Suez Canal University

Faculty of **Faculty of Computers**

Implementation and tracing of the following by two different methods: A tree dictionary and searching a maze

Student name	Amir Haytham Muhammad Salama
Faculty	Faculty of Computers and Informatics
Level	Third Level
Department	Computer Science
National ID	29906261900473
Student code	1714103022
Program	Computer Science Program
Course	Programming Logic (Prolog)

1. Introduction

As we know in real world, there is a lot of problems that has been solved with many ways, one of them is prolog. So, these problems solved with others programming languages or either in pseudocode. In my research, I will solve the tree dictionary and searching maze problems. So, let's introduce our problems. First Problem, I will explain also searching maze which is searching about something lost, so the maze will be our main idea. I mean, we will enter many path to check this object is there or not. So, let's get started. Second Problem, Tree Dicitionary Problem, in the traditional way of searching, we search sequentialy. I mean, we search in all the facts which is in the data base. Therefore, It is not something good to do it. So, I will approach a new way to do this using tree search.

2. Research items

- Introducing Searching Maze Problem Using
- Searching Maze Problem with a description of the problem in detailed
- Using two ways for Searching Maze Problem.
- Searching Maze Code Tracing.
- Introducing Sorted Tree Dictionary Using PROLOG
- Sorted Tree Dictionary Problem Using Backtracking and Cut
- Sorted Tree Dictionary Problem with a description of the problem in detailed.
- Sorted Tree Dictionary Code Tracing.

3. Research

- Searching Maze **[1,2,3,4,5]**

Suppose, we have a room and a phone is lost from me and I want to search for it. So, I will go every room and I will sign it with a number, and I will stop at every door of the room and I have an empty list. First room I enter it, I will write its number and I will search for a mobile to look for it If I don't find it, I will go to the next room. When I enter any room, I write its number in my list or make it visited in more generalized, and that means the telephone is not there. So, the idea of the application is to find the shortest path to find this moblie. At this figure, so we have a flat and I want to get the room which has the mobile, so i will get every room and I will say from => to. So, we have to cases. At first one, the phone that I want to get is existed in the room that I will visit. So the code which explains this is the following one:

```
%searching maze

d(a,b).
d(b,e).
d(b,c).
d(d,e).
d(c,d).
d(e,f).
d(g,e).

hasphone(C).

go(X,X,T).
go(X,Y,T):-(d(X,Z);d(Z,X)),\+member(Z,T) ,go(Z,Y,[Z|T]).
```

Let’s search query:

```
?- hasphone(c),go(a,c,[]).
    true
```

Let’s trace it:

```
Call: hasphone(c)
 Exit: hasphone(c)
 Call:go(a, c, [])
 Call:d(a, _5220)
 Exit:d(a, b)
 Call:lists:member(b, [])
```

```
Fail:lists:member(b, [])
Redo:go(a, c, [])
Call:go(b, c, [b])
Call:d(b, _5226)
Exit:d(b, e)
Call:lists:member(e, [b])
Fail:lists:member(e, [b])
Redo:go(b, c, [b])
Call:go(e, c, [e, b])
Call:d(e, _5232)
Exit:d(e, f)
Call:lists:member(f, [e, b])
Fail:lists:member(f, [e, b])
Redo:go(e, c, [e, b])
Call:go(f, c, [f, e, b])
Call:d(f, _5238)
Fail:d(f, _5238)
Redo:go(f, c, [f, e, b])
Call:d(_5236, f)
Exit:d(e, f)
Call:lists:member(e, [f, e, b])
Exit:lists:member(e, [f, e, b])
Fail:go(f, c, [f, e, b])
Redo:go(e, c, [e, b])
Call:d(_5230, e)
Exit:d(b, e)
Call:lists:member(b, [e, b])
Exit:lists:member(b, [e, b])
Redo:d(_5234, e)
Exit:d(d, e)
Call:lists:member(d, [e, b])
```

```
Fail:lists:member(d, [e, b])
Redo:go(e, c, [e, b])
Call:go(d, c, [d, e, b])
Call:d(d, _5238)
Exit:d(d, e)
Call:lists:member(e, [d, e, b])
Exit:lists:member(e, [d, e, b])
Call:d(_5236, d)
Exit:d(c, d)
Call:lists:member(c, [d, e, b])
Fail:lists:member(c, [d, e, b])
Redo:go(d, c, [d, e, b])
Call:go(c, c, [c, d, e, b])
Exit:go(c, c, [c, d, e, b])
Exit:go(d, c, [d, e, b])
Exit:go(e, c, [e, b])
Exit:go(b, c, [b])
Exit:go(a, c, [])
True
```

So, I have supposed that the mobile in the room that I will visit, and this room is C, and in this case if the mobile isn't existed in the room that I will visit, the result will be false. When we go to the room that we will visit, he gave me true.

