,Right),Key,t(Left,Key1,NewRight)):-
      Key>Key1,!,
      delete_bst(Right,Key,NewRight).
% found key; let's solve the easy cases discussed before
delete_bst(nil,Key,nil):-!,
            write(Key),write('not in tree').
delete_bst(t(nil,Key,nil),Key,nil):-!.
delete_bst(t(nil,Key,Right),Key,Right):-!.
delete_bst(t(Left,Key,nil),Key,Left):-!.
% "difficult" case postponed
```

Delete cases

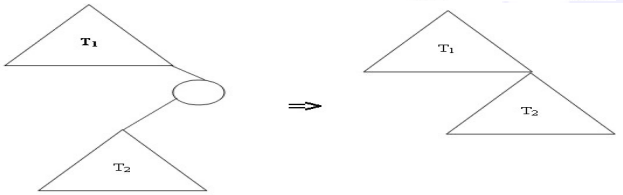
1



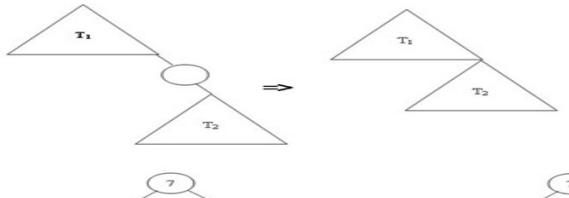
2



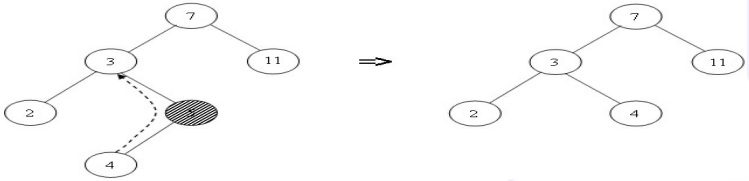
3



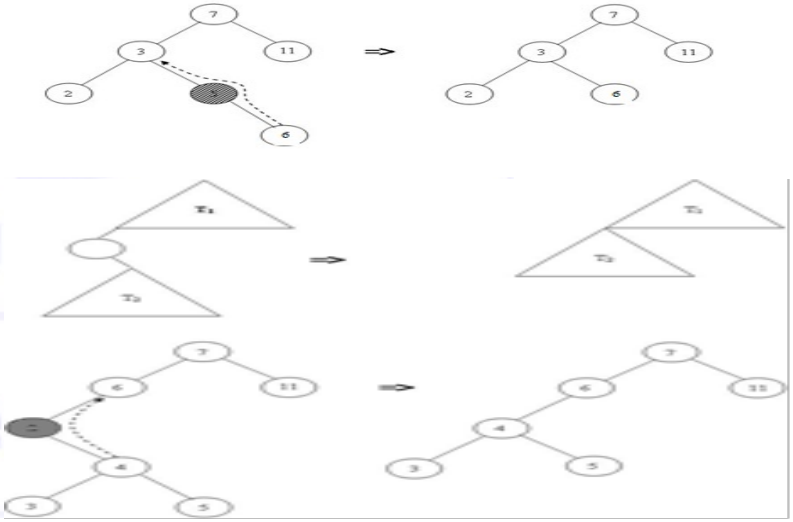
4

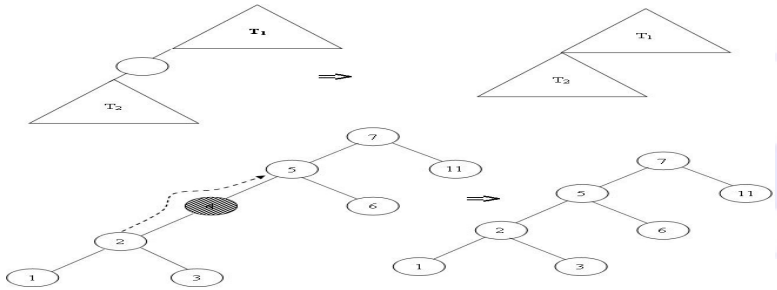


5



6





Delete from BST - code

