



*Escola Politécnica da USP - Depto. de Enga. Mecatrônica*

# PMR-3510 Inteligência Artificial

Aula 6- Resolução de problemas por  
máquinas usando estruturas

*Prof. José Reinaldo Silva*

*reinaldo@usp.br*





## *Trabalho em grupo*

*Os grupos estão já definidos e registrados no e-disciplinas. A configuração é a seguinte:*

Grupo 1:

Alyson Akio Haro  
Alexandre Inoue  
Eduardo Kose  
Vitor Fukuda

Grupo 2:

Alex Majima  
Gabriel Ferreira  
Lucas de Angelis  
Thiago Ferraz

Grupo 3:

Gabriel Tutia  
Lucas Palopoli  
Pedro dos Santos Melo  
Samuel Monção

Grupo 4:

Gustavo Novello  
Luiz Guilherme Sabino  
Alessandro Ezequiel Junior  
Gabriel Negre

Grupo 5:

Gabriel Yida  
Vinicius Santiago  
Danilo Polidoro

Grupo 6:

Guilherme Aires de França  
Henrique Peterlevitz  
Wagner Geraldo Ferreira



## *Estrutura de um “resolvedor automático de problemas”*

*Para dotar uma máquina da capacidade de resolver problemas (ou uma classe de problemas) é preciso ter uma estrutura com os seguintes atributos:*



1. uma descrição clara do “estado inicial” ou seja das condições iniciais do problema a ser resolvido; ✓
2. uma descrição clara do objetivo ou “estado final”, de modo que seja possível saber quando (e se) o problema foi resolvido; ✓
3. em cada estágio do processo de solução saber quais os próximos estados que podem ser atingidos; ✓
4. poder escolher um (ou o melhor) caminho entre os estados acima;
5. saber que operadores (ou passos) aplicar para fazer a “transição” para um próximo estado;
6. discernir se estamos convergindo para a solução.



## Cronograma de entrega do trabalho em grupo

O cronograma de trabalho será dividido em "milestones":

Setembro 2018						
Domingo	Segunda	Terça	Quarta	Quinta	Sexta	Sábado
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29

07: Independência do Brasil 22: Início da primavera  
02 - Quarto Minguante 09 - Lua Nova 16 - Quarto Crescente 23 - Lua Cheia

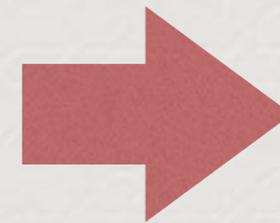
Outubro 2018						
Domingo	Segunda	Terça	Quarta	Quinta	Sexta	Sábado
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

21: Início do outono no hemisfério sul 12: Noiv. Sim. Apertadura 15: Dia dos Professores  
02 - Quarto Minguante 09 - Lua Nova 16 - Quarto Crescente 24 - Lua Cheia 31 - Quarto Minguante



## *Uma outra forma seria gerar o espaço de estados enquanto se busca a solução*

1. uma descrição clara do "estado inicial" ou seja das condições iniciais do problema a ser resolvido;
2. uma descrição clara do objetivo ou "estado final", de modo que seja possível saber quando (e se) o problema foi resolvido;
3. em cada estágio do processo de solução saber quais os próximos estados que podem ser atingidos;
4. poder escolher um (ou o melhor) caminho entre os estados acima;
5. saber que operadores (ou passos) aplicar para fazer a "transição" para um próximo estado;
6. discernir se estamos convergindo para a solução.