Another trace to explain if the phone isn't existed in the room that I will visit:

```
d(a,b).
d(b,e).
d(b,c).
```

```
d(d,e).
d(c,d).
d(e,f).
d(g,e).

hasphone(C).

go(X,X,T).
go(X,Y,T):- d(X,Z), \+member(Z,T) ,go(Z,Y,[Z|T]).
go(X,Y,T):- d(Z,X), \+member(Z,T) ,go(Z,Y,[Z|T]).
```

Let’s search go a query :

?- hasphone(d),go(a,c,[]).

True

Let’s trace it:

```
Call:hasphone(d)
 Exit:hasphone(d)
 Call:go(a, c, [])
 Call:d(a, _5588)
 Exit:d(a, b)
 Call:lists:member(b, [])
 Fail:lists:member(b, [])
 Redo:go(a, c, [])
 Call:go(b, c, [b])
 Call:d(b, _5594)
 Exit:d(b, e)
 Call:lists:member(e, [b])
 Fail:lists:member(e, [b])
 Redo:go(b, c, [b])
 Call:go(e, c, [e, b])
 Call:d(e, _5600)
 Exit:d(e, f)
```

```
Call:lists:member(f, [e, b])
Fail:lists:member(f, [e, b])
Redo:go(e, c, [e, b])
Call:go(f, c, [f, e, b])
Call:d(f, _5606)
Fail:d(f, _5606)
Redo:go(f, c, [f, e, b])
Call:d(_5604, f)
Exit:d(e, f)
Call:lists:member(e, [f, e, b])
Exit:lists:member(e, [f, e, b])
Fail:go(f, c, [f, e, b])
Redo:go(e, c, [e, b])
Call:d(_5598, e)
Exit:d(b, e)
Call:lists:member(b, [e, b])
Exit:lists:member(b, [e, b])
Redo:d(_5602, e)
Exit:d(d, e)
Call:lists:member(d, [e, b])
Fail:lists:member(d, [e, b])
Redo:go(e, c, [e, b])
Call:go(d, c, [d, e, b])
Call:d(d, _5606)
Exit:d(d, e)
Call:lists:member(e, [d, e, b])
Exit:lists:member(e, [d, e, b])
Call:d(_5604, d)
Exit:d(c, d)
Call:lists:member(c, [d, e, b])
Fail:lists:member(c, [d, e, b])
Redo:go(d, c, [d, e, b])
Call:go(c, c, [c, d, e, b])
Exit:go(c, c, [c, d, e, b])
Exit:go(d, c, [d, e, b])
Exit:go(e, c, [e, b])
```

```
 Exit:go(b, c, [b])
 Exit:go(a, c, [])
1true
```

Some Notes for the above code:

I have done the same query that I will visit the same room ‘C’, and if I do the same query for the pervious code, the result will be **False,** Why? Because When at first one, I know that the phone will be existed in the same that I will visit. But the second code asked to do the same query, and the idea if the phone isn’t existed in the room that I will visit. So, I have added new two rules and one base case to the code:

a. go(X,X,T).
b. go(X,Y,T):- d(X,Z), \+member(Z,T) ,go(Z,Y,[Z|T]).
c. go(X,Y,T):- d(Z,X), \+member(Z,T) ,go(Z,Y,[Z|T]).

