600.325/425 — Declarative Methods
Homework 3: (Constraint) Logic Programming

Spring 2017
Prof. Jason Eisner

Due date: Friday, April 7, 2 pm

This homework consists of some simple programming exercises. The main goal is to get you comfortable doing logic programming and constraint logic programming, not to attack larger-scale problems as in previous homeworks.

As you know, you have already written Prolog programs for this class; ECL^{i}PS^e is built by adding features to Prolog, and in this homework you will continue to use ECL^{i}PS^e as your Prolog interpreter. Be especially sure to do problems 7, 8, and 11. In the next homework, we may try to obtain more efficient solutions to those problems, using dynamic programming rather than backtracking.

Instructions

Academic integrity: As always, the work you hand in should be your own, in accordance with the regulations at `http://cs.jhu.edu/integrity-code`. Partners are *not* allowed for this homework.

325 vs. 425: Problems marked "**425**" are only required for students taking the 400-level version of the course. Students in 325 are encouraged to try and solve them as well, and will get extra credit for doing so.

Getting started: To obtain the homework files, run the following on a Unix terminal (e.g. `ugrad` machines:

```
wget https://www.cs.jhu.edu/~jason/325/hw3.zip
unzip hw3.zip
```

The resulting `hw3` directory includes some "stub" implementations that you will complete.

Testing your code: You can run some simple tests on your completed implementations with

```
eclps < runtests.ecl
```

This pipes a series of queries to the REPL (so you don't have to type them all out yourself). The provided tests are extremely limited and you should write a few more of your own.

Handing in your work: You will submit a written part and a programming part on Gradescope. **Please do not put your name in either part.**

- For the **written** part: Please submit to **Homework 3 Written** a PDF with each problem on a separate page in your submission. Note that every problem (**1 .. 11**) has at least one written question. After submitting, please mark the area for each question using Gradescope's interface.

- For the **programming** part: In your `hw3` directory, run

  ```
  ./make_sol.sh
  ```

  This will check for the following files: `warmups.ecl`, `cut.ecl`, `more_lists.ecl`, `trees.ecl`, `runtests.ecl` and create a zipped file. Submit this zipped file to **Homework 3 Programming** on Gradescope.

You are allowed unlimited submissions, but the problems will not be graded until after the deadline. Please try submitting early (even if it is incomplete) so that Gradescope can check that your output formats are correct.

Warmups

[37 or 39 points]

1. For each of the following pairs of terms, what new term—if any—is found by unifying them? And what variable bindings are produced by the unification?

   (*Hint:* A term like `foo(penguin,X,X)` can be regarded as standing for the infinite set of terms that could be obtained by instantiating its variables, such as `foo(penguin, any(old,subterm), any(old,subterm))`. Then unifying two terms corresponds to intersecting two sets.)

   (*Hint:* In each case, you could use ECL^{i}PS^e to check most of your answer.)

   (a) **[2 points]** `foo(X,X,Y)` with `foo(A,B,B)`?

   (b) **[2 points]** `foo(X,Y)` with `foo(A,B,C)`?

   (c) **[2 points]** `[X,2,X,4,Y]` with `[1,A|B]`?

   (d) **[2 points]** `f(A,g(B))` with `f(h(D),E)`?

2. (a) **[2 points]** Write a predicate `greater_than/2` that compares two Peano integers. For example, `greater_than(s(s(z)),s(z))` should return `No`, but `greater_than(s(s(s(z))),s(s(z)))` should return `Yes`.

   Your program should only be a very few lines. In fact, that is true for all the programs on this homework. But you will really have to think declaratively to figure out what those few lines should be!

   Note: This definition would actually work:

   ```
   greater_than(A,B) :- add(B,s(_),A).
   ```

   In other words, $A > B$ if you can get from B to A by adding some positive number. However, for purposes of this question, just write `greater_than` directly and as simply as possible. Don't call `add` or anything else.

   Note: Assume that the documentation for your predicate says "This predicate should be called with Peano integers as its arguments. There is no guarantee about what the results will be otherwise." (This bit of documentation is known as a contract with the caller—if a function/method/predicate is called *in a certain way*, then it will guarantee certain things about the results. A type system could help prevent a caller from violating the contract, e.g., it could

prevent the caller from passing in anything but a Peano integer. But standard Prolog is untyped. This is because a variable `X` means "any term"—there is no way to create a variable that is only capable of unifying with Peano integers.)

