Logic Programming

Rodica Potolea
Camelia Lemnaru

Lecture #8

CS@TUCN

Agenda

- Incomplete Trees – review
  - `search` to fail when `key` not in tree
  - `search` to fail when <`key`,`value`> given, `key` is in tree, but with a different `value`
- Difference lists
  - Review
  - Quicksort
  - queues
  - Flattening lists
- (other) Built in predicates

Search/insert in Incomplete BST

- The regular from a BST in an Incomplete BST has dual behavior:
  - Returns found in case in tree
  - Inserts it at the bottom level in case not found

- If want to fail if key not found? Add an explicit **first** clause as:

```
search(_,T):-
        var(T),!,
        fail.
```

- What other situations? Nodes contain <key,value>
- `search` to fail if key exists yet with a different value!

```
search(Key,n(_,a(Key1,Info1),_),Info):-
        Key=Key1,!,
        Info=/=Info1,!,
        fail.
```

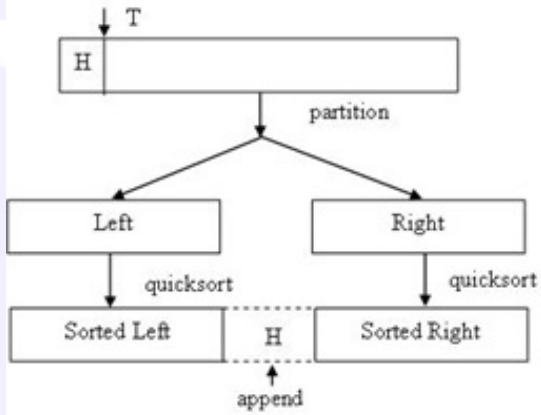
TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

Difference lists

- Difference lists
  - Pair of arguments (First,Last) or First\Last, pointing to the first and respectively AFTER the last element of the list
  - Allow to speed up execution (avoid second traversal of list)
  - Needs special initial call => make the warm up call by setting Last on call to []
  - What if instead of [] we have some content?
  - Stop condition: empty list as a difference; therefore, the difference between the same variable (L,L).

Quicksort

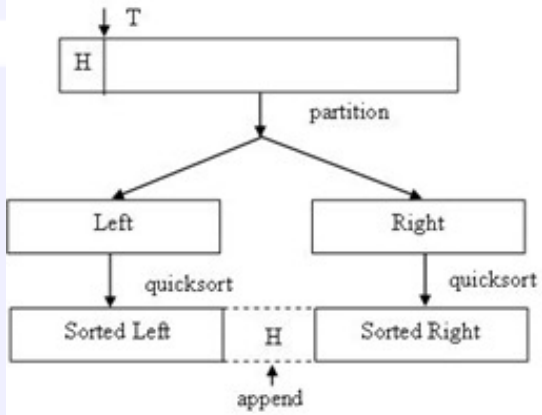
```
partition(X,[H|T],[H|Left],Right):-
      H<X,!,
      partition(X,T,Left,Right).
partition(X,[H|T],Left,[H|Right]):-
      partition(X,T,Left,Right).
partition(_,[],[],[]).
```




```
quicksort([H|T],Result):-
      partition(H,T,Left,Right),
      quicksort(Left,SortedLeft),
      quicksort(Right,SortedRight),
      append(SortedLeft,[H|SortedRight],Result).
quicksort([],[]).
```

Quicksort - analysis

- A divide et impera strategy
- Cost analysis per cases:
  - O(nlgn) in best and average case (why?)
  - O(n^2) in worst (which?)
- Multiplicative constant larger than in imperative implementations! Why?
- How can be avoided?