And when I do this query go(a, c, []), hasphone(c). First thing the query will match with b, Why? Because, a not equal to c, X = a, Y = c, and T will be empty because I didn’t visit any room at all, Z = b, so the first part of rule checked. Second Part of this rule, and I have built in predicit member that will check if the atom that I put it is existed in the list or not, so if it is not existed it will inform us TRUE, so I don’t visit ‘b’ so I will put in the empty list, so the new recursion rule state will be go(b, c, [b | []]) -b has been added to the list-. 2. go(b, c, [b | []]) will match with b, so X will be assigned with b, Y = c, list will have ‘b’, Z = e, and my list will have e and b, so the new recursion rule state will be go(e, c, [e | b]) -e & b have been added to the list-. 3. go(e, c, [e | b]) will match with first fact has an e, (e,f) at the left part, so X will be assigned with e, Y = c, list will have ‘b’ and ‘e’, Z = f, and my list will have e and b. So new recursion rule state will be go(f, c, [b | [f , e, b]]) -b has been added to the list-. 4. go(b, c, [b | []]) will match with b, so X will be assigned with f, Y = c, list will have ‘b, e, f’, and I will see which fact has ‘f’ as the second parameter, d(e,f), Z = e, and my list will have e, b and f. At the end, the state will be go(c,c,[f,e,b]) and it will be matched with a which is the stop condition.

Hand Tracing for what we have said:

```
?- 1. go(a, c, []), hasphone(c).
1.b   go(a,c,[]):-
1.b go(a,c,[]):-d(a,Z),Z=b
               \+member(b,[]), TRUE
                                2. go(b,c,[b|[]]).
2.b go(b,c,[b]):- d(b,Z), Z=e \+member(e,[b]), TRUE

3. go(e,c,[e|b]). FALSE
3.b go(e,c,[e,b]):- d(e,Z), Z=f
                    \+member(f,[e,b]), TRUE

4. go(f,c,[f|[e,b]]).  FALSE
4.b go(f,c,[f,e,b]):- d(f,Z), FALSE
                    \+member(Z,T),
                    go(Z,Y,[Z|T])
4.c go(f,c,[f,e,b]):- d(Z,f),  Z=e
                      \+member(e, [f,e,b]) false
                      go(Z,Y,[Z|T])
2.b go(X=b, Y=c, [f,e,b]):- d(b,Z), Z=c
                    \+member(c,[f,e,b]), true

5. go(c,d,[c|[f,e,b]])
5.a go(c,c,[f,e,b]). true
```

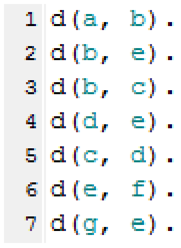
Another In detailed Explanation for Searching Maze:[6,7]

It is a dark and stormy night. As you drive down a lonely country road, your car breaks down, and you stop in front of a splendid palace. You go to the door, find it open, and begin looking for a telephone. How do you search the palace without getting lost? How do you know that you have searched every room? Also, what is the shortest path to the telephone? It is for such situations that maze-searching methods have been devised.

It is useful to keep lists of information in many computer programs, such as those for searching mazes, and search the list if certain information is needed at a later time. For example, if we decide to search the palace for a telephone, we might need to keep a list of the room numbers visited so far, so we don't go round in circles visiting the same rooms over and over again. What we do is to write down the room numbers visited on a piece of paper. Before entering a room, we check to see if its number is on our piece of paper. If it is, we disregard the room, because we had to be there previously. If the room number is not on the paper, we write down the number, and enter the room. And so on until we find the telephone. There are some refinements to be made to this method, and when we address graph searching we will do so later. But first, let's write down the steps in order, so we know what problems there are to solve:

- Go to the door of any room.
- If the room number is on our list, ignore the room and go to Step 1.
- If there are no rooms in sight, then "backtrack" through the room we went through previously, to see if there are any other rooms near it.
- Otherwise, add the room number to our list.
- Look in the room for a telephone.
- If there is no telephone, go to Step 1. Otherwise we stop, and our list has the path that we took to come to the correct room.