(b) **[2 points]** What is the runtime of comparing ground Peano integers `greater_than/2`, in terms of the magnitudes n, m of those integers?

(c) **[2 points]** After testing your predicate in mode (+,+) as above, also try the other basic modes (+,-), (-,+), and (-,-). What happens in each case? Explain.

(d) **[2 points]** The query
`greater_than(s(A),B), greater_than(s(B),A).` `%` $A+1 > B$ and $B+1 > A$
enumerates infinitely many answers through backtracking:

```
A = z, B = z ;
A = s(z), B = s(z) ;
A = s(s(z)), B = s(s(z)) ;
A = s(s(s(z))), B = s(s(s(z))) ;
```

and so on. In each answer, `A` and `B` are the same term.

It is tempting to think that this is one of those cases where the standard Prolog solver is just not all that smart, and that a smarter solver would just return `A=B` and leave it at that.

However, in fact that would be the wrong conclusion. Any correct Prolog solver would have to enumerate solutions in this case. Why?

(e) **[2 points]** **[425]** When Prolog answers

```
A = s(s(z)), B = s(s(z)) ;
```

in the previous question, do `A` and `B` point to the *same* copy of `s(s(z))` in memory, or to two wholly disjoint copies of `s(s(z))` in memory, or to two partly shared copies (e.g., with the same `z` but different `s`'s)? To answer this, you should hand-simulate on paper the steps that the Prolog solver goes through when answering this query.[1]

Remark: In Java, we would say that `A.equals(B)` because both `A` and `B` refer to `s(s(z))`: they "compare as equal." But in Java, we could also ask whether `A==B`: `A` and `B` actually the *same object* in memory? That's what is being asked here.

This is an important question in Java, since if they are the same object, then modifying `A` would change `B` as well. In Prolog, the question doesn't matter (except for efficiency) because you can't modify ground terms like `s(s(z))` anyway, even by unification. And as far as I know, there is no way to ask the Prolog interpreter whether two ground terms are the *same object* in memory. But I am asking you to figure it out based on your knowledge of the Prolog solver.

3. (a) **[2 points]** Write a predicate `duplicate/2` that duplicates every element of a list. Querying `duplicate([a,b,c], L)` should return `L = [a,a,b,b,c,c]`, and querying `duplicate(M, [a,a,b,b,c])` should fail (i.e., answer `No`).

(b) **[1 point]** What should be returned by the query `duplicate(M, [f(3),f(X),X,Y])`? Try it and see if it works!

[1] Assume that your version of Prolog doesn't go to any special trouble to save space by finding and reusing old copies of ground terms. Although there are probably Prologs that do this, using a technique called "interning" or "hash consing" ...

4. (a) **[4 points]** Write a predicate `swap/2` that nondeterministically chooses two adjacent elements in a list and swaps them. Querying `swap([a,b,c,d], L)` should nondeterministically bind `L` to `[b,a,c,d]`, `[a,c,b,d]`, and `[a,b,d,c]`. (That is, you should get these three lists when you repeatedly press semicolon.) What should `swap([a],L)` or `swap([],L)` do, and why?

   (b) **[2 points]** Consider this definition:

   ```
   mystery(W,Z) :- swap(W,X), swap(X,Y), swap(Y,Z).
   ```

   What does this do? How many values for `L` will be returned by `mystery([1,2,3,...`n`],L)`? Are all of these values different from one another? Why or why not?

5. (a) **[4 points]** Write a Prolog constraint `pow(A,B,C)` that is satisfied iff $\texttt{C} = \texttt{A}^{\texttt{B}}$. You may assume that `A`, `B`, and `C` are all Peano integers.

   You should use `add(A,B,C)` and `mult(A,B,C)`, which we defined in class. However, please state the exact definitions you are using, because the runtime in a later question will depend on the exact definitions you choose. (Some versions are more efficient than others!)

   (b) **[2 points]** How will your program answer the query `pow(z,z,C)`? In other words, what is 0^0?[2]

   (c) **[2 points]** What will your program answer if `A`, `B`, and `C` are *not* all Peano integers?