```
% delete_bst/3
% delete_bst(in_tree, key_to_del,out_tree)
delete_bst(t(Left,Key1,Right), Key,t(NewLeft,Key1,Right)):-
      Key<Key1,!,
      delete_bst(Left, Key,NewLeft).
delete_bst(t(Left,Key1,Right), Key,t(Left,Key1,NewRight)):-
      Key>Key1,!,
      delete_bst(Right, Key,NewRight).
delete_bst(nil,Key,nil):-!,
      write(Key),write(' nu exista in arbore').
delete_bst(t(nil,Key,nil), Key,nil):-!.
delete_bst(t(nil,Key,Right), Key,Right):-!.
delete_bst(t(Left,Key,nil), Key,Left):-!.
```

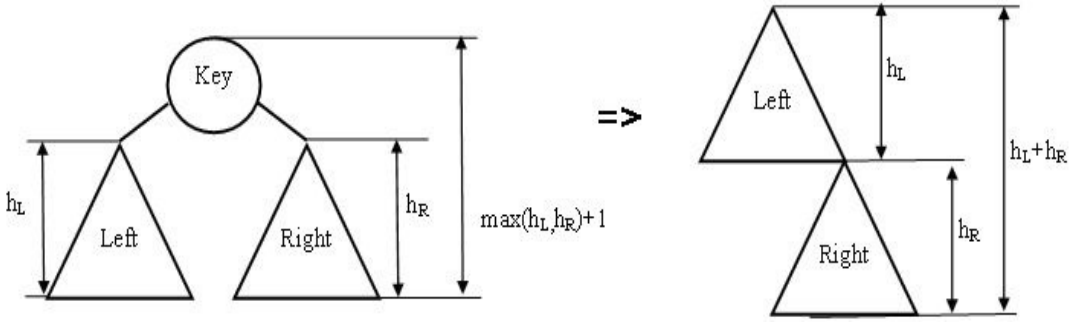
Qs:

- What cases did we cover so far?
- Which clause covers which case? Explain!
- Do we need all clauses so far? Why/why not? Justify

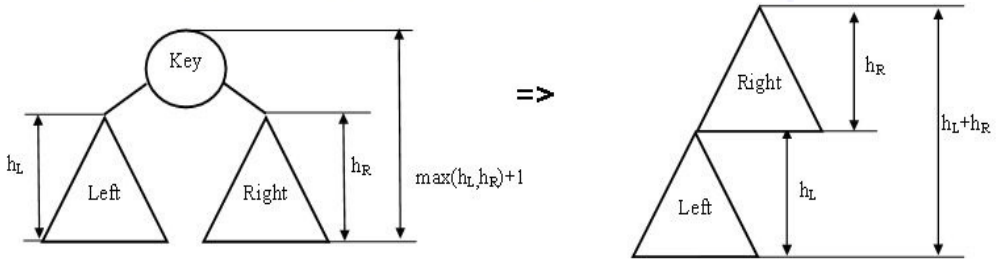
Delete from BST – code – contd.

- We reached the node containing the key to delete
- Has 2 children = is the root of a subtree => problem is remove the root of the tree
- How to…? Think! We have alternatives:
- Either link the subtrees:
  - Easy to apply (time estimate)
  - Disadvantage?

Sol1



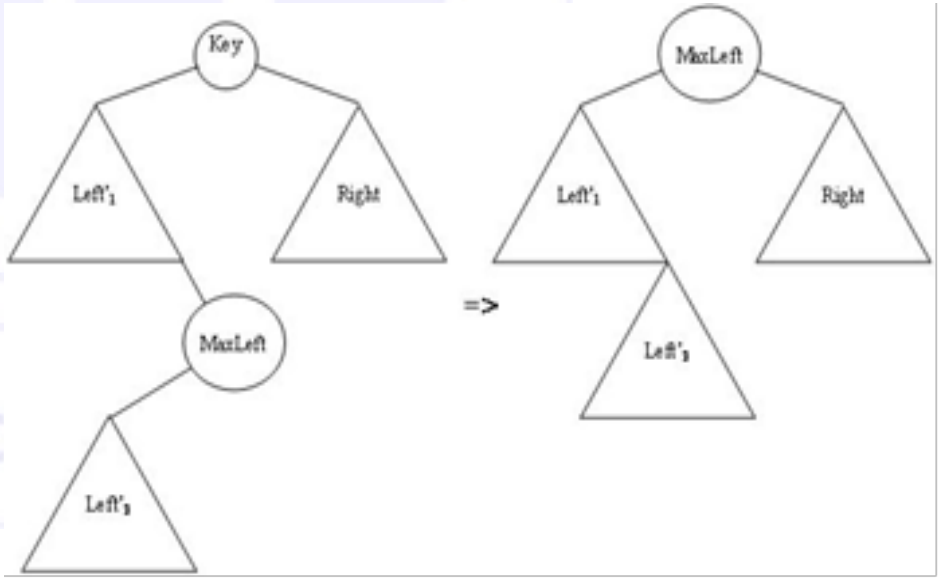
Sol2



Delete from BST – code – contd.

- Or replace the problem to solve with a problem you know to solve!
- Remove INSTEAD the node you want to remove (is the root, remember this) with another one. How:
  - Find an easy to remove node
  - Place it instead of the root (is possible?)
  - To be possible, that node should be a node in inorder:
    - Either before the key in the root
    - Or after the key in the root
  - So replace with
    - Predecessor
    - Successor
  - !
  - Are those nodes with the desired property?
  - Justify

Sol3



Delete from BST – code – contd.