We shall assume that room numbers are constants, but it does not matter whether they are numbers or atoms. First, we can solve the problem of how to look up room numbers on the piece of paper by using the member predicate, representing the piece of paper as a list. Now we can get on with the problem of searching the maze. Let us consider a small example, where we are given the floor plan of a house, with letters labelling the different rooms, as shown in the following figure. Notice that gaps in the walls are meant to represent doors, and that room a is simply a representation of the space outside the house. There are doors from rooms a to b, from c to d, from f to e, and so forth. The facts about where there are doors can be represented as Prolog facts. Notice that the information about doors is not redundant. For example, although we have said that there is a door between room g and room e, we have not said that there is a door between room e and room g: we have not asserted d(e, g).



```
d(a, b).
d(b, e).
d(b, c).
d(d, e).
d(c, d).
d(e, f).
d(g, e).
```

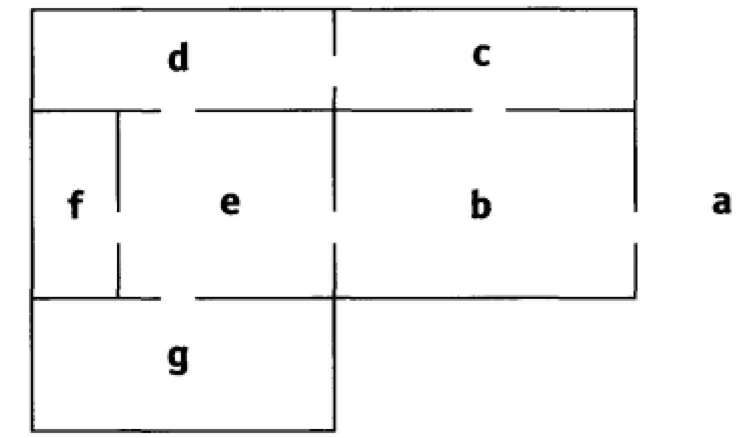


Fig. A floor plan and the program that represents it

To get around this problem of representing two-way doors, we could write a duplicate d fact for each door, reversing the arguments. Or, we could make the program
recognise that each door fact can be interpreted in two ways. This is the alternative we choose in the program that follows.
To go from one room to another, one of two situations must be recognised:
• we are in the room we want to go to, or
 • We have to go through the door and (recursively) remember these cases again.

Consider the goal go(X, Y, T), which succeeds if it is possible to go from room X to room Y. The third argument T is our piece of paper we 're holding, that has a "trail" of the room numbers we've been to so far. If we are already in room Y (this is, if X = Y) the boundary condition for going from room X to room Y will be. This is represented as the clause:

```
go(X, X, T).
```

Otherwise we'll select some adjacent space, name it Z, and see if we've ever been to it. If we do not, then we are going from Z to Y, adding Z to our list. All of this is represented as the following clause:

```
go(X, Y, T) :- d(X, I), \+ member(Z, T), go(Z, Y, [Z|T]).
```

In words, this could be interpreted as:
To "go" from X to Y, without going through the rooms on T, find a door from X to an adjacent room (Z), make sure Z is without already on the list, and escape from Z to Y, use the T-list with Z attached.

There are three ways that failures can occur in the use of this rule. First, if X has no door to anywhere. Second, if we select the door that is on the list. Third, if we cannot "go" to Y from the Z we chose because it fails deeper in the recursion. If the first target d(X, Z) fails then this use of go will cause failure. At the top level (not a recursive call), this means that there is no path from X to Y. At lower levels, it simply means we must backtrack to find a different door. As described the program treats each door as a single-way door. If we presume that the door from room a to room b is exactly the same as the door from room b to room a, then as mentioned above, we need to make that clear. There are two ways to bring this information into the system instead of presenting a duplicate fact for each d fact but with the claims reversed. The most obvious way is to add a specific rule, and to give:

```
go(X, X, T).
go(X, Y, T):- d(X, Z), \+ member(Z, T), go(Z, Y, [Z|T]).
go(X, Y, T):- d(Z, X), \+ member(Z, T), go(Z, Y, [Z|T]).
```

Or, the semicolon predicate (for disjunction) can be used:

```
go(X, X, T).
go(X, Y, T) :-
      (d(X, Z); d(Z, X)),
      \+ member(Z, T),
      go(Z, Y, [Z|T]).
```

But perhaps the clearest way is to keep the program simple, and augment the d(X,Y)
relation so that it is symmetric:

```
d(a, b). d(b, a).
d(b, e). d(e, b).
d(b, c). d(c, b).
d(d, e). d(e, d).
d(c, d). d(d, c).
d(e, f). d(f, e).
d(g, e). d(e, g).
```