   (d) **[3 points]** In general, how efficient is it to answer a query of mode `pow(+A,+B,-C)`? (This mode means that `A` and `B` are ground terms representing Peano integers, whereas `C` is a variable that Prolog must fill in.) What is the Prolog solver doing internally? Give the asymptotic complexity of this query as a function of the integers `A` and `B`.

   (e) **[3 points]** How would you use your `pow` constraint to ask about roots or logarithms? What happens when you try these queries out, with the above definitions? What is the Prolog solver doing internally?

The Prolog Cut

6. **[10 points]** Let's experiment a little with the Prolog cut. Remember that the cut is *not* declarative—it is a way of controlling the backtracking behavior of the Prolog solver. Specifically, it tells the solver *not* to backtrack in certain circumstances, "cutting off" the alternative branches of the backtracking search tree. If there are no solutions to the constraint problem on those alternative branches anyway, it is a "green cut" that simply speeds up the solver. If it kills off what would otherwise be solutions, it is a "red cut."

   (a) **[0 points]** The `member/2` function is already built in (because it is so useful). Try the following queries at the ECL^{i}PS^e prompt, using semicolon to get *all* the solutions:

   - `member(X,[a,b,c,b]).`
   - `member(b,[a,b,c,b]).`
   - `member(b,[a,Y,c,b]).`

   Feel free to experiment with other queries as well, such as `member(X,[Y,Z])` or `member(f(0,X),[a,f(0,1),f(0,2),f(Y,3)]`.

   (b) **[0 points]** Now try these versions with an added `!` subgoal, the cut:

[2] The mathematical "right" answer isn't obvious since ordinarily $0^n = 0$ while $n^0 = 1$.

- `member(X,[a,b,c,b]), !.`
- `member(b,[a,b,c,b]), !.`
- `member(b,[a,X,c,b]), !.`

(c) **[0 points]** Let's package up the above pattern like this:

```
cutmember(X,List) :- member(X,List), !.
```

Make sure that that works as expected:

- `cutmember(X,[a,b,c,b]).`
- `cutmember(b,[a,b,c,b]).`
- `cutmember(b,[a,X,c,b]).`

You do not have to hand anything in for the above experiments.

(d) **[2 points]** Describe in words what `cutmember` does. Is the cut here red or green? If you want to check whether a ground integer appears in a long list of ground integers, what are some reasons to prefer `cutmember` over `member`?

(e) **[0 points]** Now try the following. Try to predict what they will do before you try them!

- `cutmember(L,[[a,b],[],[c,d]]), member(X,L).`
- `member(L,[[a,b],[],[c,d]]), cutmember(X,L).`
- `member(L,[[],[a,b],[c,d]]), cutmember(X,L).`

(f) **[6 points]** You might expect the following three to do the same thing as the ones above. Do they? Why or why not?

- `member(L,[[a,b],[],[c,d]]), !, member(X,L).`
- `member(L,[[a,b],[],[c,d]]), member(X,L), !.`
- `member(L,[[],[a,b],[c,d]]), member(X,L), !.`

(g) **[2 points]** Based on your experiments (and any further experiments you might try), explain in words how the cut interacts with subroutine calls.

7. **[10 points]** Here is a problem that we may return to if we cover Dyna this year.

We'd like to write a predicate `uniq/2` that eliminates duplicates from a list. This might be useful in using lists to represent sets, for example.

The desired behavior is best explained by a few examples. The query
`uniq([a,a,b,b,b,b,b,c,c], L)` should return `L = [a,b,c]`. The query
`uniq([1,a,b,2,3,4,b,5,6,a,7], L)` should return `L = uniq([1,2,3,4,b,5,6,a,7])`. Only the last copy of each element survives.

The following comes pretty close. It says that the first element can be thrown away if it is not an element of the rest of the list, and that the rest of the list should be uniqified.

```
uniq([],[]).
uniq([X|Xs],Ys) :- member(X,Xs), uniq(Xs,Ys).
uniq([X|Xs],[X|Ys]) :- uniq(Xs,Ys).
```

(a) **[2 points]** What goes wrong with the above solution?

(b) **[2 points]** To see why this problem can't be solved in pure Prolog, consider the query `uniq([3,X],[3,X])`. What would you *like* it to return, if it could? Can Prolog do this?

(c) **[2 points]** So let's go to impure Prolog. Where should you insert a cut or cuts into the program above, so that `uniq/2` will do the right thing, at least if its first argument is ground? Do this in `cut.ecl`.