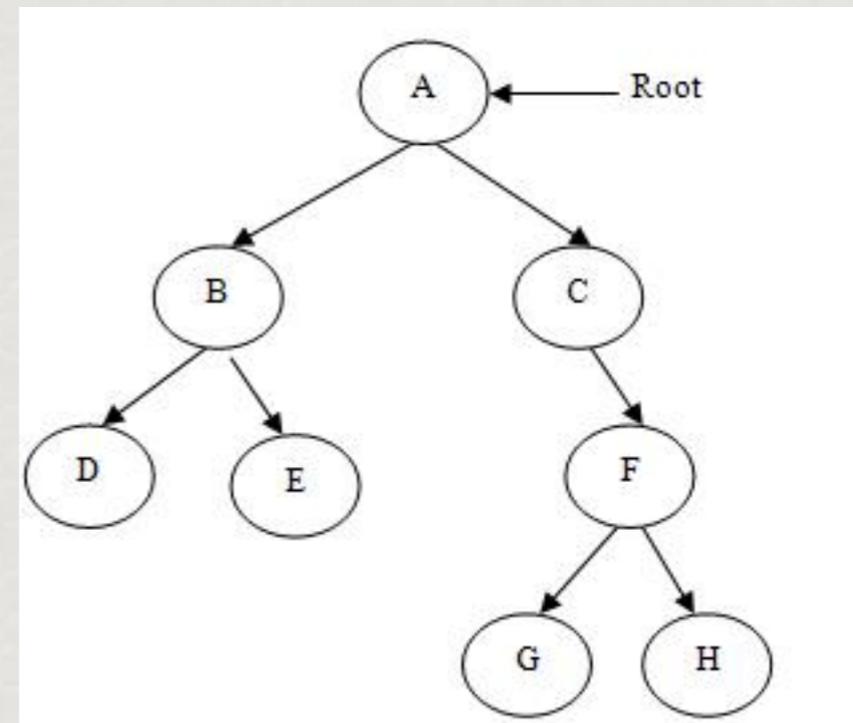


*estrutura*



## *Usando árvores como base para a solução*

*Uma opção muito interessante é modelar o espaço de estados na forma de uma árvore*





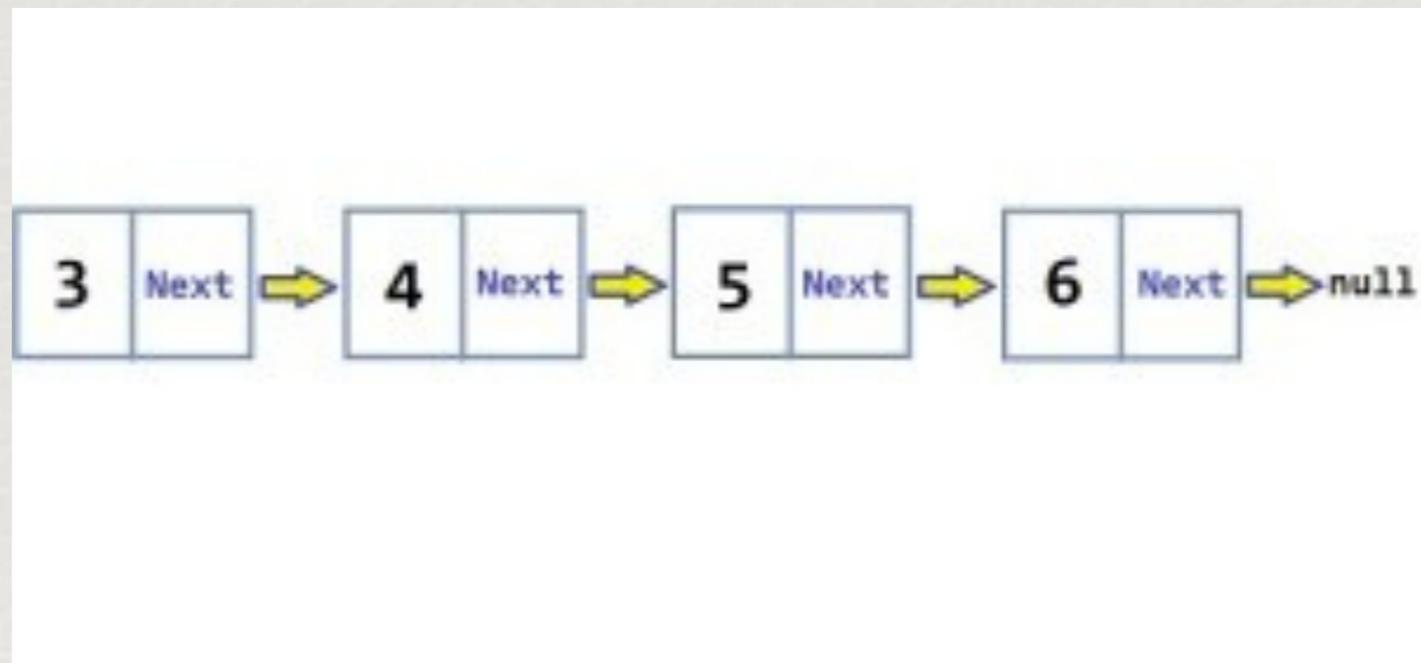
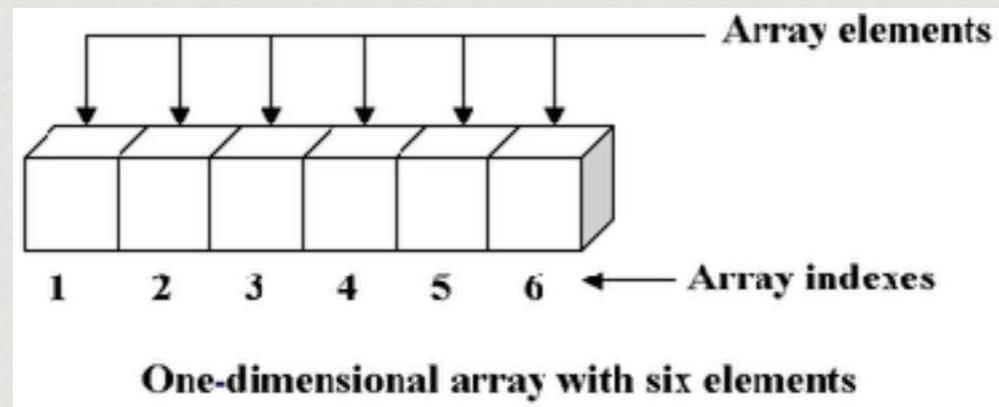
## *Estrutura de Listas*