```
delete_bst(t(Left,Key,Right), Key,t(NewLeft,MaxLeft,Right)):-!,
        deleteMaxFromTree(Left,MaxLeft,NewLeft).

% deleteMaxFromTree/3
% deleteMaxFromTree(in_tree,max_key,out_tree).
```

- The predicate takes a tree as input `in_tree`, finds the max key `max_key` (the rightmost node) and removes it from the tree, returning the tree without that key `out_tree`.

```
deleteMaxFromTree(t(Left,Key,Right),MaxKey,t(Left,Key,NewRight)):-
        deleteMaxFromTree(Right,MaxKey,NewRight).
deleteMaxFromTree(t(Left,MaxKey,nil),MaxKey,Left):-!.
```

- Clause 1 says: as long as the right subtree is not empty, go to the right
- Clause 2 says: if right is `nil`, you found the max key `MaxKey` and `Left` is out tree.
- Do we ever reach second clause? Infinite loop. Clauses not in correct order! Let's change it!

Delete from BST – code – contd.

```
deleteMaxFromTree(t(Left,MaxKey,nil),MaxKey,Left):-!.
deleteMaxFromTree(t(Left,Key,Right),MaxKey,t(Left,Key,NewRight)):-
        deleteMaxFromTree(Right,MaxKey,NewRight).
```

- Now is highly inefficient! We always try the non-useful clause first! Too much!
- Do we really need to have the fact first?
- No, is good to go second (JUSTIFY!).

```
deleteMaxFromTree(t(Left,Key,Right),MaxKey,t(Left,Key,NewRight)):-
        deleteMaxFromTree(Right,MaxKey,NewRight).
deleteMaxFromTree(t(Left,MaxKey,nil),MaxKey,Left):-!.
```

Delete from BST – complete code

```
delete_bst(t(Left,Key1,Right),Key,t(NewLeft,Key1,Right)):-
        Key<Key1,!,
        delete_bst(Left,Key,NewLeft).
delete_bst(t(Left,Key1,Right),Key,t(Left,Key1,NewRight)):-
        Key>Key1,!,
        delete_bst(Right,Key,NewRight).
delete_bst(Key,nil,nil):-!,
        write(Key),write('not in tree').
delete_bst(t(nil,Key,Right),Key,Right):-!.
delete_bst(t(Left,Key,nil),Key,Left):-!.
delete_bst(t(Left,Key,Right),Key,t(NewLeft,MaxLeft,Right)):-!,
        deleteMaxFromTree(Left,MaxLeft,NewLeft).