Now for finding the telephone. Consider the hasphone(X) target that succeeds if a telephone is in Room X. If we want to say room g has a computer, we just write our database with it
hasphone(g).

in it. If we start at room a, one possible question we ask in order to find the way to the phone is:

```
?- go(a, X, []), hasphone(X).
```

This is a "generate and evaluate" query, which identifies potential rooms and then tests them for a computer. Another approach is first to satisfy the hasphone(X), then to see if we can go from a to X:

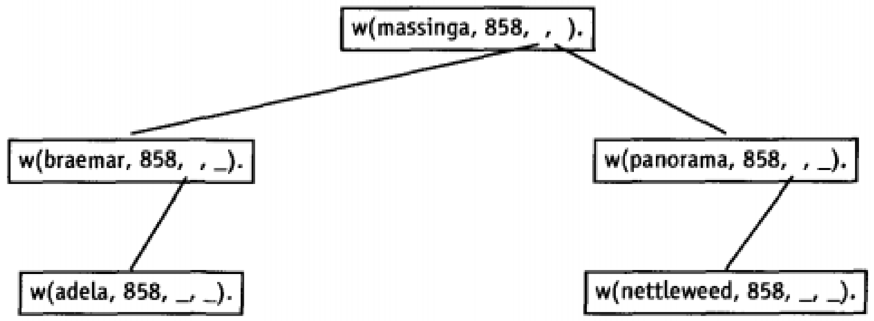
```
?- hasphone(X), go(a, X, []).
```

This approach is more effective because it means we "know" where the telephone is before we start the search. Initialising the third argument to the empty list means that we start with a clean piece of paper. This can be changed to provide variety. So the question, locate the telephone without entering rooms d and f.

```
?- hasphone(X), go(a, X, [d,f]).
```

• A Sorted Tree Dictionary:[1,2,3,4,5]

At first, if we want to do an App on PROLG, and I will put facts on this program. These facts will be horses names and its coins, and after we put these facts, I want to ask about them, so I will ask about the horse. Either if it is existed in the data base or not, or its coins values. PROLOG will search for this query in all the data base. And that is the point, if my data base has a hundreds of facts, so I will go to search them all, and that is not logical. So, the solution is, if we consider every fact as a node, and it has a left and right. So, the node in the right has a greater value than its parent, and the node in the left has less value than its parent. So, we will use the horse name as a key. Consider the following figure:

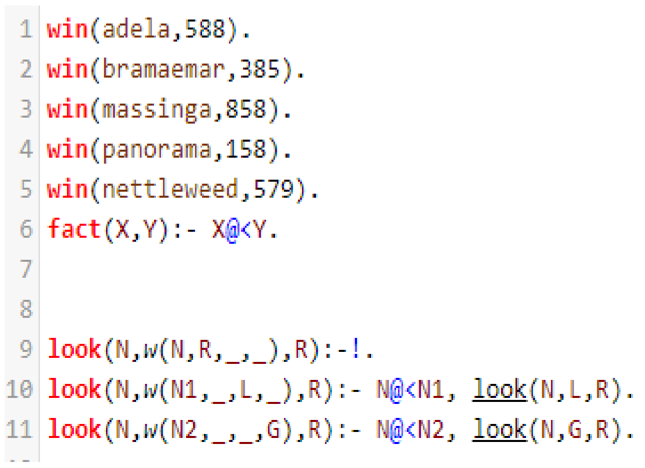


w(adela,588.(
w(bramaemar,385.(
w(massinga,858.(
w(panorama,158.(
w(nettleweed,579.(

So, we will consider the massinga as a root. like the above tree, panorama is greater than massinga, and braemar is less than massinga so it will be put in the left. The same process will be applied to the childs. To make these comparisons, we have to use a built in function fact(X,Y):- X@<Y.

It will consider every single node, and It will put in its right place. Because it works true or false, and It will give true, If for example a < b, and false if a > b. So let’s consider a structure to represent our tree.

w(massinga,300,
w(bramaemar,200, w(panorama,100
w(adela,400(_,(_,_, w(nettleweed,500,_,_),_)).