(d) **[2 points]** What does your modified program do on the query `uniq([3,X],L)`? How about the query `uniq([3,X],[3,X])`? Why?

(e) **[2 points]** Unlike your `cutmember` queries in the previous problem, some `uniq` queries may unify with 2 or more clause heads. (This is true both for queries from the ECL^{i}PS^e command line and queries that appear as subgoals in the body of another clause.) So Prolog would ordinarily backtrack through all these choices of matching clause head, just as it also can backtrack through ways of satisfying the constraints in a given clause body.

Based on your experience above, explain how the cut affects this process. Does the order of the clauses matter? (Try it out!)

More Lists

8. **[15 or 20 points]** Here is a problem about increasing subsequences that we may return to if we cover Dyna this year.

(a) **[9 points]** Write a predicate `inc_subseq/2` that is true whenever its first argument is a ground list of integers and its second argument is a strictly increasing subsequence of that list. You may assume that the first argument will be ground.

For example, the query

```
inc_subseq([3,5,2,6,7,4,9,1,8,0], S)
```

should yield (among other things) `S = [3,5,6,7,9]`, `S = [3,5]`, `S = [5,6,7,8]`.

Make sure your code is capable of generating **all** strictly increasing subsequences, and make sure your subsequences are **strictly** increasing (i.e. use '`<`' instead of '`=<`').

One option would be to write `inc_subseq(Xs,Ys) :- subseq(Xs,Ys), ordered(Ys)`, and then define subseq/2 and ordered/1. Don't actually do it this way, as it's too inefficient to generate-and-test all 2^n subsequences. However, you can get some inspiration for your answer from the definition of ordered/1 that we developed in class: notice that that definition sometimes looks at the first *two* elements of a list, treating length-1 lists separately.

(b) **[2 points]** The following query finds all increasing subsequences of the specified list and tells you how many of them there are.

```
findall(S,inc_subseq([3,5,2,6,7,4,9,1,8,0],S),List), length(List,N).
```

Note that `length/2` is built-in.

Modify this query to find and count only subsequences of length 3.

(c) **[2 points]** In your answer to the previous question, is it more efficient to process the "`length`=3" constraint before or after the "`inc_subseq`" constraint? Explain.

(d) **[2 points]** The built-in ECL^{i}PS^e predicate `minimize/2` tries to find the minimum-cost satisfying assignment of some query; `minimize(Pred, X).` will assign to `X` the minimum value that satisfies `Pred`. To use this, you will have to load the `branch_and_bound` library by putting the command `:- lib(branch_and_bound).` at the beginning of your program. Naturally, you can use `minimize/2` to maximize a function by negating the cost. For example,

```
minimize( (member(X,[4,6,8,4,8,2]), Cost is -X),
          Cost).
```

will return `X = 8` (and `Cost = -8`).

Give a command that you could run to find the **longest** increasing subsequence of a given list. In the next part of the homework, we will discover a much more efficient way to compute this.

(e) **[425] [5 points]** Your `inc_subseq` program uses `<`, which only works on numeric variables that have already been instantiated.[3] Change this to `#<` and add `:-lib(ic)` at the beginning of your program. You are now doing *constraint* logic programming.

The program should still work on all the examples it worked on before. But now the first argument can contain uninstantiated variables.

Explain the meaning of the following query. Then try it and report the results. Are they correct? What do you conclude about how ECL^{i}PS^e handles the interaction between Prolog's usual backtracking (used in `inc_subseq`) and ECL^{i}PS^e's constraint store?

```
Vars=[A,B,C,D,E], Vars::1..4, inc_subseq(Vars,[E,C,A]),
   labeling(Vars).
```

Now try the query with the final "`labeling(Vars)`" omitted. Remember that "`labeling(Vars)`" says to instantiate all the uninstantiated variables in a way that satisfies the '`#<`' constraints. So you are now skipping that step, but Prolog's usual backtracking still happens.

The results will now have an odd format. Explain what is going on. You may find it helpful to try a simpler query first, such as

```
X::1..10, inc_subseq([7,X,3],S).
```

9. **[15 or 20 points]** Let's do a little more on *constraint* logic programming.

(*Note:* The constraints defined below could be handled more efficiently by specialized propagators, but I haven't shown you how to write your own propagators in ECL^{i}PS^e.)