*Uma estrutura básica em programação (lógica) é a lista. Conceitualmente uma lista é uma sequencia de registros homogênea. Normalmente estas listas são indexadas (arrays) ou direcionadas por ponteiros (listas ligadas).*

*No caso da programação em Prolog a lista segue seu conceito mais básico e é composta de uma cabeça (head) ou o primeiro elemento da lista, e o corpo (body) que é o restante da lista. Assim, o primeiro elemento é referenciado com um componente da lista e o restante como uma lista*

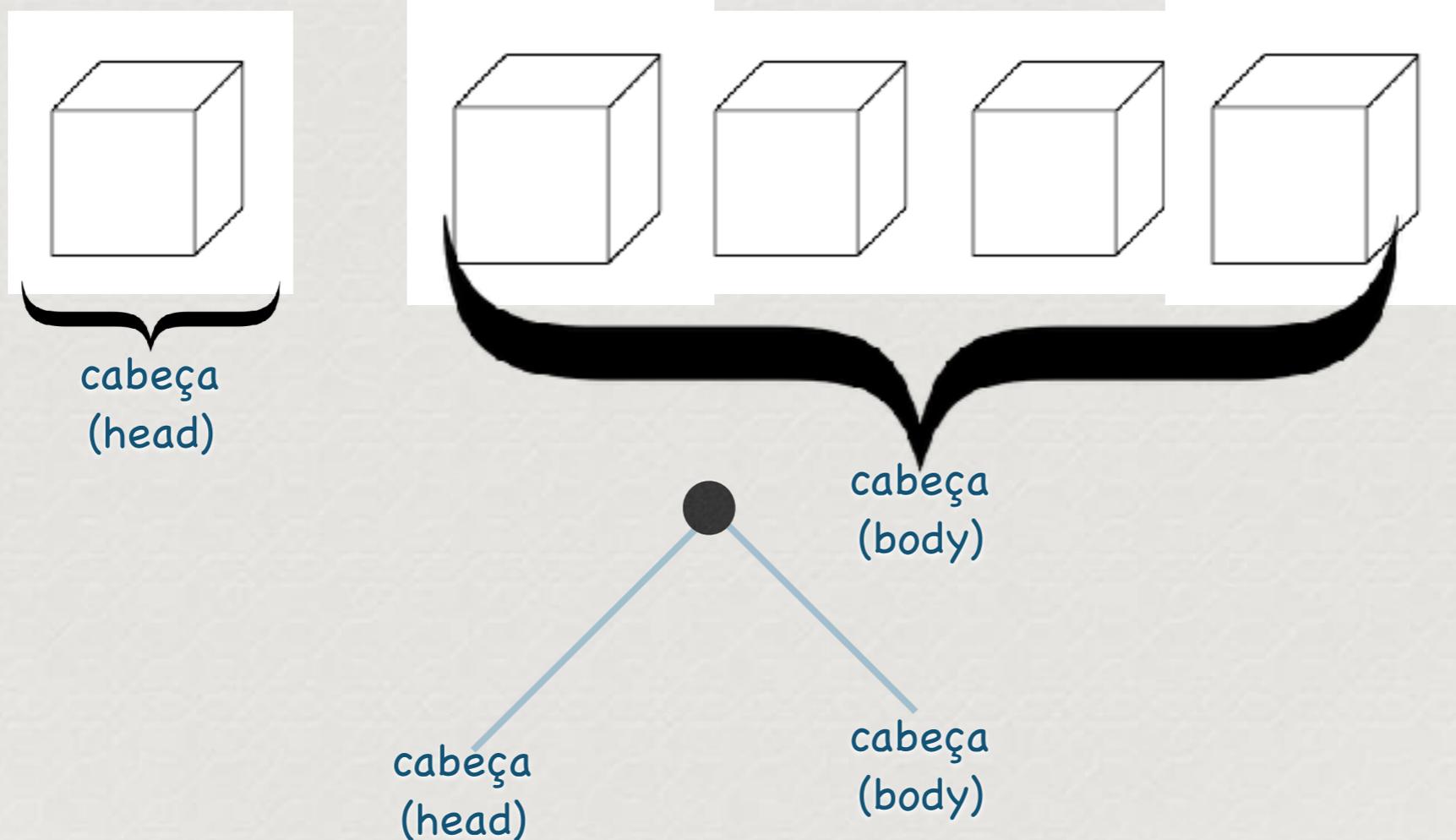