deleteMaxFromTree(t(Left,Key,Right),MaxKey,t(Left,Key,NewRight)):-
        deleteMaxFromTree(Right,MaxKey,NewRight).
deleteMaxFromTree(t(Left,MaxKey,nil),MaxKey,Left):-!.
```

Delete from BST – code analysis

- Search phase: O(h)
- Delete (no matter the solution chosen): O(h)
- Overall: 2h
- O(h), constant 2. Is it so?
- It is just O(h), constant 1. JUSTIFY!
- h=? Make a general analysis per types of trees.

Incomplete structures Lists

- Structures ended in variables = instead of the specific empty structure it ends in a variable.
- This IS a list ended in variable: L_1=[1,2,3**|A**]
  - Tail is VARIABLE
  - Can you specify its length?
- This IS NOT a list ended in variable L_2=[1,2,3**,B**]
  - Last element variable, can be anything (including a list, such as [5,6]). BUT in case it unifies a variable, the structure would be a heterogeneous one: L_2=[1,2,3,[5,6]]

Incomplete Lists

- Basic operations
  - Search
  - Insert

```
member(H,[H|_]).
member(H,[_|T]):-
       member(H,T).
?- L=[1,2,3|_],member(2,L).
Yes, L=[1,2,3|_]

?- L=[1,2,3|_],member(4,L).
Yes, L=[1,2,3,4|_]
```

- Bad behavior. Why? How can we fix the issue?

Incomplete Lists - search

```
member_IL(H,[H|_]):-!.    %item found, stop. For good!
member_IL(H,[_|T]):-      %not found,
      member_IL(H,T).     % go search in tail
member_IL(_,L):-
      var(L),!,           %reached final free variable
      fail.               %stop with failure.
?- L=[1,2,3|_],member_IL(2,L).
Yes, L=[1,2,3|_]

?- L=[1,2,3|_],member_IL(4,L).
Yes, L=[1,2,3,4|_]
```

- Again same bad behavior. How come we didn't fix the issue?

Incomplete Lists – search – contd.

- NEVER reach clause #3 in the prev. code? Why?

```
member_IL(_,L):-
        var(L),!,              %this order: check, cut, fail
        fail.
member_IL(H,[H|_]):-!.         %cut mandatory
member_IL(H,[_|T]):-
        member_IL(H,T).
?- L=[1,2,3|_],member_IL(2,L).
Yes, L=[1,2,3|_].              %exe stops on clause 2
?- L=[1,2,3|_],member_IL(4,L).
No.                            %exe stops on clause 1
```

- **Stop conditions** for **incomplete** structures are (i) with **cut** (!) and (ii) ALWAYS **come first**. Otherwise, it is as if they are missing!
  - Starts with failure condition(s). It MUST be explicit (NEVER default).
  - Continues with the successful stop condition(s).

Incomplete Lists – insert

```
insert_IL(X,L):-
      var(L),          % reached the final free variable?
      !,               % stop backtracking
      L=[X|_].         % update the list with the new added item
insert_IL(H,[H|_]):-!.%item found, and don't allow backtracking
insert_IL(H,[_|T]):-
      insert_IL(H,T).

q1:?- L=[1,2,3|_], insert_IL(4,L).
Yes, L=[1,2,3,4|_].
q2:?- L=[1,2,3|_], insert_IL(3,L).
Yes, L=[1,2,3|_].
```

- q1: pure insert behavior (stop on clause 1)
- q2: member behavior (stop on clause 2)

Incomplete Lists – insert – contd.

- We don't really need clause 1. Clause 2 covers it (default).

```
insert_IL(H,[H|_]):-!.
insert_IL(H,[_|T]):-
      insert_IL(H,T).
```

q1:?- L=[1,2,3|_], insert_IL(4,L).

Yes, L=[1,2,3,4|_].

Has pure insert behavior. Clause behaves as:

```
insert_IL(X,L):-
      var(L),!,L=[X|_].
```

q2:?- L=[1,2,3|_], insert_IL(3,L).

Yes, L=[1,2,3|_].

Has member. Clause behaves as:

```
member(X,[H|_]):-X=H, !.
```