(a) **[2 points]** Explain what this mystery predicate `p/2` does. (Feel free to try it out, or to try part (b), which uses it.)

```
:- lib(ic).
p([],1).
p([X|Xs],A) :- p(Xs,B), A #= X*B.
```

(b) **[2 points]** Explain what the following mystery predicate `q/2` does. Note that it is defined in terms of `p/2` above. You may want to try a query such as `q(L,20)`.

[3] If you try to run it on a list that contains variables, you will get an `Instantiation Fault`.

```
q(List,N) :- p(List,N), List::2..N, labeling(List).
```

(c) **[2 points]** The definition of `p/2` appears to set up a constraint program with exactly three numeric variables (`A`, `B`, `X`). Explain why this is false. How large is the constraint program, really?

(d) **[4 points]** Explain why `q([7,R],20)` returns "no" while while `q([7|Rs],20)` fails to halt. What, in detail, is the latter query doing as it runs forever?

(e) **[425] [5 points]** We saw in class that the following attempted definition of the "alldifferent" constraint is incorrect:

```
:- lib(ic).
adiff([]).
adiff([X]).
adiff([X|Xs]) :- member(Y,Xs), X #\= Y, adiff(Xs).
```

This definition says only that each list element must differ from *some* subsequent element. The *particular* subsequent element is chosen nondeterministically by backtracking in `member/2`. You can actually see this happen if you try the query `adiff([X,Y,Z])`. The first answer has delayed goals $X \neq Y, Y \neq Z$. The second answer has delayed goals $X \neq Z, Y \neq Z$ instead.

Fix the definition so that it has the appropriate meaning: each list element must differ from *all* subsequent elements. This means setting up $n(n-1)/2$ constraints without undoing any of those constraints by backtracking.

Thus, if you try the query `adiff([X,Y,Z])` with your revised definition, you should see 3 delayed goals $X \neq Y, X \neq Z, Y \neq Z$. And if you try `adiff([V,W,X,Y,Z])`, you should see a total of $5(5-1)/2 = 10$ delayed goals.

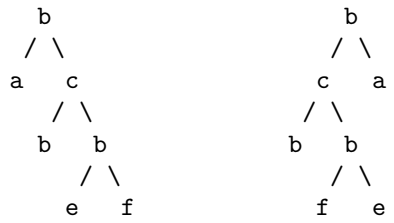
(*Hint:* Define a helper function to say that `X` must differ from all members of `Ys`.)

Trees

10. **[11 points]** Enough with lists. Prolog terms can represent arbitrary trees, so let's do a tree problem.

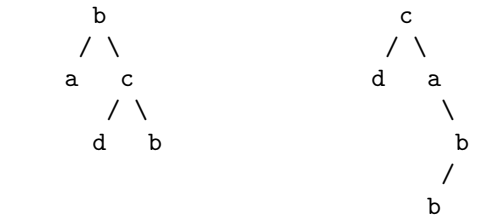
Two binary trees T and T' (not necessarily search trees) are called *isomorphic* if there exists a one-to-one correspondence between nodes of T and nodes of T', such that if $n \in T$ corresponds to $n' \in T'$, then (1) n and n' have the same label, and (2) the parent of n also corresponds to the parent of n' (or else neither one has a parent).

For example, the following trees *are* isomorphic:



"Isomorphic" literally means "same shape." Note that our definition considers the shape to be unchanged if the two children of a node are swapped. That is, we don't care about the order of siblings.[4]

By contrast, the following trees are not isomorphic. Even though they both contain the same collection of labels (`a`, `b`, `b`, `c`, `d`), the parent-child relations are quite different.



The following trees are not isomorphic either:

(a) **[8 points]** In Prolog, we can represent binary trees using structures of the form `t(Label, Left, Right)`, where `Label` denotes a label for the root node, and `Left` and `Right` are either '`nil`' or are tree structures themselves.

Define `isotree/2` so that `isotree(T1, T2)` should be true iff `T1` and `T2` are isomorphic. For example, your function should return '`Yes`' for the query

```
isotree(
   t(d, t(a, nil, nil), t(d, t(b, nil,nil), t(c, nil, nil) ) ),
   t(d, t(d, t(b, nil,nil), t(c, nil, nil) ), t(a, nil, nil) )   ).
```

since the two trees in question are isomorphic.