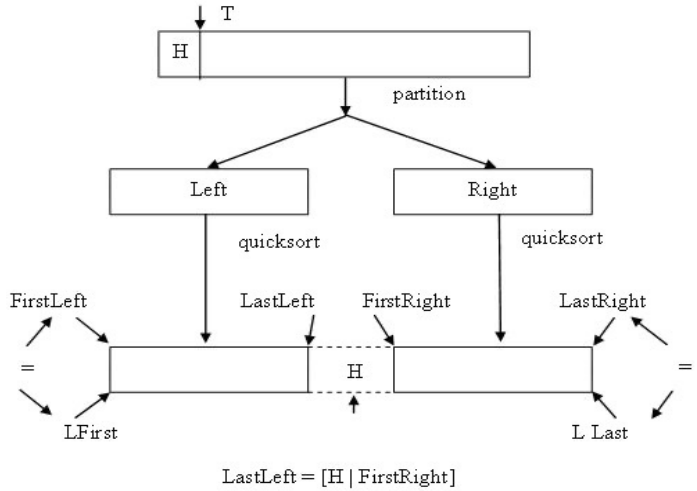

```
quicksort([H|T],Result):-
        partition(H,T,Left,Right),
        quicksort(Left,SortedLeft),
        quicksort(Right,SortedRight),
        append(SortedLeft,[H|SortedRight],Result).//how can
                                                  //we skip it?

quicksort([],[]).
```

Quicksort – with DL

%qsort_LD1/3

%qsort_LD1(in_list,first_olist,last_olist)



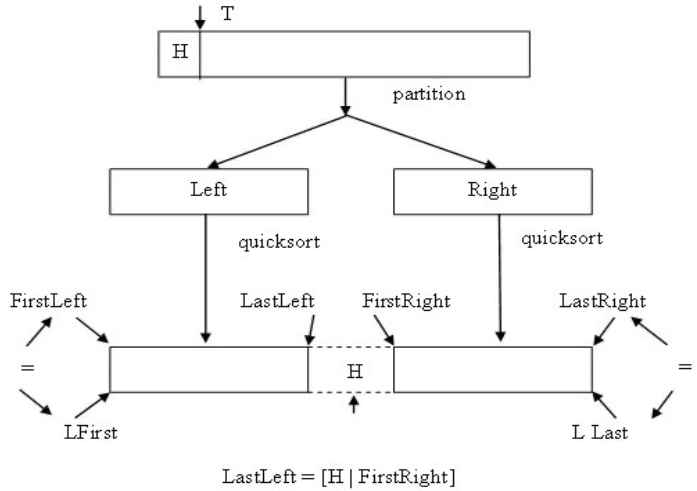

```
qsort_LD1 ([H|T],LFirst,LLast):-
      partition(H,T,Left,Right),
      qsort_LD1(Left,FirstLeft,LastLeft),
      qsort_LD1(Right,FirstRight,LastRight),
      LFirst=FirstLeft,
      LLast=LastRight,
      LastLeft=[H|FirstRight].
qsort_LD1 ([],L,L).
```

Q: `partition` should not return difference lists? How should we do that? Oral discussion.

Quicksort – with DL (default unifications)

%qsort_LD2/3

%qsort_LD2(in_list,first_olist,last_olist)




```
qsort_LD2([H|T],LFirst,LLast):-
        partition(H,T,Left,Right),
        qsort_LD2(Left,LFirst,[H|Right]),
        qsort_LD2(Right,Right,LLast).
qsort_LD2([],List,List).
```

Q needs special initial call, to initialize last to empty:

```
quicksort2(List,Result):-
        qsort_LD2(List,Result,[]).
```

Queues with D
(section 15.4 in The art of Prolog – Shapiro)

```
%enqueue/5
%enqueue(el2EnQ,   firstQBefore,lastQBefore,
                   firstQAfter,lastQAfter)

enqueue(Eel,First,[Eel|Last],Fist,Last).
```

Means:

```
enqueue(Eel,FQB,LQB,FQA,LQA):-
            FQA=FQB//after adding in the Q, it starts in the same place
            LQB=[H|LQA],  //before adding, the list ends IN FRONT
            H= Eel.       //of the item to be added
```

How to query it?

```
?-enqueue(Item,FB,LB,FA,LA).
```

In the usual employment FB and LB are the First/Last element in the Queue before the call (known arguments), and FA and LA the First/Last element in the Queue after the call (unknown arguments at call time).