## Instancias da estrutura abstrata de lista



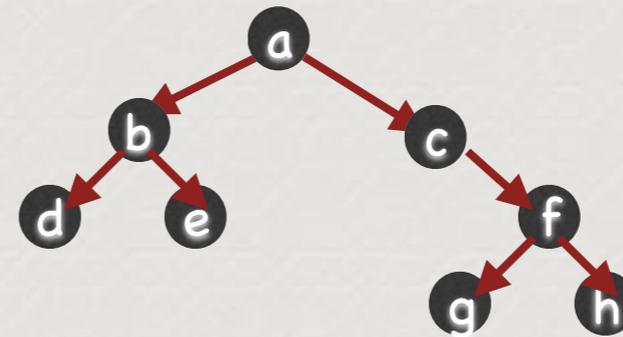
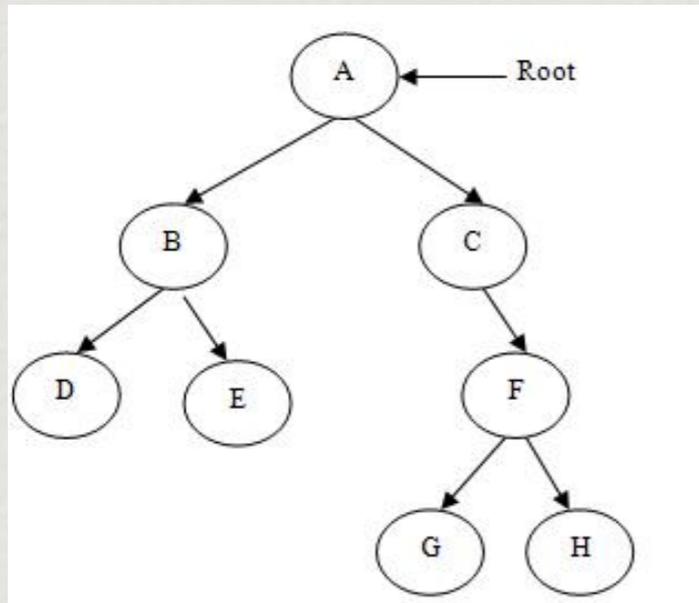


## Estrutura abstrata de lista





*Portanto é possível utilizar uma lista para implementar uma estrutura de árvore*



[a, [b, [d, e]], [c, [f, [g, h]]]]



## *Em prolog...*

<b>List</b>	<b>Head</b>	<b>Tail</b>
[a, b, c]	a	[b, c]
[]	(none)	(none)
[[the, cat], sat]	[the, cat]	[sat]
[the, [cat, sat]]	the	[[cat, sat]]
[the, [cat, sat], down]	the	[[cat, sat], down]
[X+Y, x+y]	X+Y	[x+y]