For the query

```
isotree(
   t(d, t(X, nil, nil), t(d, t(b, nil,nil), Y ) ),
   t(d, t(d, t(b, Z ,nil), t(c, nil, nil) ), t(a, nil, nil) )   ).
```

your program should return the assignments to `X`, `Y`, and `Z` that make the trees isomorphic, namely `X = a`, `Y = t(c, nil, nil)`, and `Z = nil`.

Note that there are two ways for `t(X, L1, R1)` and `t(X, L2, R2)` to be isomorphic—we could have `L1 = L2` and `R1 = R2`, or else `L1 = R2` and `R1 = L2`.

(b) **[3 points]** Give an example isotree query (on two constant trees) that demonstrates that Prolog may have to do an exponential amount of work to determine that two trees are not isomorphic. Your example should "tease" Prolog, leading it down many long blind alleys—apparently promising options that consume a lot of time but never actually work out.

[4] The trees above could both be family trees of the same family. Each shows a matriarch, `b`, with children `a` and `c`. The `c` child has twin daughters of his own, both named `b` (after his mom). The trees are the "same" – they just sometimes make different choices about which sibling to draw on the left and which to draw on the right.

Or you could think of them as two drawings of the same hanging mobile. As the mobile twists in the wind, sometimes the left and right subtrees of a node may switch places. Still, you can see from the drawings that the mobile always retains its overall shape, including the labels.

11. **[12 or 18 points]** After that warmup, here is another problem that we may return to if we cover Dyna this year.

    As you remember from your data structures class, a binary tree is a *binary search tree* if every node has the property that its label is greater than all of its descendants on its left branch, and its label is less than all of its descendants on its right branch.

    As an exercise, we will use Prolog to construct binary search trees in various ways.

    (a) **[6 points]** Write a predicate `inorder1(Tree,List)` that is true if `List` is the set of node labels of `Tree` produced by an in-order traversal. For example,

    ```
    inorder1(t(d,t(b,t(a,nil,nil),t(c,nil,nil)),t(e,nil,nil)), List).
    ```

    yields `List=[a,b,c,d,e]`.

    Hint: Use `append/3`, which is built-in.

    (b) **[6 points]** You should be able to use inorder1 to construct a binary search tree, by requiring that the tree's node labels must be your desired set of search keys, in the correct order:

    ```
    inorder1(Tree,[a,b,c,d,e]).
    ```

    This is one of the coolest things about declarative programming: when you wrote inorder1, you were probably thinking about how to map deterministically from trees to lists, but you can then use it backwards, to map nondeterministically from lists to possible trees!

    However, although this is correct in principle, it will run into an infinite recursion. Why?

    Write a new version, `inorder2`, that can handle this query. It should use the same constraints as `inorder1` but in a different order. What is accomplished now by your call to `append/3`?

    Use `inorder2(Tree,[a,b,c,d,e])` under the control of `findall` to determine how many legal binary search trees there are.

    What happens if you try `inorder2` on the example of problem 11a? Why?

    (c) **[425] [6 points]** Searches in a binary tree are more efficient if the node being searched for is closer to the root of the tree. Deeper nodes take longer to find: navigating to a node at depth d requires visiting its d ancestors first. (The root is said to have depth 0, its children have depth 1, etc.)

    Write a function `total_depth/2` that computes the total depth of all non-nil nodes in a tree. (Read that carefully: Not the depth of the deepest node, but the sum of the depths of *all* nodes.) This is proportional to the amount of time it would take to search once for each element.

    Use `minimize/2` from the `branch_and_bound` library to construct `inorder3`, which constructs a tree like `inorder2` but requires it to have *minimum total depth*. (The result should turn out to be a complete binary tree, except that the deepest layer of nodes need not be left-aligned.)

    (d) **[extra credit challenge]** The above method for building efficient search trees is inefficient: it constructs *every possible* binary search tree, and from them chooses the one with minimum depth. In short, it's generate-and-compare. You can build the trees faster if you're willing to make your program less declarative.

    Write a Prolog predicate `inorder4` that constructs complete binary trees (or balanced trees, if you prefer) without using the generate-and-test method. How fast does this run, compared with your answer to the previous problem?

Note: If some elements are searched more than others, then a complete tree may no longer be optimal. If we know how often each element will be searched then we should minimize the *average depth of a searched element*. Do you have any ideas about how to do this efficiently?