Queues with D
(section 15.4 in The art of Prolog – Shapiro)

```
%dequeue/5
%dequeue(el2DeQ,   firstQBefore,lastQBefore,
                   firstQAfter,lastQAfter)
dequeue(Eel,[Eel|First],Last,Fist,Last).
```

Means:

```
dequeue(Eel,FQB,LQB,FQA,LQA):-
            FQB=[H|FI],//before removing from the Q, the Q has
            Eel=H,  //extracted element is the front of the Q
            FI=FQA,  //some items which is removed, followed
                     // by the reminding = FirstQueueAfter
            LQA=LQB.  //after removing, the list ends where it ended
                     //as removal is from front
```

How to query it?

```
?-dequeue(Item,FB,LB,FA,LA).
```

In the default meaning, Item gets instantiated at the end of the execution. What happens if we ask to dequeue some specific element (say 7), yet, it is NOT the first in the queue? That is, we specify Item.

Types of lists

- Regular (ended in [])
- Ended in variable
- Difference lists
- Write specific predicates to make transformations among each of the 3 known types of lists (section 7.3 of the textbook):
  - **Regular to ended in variable**
  - Regular to difference
  - Ended in variable to regular
  - **Ended in variable to difference**
  - Difference to regular
  - Difference to ended in variable

Deep lists

- What is a deep list?

L=[1,1,[2,2,2,[3,3,[4],3],2,2,[3,[4,[5,5,5]]],2]]

- Levels? How do we know?
- Flatten deep lists. Meaning?

L=[1,1,2,2,2,3,3,4,3,2,2,3,4,5,5,5,2]

- How, without knowing structure/#of levels?

Flatten Deep lists

```
%deep2flat/2
%deep2flat(deep_list,flat_list).

deep2flat([],[]):-!.
deep2flat(H,[H]):-atomic(H),!.
deep2flat([H|T],[H|FlattenedTail]):-
      atomic(H),!,
      deep2flat(T,FlattenedTail).
deep2flat([H|T],FlattenedList):-
      deep2flat(H,FlattenedHead),
      deep2flat(T,FlattenedTail),
      append(FlattenedHead,FlattenedTail,FlattenedList.
```

L=[1,1,[2,2,2,[3,3,[4],3],2,2,[3,[4,[5,5,5]]],2]]

Flatten Deep lists – with difference lists - explicit

```
%deep2flat_dl/3
%deep2flat_dl(deep_list,first_flat_list, last_flat_list).

deep2flat_dl([],First,Last):-!,First=Last.
deep2flat_dl(H,First,Last):-  atomic(H),!, //[H]
                              First=[H|Last].
deep2flat_dl([H|T],First,Last):-
                    atomic(H),!,
                    First=[H|FirstT],
                    deep2flat_dl(T, FirstT,LastT),
                    Last=LastT.
deep2flat_dl([H|T],First,Last):-
                    deep2flat_dl(H,FirstH,LastH),
                    deep2flat_dl(T,FirstT,LastT),
                    First=FirstH,
                    LastH= FirstT,
                    Last=LastT.
```

Flatten Deep lists – with difference lists - default

```
%deep2flat_dl/2
%deep2flat_dl(deep_list,first_flat_list, last_flat_list).

deep2flat_dl([],Last,Last):-!.
deep2flat_dl(H,[H|Last],Last):-atomic(H),!.
deep2flat_dl([H|T],[H|FlattenedTail],Last):-
            atomic(H),!,
            deep2flat_dl(T,FlattenedTail,Last).
deep2flat_dl([H|T],FlattenedList,Last):-
            deep2flat_dl(H,FlattenedList,Int),
            deep2flat_dl(T,Int,Last).

flatten(Deep,Flat):-
       deep2flat_dl(Deep,Flat,[]).
```