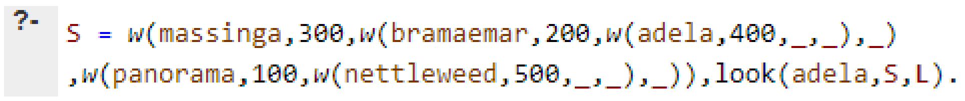
```
win(adela,588).
win(bramaemar,385).
win(massinga,858).
win(panorama,158).
win(nettleweed,579).
fact(X,Y):- X@<Y.


look(N,w(N,R,_,_),R):-!.
look(N,w(N1,_,L,_),R):- N@<N1, look(N,L,R).
look(N,w(N2,_,_,G),R):- N@<N2, look(N,G,R).
```

win(adela,588.(
win(bramaemar,385.(
win(massinga,858.(
win(panorama,158.(
win(nettleweed,579.(
fact(X,Y):- X@<Y.

look(N,w(N,R,_,_),R.!-:(
look(N,w(N1,_,L,_),R):- N@<N1, look(N,L,R.(
look(N,w(N2,_,_,G),R):- N@<N2, look(N,G,R).

at the query:
?-



```
?-  S = w(massinga,300,w(bramaemar,200,w(adela,400,_,_),_)
    ,w(panorama,100,w(nettleweed,500,_,_),_)),look(adela,S,L).
```

(_,(_,_,S = w(massinga,300,w(bramaemar,200,w(adela,400 ,w(panorama,100,w(nettleweed,500,_,_),_)),look(adela,S,L).

So, L = 400

- Tracing this code:

trace, (S = *w*(massinga,300,*w*(bramaemar,200,*w*(adela,400,_,_),_) ,*w*(panorama,100, *w*(nettleweed,500,_,_),_)),*look*(adela,S,L)). As following:

```
Call:_4514=w(massinga, 300, w(bramaemar, 200, w(adela, 400, _4590, _4592), _4582), w(panorama, 100, w(nettleweed, 500, _4610, _4612), _4602))

Exit:w(massinga, 300, w(bramaemar, 200, w(adela, 400, _4590, _4592), _4582), w(panorama, 100, w(nettleweed, 500, _4610, _4612), _4602))=w(massinga, 300, w(bramaemar, 200, w(adela, 400, _4590, _4592), _4582), w(panorama, 100, w(nettleweed, 500, _4610, _4612), _4602))

 Call:look(adela, w(massinga, 300, w(bramaemar, 200, w(adela, 400, _4590, _4592), _4582), w(panorama, 100, w(nettleweed, 500, _4610, _4612), _4602)), _4518)

Call: adela@<massinga
 Exit: adela@<massinga
 Call: look(adela, w(bramaemar, 200, w(adela, 400, _4590, _4592), _4582), _4518)
 Call: adela@<bramaemar
 Exit: adela@<bramaemar
 Call: look(adela, w(adela, 400, _4590, _4592), _4518)
 Exit: look(adela, w(adela, 400, _4590, _4592), 400)
 Exit: look(adela, w(bramaemar, 200, w(adela, 400, _4590, _4592), _4582), 400)

 Exit:
look(adela, w(massinga, 300, w(bramaemar, 200, w(adela, 400, _4590, _4592), _4582), w(panorama, 100, w(nettleweed, 500, _4610, _4612), _4602)), 400)

L = 400,

S = w(massinga, 300, w(bramaemar, 200, w(adela, 400, _1310, _1312), _1302), w(panorama, 100, w(nettleweed, 500, _1330, _1332), _1322))
```

Let’s consider this query:

(_,(_,_,S = w(massinga,300,w(bramaemar,200,w(adela,400 ,w(panorama,100,w(nettleweed,500,_,_),_)),look(adela,S,L).

So, L = 400

Let’s explain the trace:

In the traditional way of writing PROLOG code, we will search over the data base. But in the tree, let’s consider this query example. We will have something like adela, so I will ask massinga, Adela is greater or less than you? So, Adela is less than me. So, if we take a look for the tracing program. It have shown that right branch of the tree have been dropped. So, We have searched for adela. Considering this rule to trace:

a. look(N,w(N,R,_,_),R.!-:(
b. look(N,w(N1,_,L,_),R):- N@<N1, look(N,L,R.(
c. look(N,w(N2,_,_,G),R):- N@<N2, look(N,G,R).