Flatten Deep lists – homework

- Try to flatten a deep least taking just elements with various properties:
  - Heads of lists ([1,2,3,4,3,4,5])
  - Atomic elements on last positions of lists ([4,3,5,2])
  - All atomic elements from lists at **odd**/even levels ([2,2,2,4,2,2,4,2])
  - Heads of lists at odd/**even** levels ([1,3,3,5])
  - Atomic elements on odd/even positions on any list
  - ….imagine ☺
  - (section 8.3 of the textbook)

L=[1,1,[2,2,2,[3,3,[4],3],2,2,[3,[4,[5,5,5]]],2]]

Creating/accessing structures Built-in predicates

- `functor(T,F,N)`
  - means T is a structure named F and arity N
  - 2 functionalities:
  - **if** nonvar(T) **then** F matches the functor of T
    N matches the arity of T
  - **if** var(T) **and** nonvar(F) **and** nonvar(N)
    then T becomes a structure
    with name F and N arguments

functor(T,F,N)

if nonvar(T) **then** F matches the functor of T

N matches the arity of T

```
?-functor(a+b,F,N).
Yes, F=+, N=2
?-functor([a,b,c],F,N).
Yes, F=., N=2
```

(why 2? Which ones?)

```
?-functor([a,b,c],F,3).
```

Fails. Why?

```
?-functor(a,F,N).
Yes, F=a, N=0
?-functor(f(a,b),F,N).
Yes, F=f,N=2.
```

functor(T,F,N)

if var(T) **and** nonvar(F) **and** nonvar(N)

then T becomes a structure with name F and N arguments

```
?-functor(F,f,3).
Yes, F=f(A,B,C).
?-functor(F,+,2).
Yes, F=A+B
```

Creating/accessing structures Built-in predicates

- `arg(N,T,A)`
  - means the N^{th} argument of T is A
  - N and (partially)T must be instantiated

```
?-arg(2,f(a,b),X).
Yes, X=b.
?-arg(1,a+b+c,X).
Yes, X=a
?-arg(3,a+b+c,Y).
No. WHY?
?-arg(2,a+b+c,Z).
Yes, Z=b+c
?-arg(1,[a,b,c],X).
Yes, X=a
?-arg(2,[a,b,c],Y).
Yes, Y=[b,c]
```

Creating/accessing structures Built-in predicates

- `X=..L` INFIX predicate (a.k.a. UNIV)
  - means list L contains as first element the functor of X, followed by arguments of X
  - 2 functionalities:
  - **if** var(X) **then** L is used to construct the appropriate structure of X; the head of L must be an atom, and becomes the functor of X
  - **if** nonvar(X) **then** functor of X matches the first element (head) of L, while the arguments of X matches with the tail of the list (left to right arguments of X, left to right in tail of L)

X=..L

if var(X) **then** L is used to construct the appropriate structure of X; the head of L must be an atom, and becomes the functor of X

```
?-X=..[a,b,c,d].
Yes, X=a(b,c,d)
?-X=..[member,a,[b,c]].
Yes, X=member(a,[b,c]).
```

X=..L

if nonvar(X) **then** functor of X matches the first element (head) of L, while the arguments of X matches with the tail of the list (left to right arguments of X, left to right in tail of L)

```
?-f(a,b,c)=..X.
Can we ask this way (with X on the left)?
Yes, X=[f,a,b,c].
?-append([H|T],L,[H|R])=..X.
Yes, X=[append,[H|T],L,[H|R]].
?-(a+b)=..L.
Yes, L=[+,a,b].
?-(a+b+c)=..L.
Yes, L=[+,a,b+c]
?-[a,b,c,d]=..[H|T].
Can we ask this way (with list on the left)?
Yes, H=., T=[a,[b,c,d]]
```

Substituting expressions

- Given a (complex) expression, it is requested to replace a given subexpression with another one.

- `subst(OldExpr,NewExpr,OldSubExpr,NewSubExpr).`
- Example: in a large arithmetic expression replace (7-2*x) whereever it occurs with another one, say (2*y+3/z)
- 2 solutions
  - =..
  - functor + arg

```
//subst/4
//subst(OldExpr,NewExpr,OldSubExpr,NewSubExpr).

//subst_args/4
//subst_args(ListOldExpr,ListNewExpr,OldSubExpr,NewSubExpr).
```

Substituting expressions =..

```
subst(Old,New,Old,New):-!.  //in case the whole expression is represented
//by only the subexpression to be replaced, the old one is replaced by the new one
subst(Val,Val,_,_):-                    //in case the expression is an atomic value
                atomic(Val),!.          //it remains unchanged
subst(Val,NewVal,OldSubExpr,NewSubExpr):-
        Val=..[F|Args], //extract from the old expression its functor and the list
                                //of arguments
        subst_args(Args,NewArgs,OldSubExpr,NewSubExpr),
//take the old list of arguments and process it with substitution
        NewVal=..[F|NewArgs]. //create the new expression with the same
//functor and the new list of arguments, updated by substitutions
subst_args([],[],_,_).
subst_args([Arg|Args],[NArg|NArgs],Old,New):-
        subst(Arg,NArg,Old,New),  //as Arg is an expression, go back to the
//other predicate
        subst_args(Args,NArgs,Old,New). //continue with the rest of
//arguments, tail of list
```

Substituting expressions =.. Just code, comments removed

```
subst(Old,New,Old,New):-!.
subst(Val,Val,_,_):-
            atomic(Val),!.
subst(Val,NewVal,OldSubExpr,NewSubExpr):-
      Val=..[F|Args],
      subst_args(Args,NewArgs,OldSubExpr,NewSubExpr),
      NewVal=..[F|NewArgs].

subst_args([],[],_,_).
subst_args([Arg|Args],[NArg|NArgs],Old,New):-
      subst(Arg,NArg,Old,New),
      subst_args(Args,NArgs,Old,New).
```

Qs:

1. Order of processing the structure?
2. Where the execution ends?

Substituting expressions functor+arg

```
subst(Old,New,Old,New):-!.          //same as before
subst(Val,Val,_,_):-                //same as before
                atomic(Val),!.
subst(Val,NewVal,OldSubExpr,NewSubExpr):-
        functor(Val,F,N),  //extract from the old expression Val its functor F
                           // and the number of arguments N
        functor(NewVal,F,N), //create a new expression NewVal from the
                           // known functor F  and N  arguments, free variables at the time
        subst_args(N,Val,NewVal,OldSubExpr,NewSubExpr).
//for each of the arguments, 1, 2, …, N, process it one at a time
subst_args(0,_,_,_,_):-!.//arg 0 needs no change
subst_args(N,Val,NewVal,Old,New):-
        arg(N,Val,OldArg),//extract the Nth arg of the old expression
        arg(N,NewVal,NewArg),//set the Nth arg of the new expression; a var at the time
        subst(OldArg,NewArg,Old,New),  //as OldArg is an expression, go back
//to the other predicate  and make the same processing
        N1 is N-1,     //process next the previous argument
        subst_args(N1,Val,NewVal,Old,New). //continue with the rest of
//arguments, tail of list
```

Substituting expressions functor+arg
Just code, comments removed

```
subst(Old,New,Old,New):-!.
subst(Val,Val,_,_):-
                atomic(Val),!.
subst(Val,NewVal,OldSubExpr,NewSubExpr):-
       functor(Val,F,N),
       functor(NewVal,F,N),
       subst_args(N,Val,NewVal,OldSubExpr,NewSubExpr).

subst_args(0,_,_,_,_):-!.
subst_args(N,Val,NewVal,Old,New):-
       arg(N,Val,OldArg),
       arg(N,NewVal,NewArg),
       subst(OldArg,NewArg,Old,New),
       N1 is N-1,
       subst_args(N1,Val,NewVal,Old,New).
```

Qs:

1. Order of processing the structure?
2. Where the execution ends?