So I asked him that if the N that I search for s the same name I search for, I will care about the result. Which result? The coins that the horse takes, at this point. I will not care about neither right, nor left. So, I have found the name that I search for. I will tell him give me the result, it should be R, and that is my stop condition. b will be if the name I search for is less than this node so far, so the greater bramch will not be considered N@<N1. And the last rule, it will be for the greater nodes, like panorama, it will be searched in the right. Last thing, we will use the cut, that means if I get the stop condition, I will stop the back tracking. Why cut technipue? To not let the program go on the backtracking for ever, or it will be cycle for infinity. In the query, I use S as a variable to put the tree on it, and L to wait the result to put on it

Another Way Explanation In Detailed:[6,7]

Suppose we wish to make associations between items of information, and retrieve them when required. For example, an ordinary dictionary associates a word with its definition, and a foreign language dictionary associates a word in one language with a word in another language. We have already seen one way to make a dictionary: with facts. If we want to make an index of the performance of horses in the British Isles during the year 1938, we could simply define facts winnings(X, Y) where X is the name of the horse, and Y is the number of guineas (a unit of currency) won by the horse. The following database of facts could serve as part of such an index:

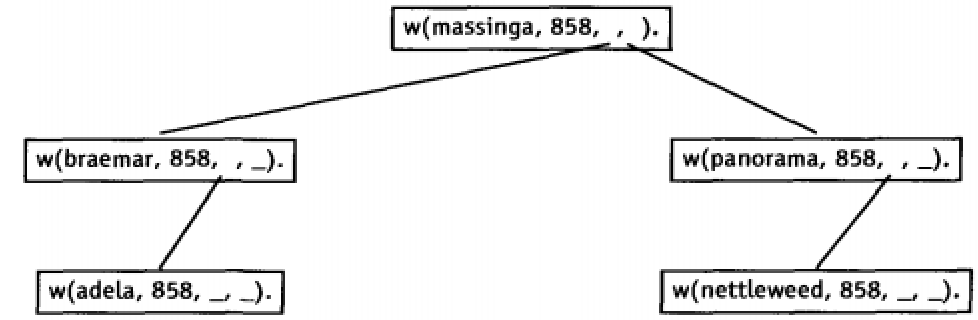
```
winnings(aban's, 582).
winnings(careful, 17).
winnings(jingling_silver, 300).
winnings(maloja, 356).
```

If we want to find out how much was won by maloja, we would simply ask the right
question, and Prolog would give us the answer:

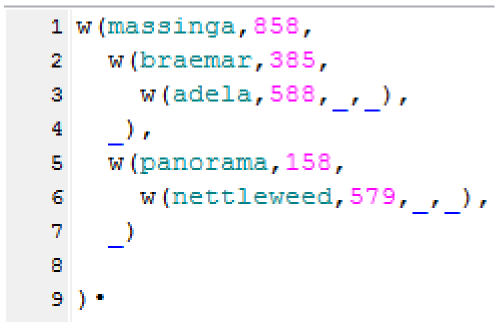
```
?- winnings(maloja, X).

X=356
```

Remember that when Prolog searches through a database to find a matching fact, it starts at the top of the database and works its way down. This means that if our dictionary database is arranged in alphabetical order, as is the one above, then Prolog will take a short amount of time to find the winnings for ablaze, and it will take longer to find the winnings for zoltan. Although Prolog can look through its database much faster than you could look through a printed index, it is silly to search the index from beginning to end if we know that the horse we are looking for is at the end. Also, although Prolog has been designed to search its database quickly, it is not always as fast as we would wish. Depending on how large your index is, and depending on how much information you have stored about each horse, Prolog might take an uncomfortably long amount of time to search the index. For these reasons and others, computer scientists have devoted much effort to finding good ways to store information, such as indices and dictionaries. Prolog itself uses some of these methods to store its own facts and rules, but it is sometimes helpful to use these methods in our programs. We shall describe one such method for representing a dictionary, called the sorted tree. The sorted tree is both an efficient way of using a dictionary, and a demonstration of how lists of structures are helpful. A sorted tree consists of some structures called nodes, where there is one node for each entry in the dictionary. Each node has four components. One of these components, called the key, is the one whose name determines its place in the dictionary (the name of the horse in our example). The other item is used to store any other information about the object involved (the winnings in our example). In addition, each node contains a tail (like the tail of a list) to a node containing a key whose name is alphabetically less than the name of the key in the node itself. Furthermore, the node contains another tail, to a node whose name is alphabetically greater than the key in the node. Let us use a structure called w(H, W, L, G) (w is an abbreviation of "winnings") where H is the name of a horse (an atom) used as the key, W is the amount of guineas won (an integer), L is a structure with a horse whose name is less than H's, and G is a structure with a horse whose name is greater than H's. If there are no structures for L and G, we will leave them uninstantiated. Given a small set of horses, the structure might look like this when written as a tree:



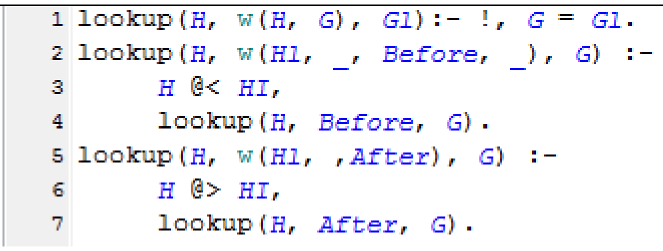
Represented as a structure in Prolog, and indented so as to illustrate the structure and not to be too wide to fit on the page, this would look like:



```
w(massinga,858,
  w(braemar,385,
    w(adela,588,_,_),
  _),
  w(panorama,158,
    w(nettleweed,579,_,_),
  _)

).
```

Now given a structure like this, we wish to "look up" names of horses in the structure to find out how many guineas they won during 1938. The structure would have the format w(H, W, L, G) as above. The boundary condition is when the name of the horse we are looking for is H. In this case, we have succeeded and need not try any alternatives. Otherwise, we must use the term comparison built-in predicates to decide which "branch" of the tree, L or G, to look up recursively. We use these principles to define the predicate lookup for which the goal lookup(H, S, G) means that horse H, when looked up in index S (a w structure), won G guineas:



```
lookup(H, w(H, G), G1):- !, G = G1.
lookup(H, w(H1, _, Before, _), G) :-
      H @< HI,
      lookup(H, Before, G).
lookup(H, w(H1, ,After), G) :-
      H @> HI,
      lookup(H, After, G).
```

If we use this predicate to search a sorted tree, in general we examine fewer horses than if we arrange them in a single list and search the list from start to finish. There is a surprising and interesting property of this lookup procedure: if we look for the name of a horse which is not in the structure, then whatever information we supply about the horse when we use lookup as a goal will be instantiated in the structure when lookup returns from its recursion. For example, the interpretation of lookup in this question

?- lookup(ruby_vintage, S, X).

is:

there is a structure, instantiated to S such that ruby_vintage is paired with X.

So, lookup is inserting new components in a partially specified structure. We can therefore use lookup repeatedly to create a dictionary. For instance,

?- lookup(abaris, X, 582), lookup(maloja, X, 356).

would instantiate X to be a sorted tree with two entries. The actual means by which lookup functions for both storing and retrieving components takes advantage of what you should know already about Prolog, so we urge you to work this out by yourself. Hint: when lookup(H, S, G) is used in a conjunction of goals, the "changes" made to S only hold over the scope of S.

4. Conclusion

So, In my research, I have given a code for PROLOG to explain the tree dictionary and the searching maze problem, and tracing the program to explain how the trace works. Also, I have attached a tracing to every single PROLOG code to explain how things are working.

5. References

1. https://www.youtube.com/watch?v=SykxWpFwMGs
2. https://www.youtube.com/watch?v=bU1vbhdFFPc
3. http://www.cs.nuim.ie/~jpower/Courses/Previous/PROLOG/
4. https://www.cpp.edu/~jrfisher/www/prolog_tutorial/pt_framer.html
5. https://www.doc.gold.ac.uk/~mas02gw/prolog_tutorial/prologpages/
6. Clocksin, W. F., & Mellish, C. S. (2012). *Programming in Prolog: Using the ISO standard*. Springer Science & Business Media.
7. Lectures: https://drive.google.com/drive/folders/1S0tzBVkGOFch7YVhPGB6yEVy5lAD1S8n?usp=sharing