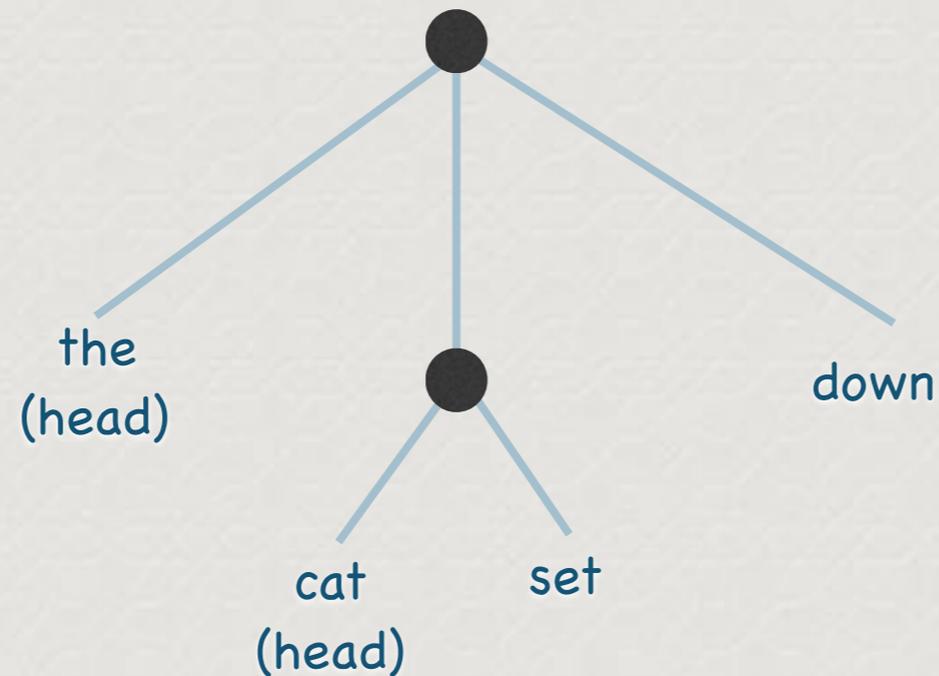
**Table 3.1.** Some lists with their head and tail

Programming in Prolog, Clocksin & Mellish



*Em prolog...*

[the, [cat, set], down]





# [www.swi-prolog.org/pldoc/man](http://www.swi-prolog.org/pldoc/man)

Did you know? You can add menus to the swipl-win.exe console in windows

Search Documentation:

## SWI Prolog Getting started quickly

- Home
- DOWNLOAD
- DOCUMENTATION
- TUTORIALS
- COMMUNITY
- USERS
- WIKI

Documentation

- Reference manual
- Overview
  - Getting started quickly
    - Starting SWI-Prolog
    - Adding rules from the console
    - Executing a query
    - Examining and modifying your p
    - Stopping Prolog
  - The user's initialisation file
  - Initialisation files and goals
  - Command line options
  - GNU Emacs Interface
  - Online help
  - Command line history
  - Reuse of top-level bindings
  - Overview of the Debugger
  - Compilation
  - Environment Control (Prolog flags)
  - An overview of hook predicates
  - Automatic loading of libraries
  - Packages: community add-ons
  - Garbage Collection
  - The SWI-Prolog syntax
  - Rational trees (cyclic terms)
  - Just-in-time clause indexing
  - Wide character support
  - System limits
  - SWI-Prolog and 64-bit machines
- Packages

### 2.1 Getting started quickly

#### 2.1.1 Starting SWI-Prolog

##### 2.1.1.1 Starting SWI-Prolog on Unix

By default, SWI-Prolog is installed as 'swipl'. The command line arguments of SWI-Prolog itself and its utility programs are documented using standard Unix **man** pages. SWI-Prolog is normally operated as an interactive application simply by starting the program:

```
swipl
Welcome to SWI-Prolog ...
...
?-
```

After starting Prolog, one normally loads a program into it using [consult/1](#), which may be abbreviated by putting the name of the program file between square brackets. The following goal loads the file [likes.pl](#) containing clauses for the predicates `likes/2`:

```
?- [likes].
true.
?-
```

Alternatively, the source file may be given as command line arguments:

```
swipl likes.pl
Welcome to SWI-Prolog ...
...
?-
```



## [lpn.swi-prolog.org/lpnpage.php](http://lpn.swi-prolog.org/lpnpage.php)

This version of Learn Prolog Now! embodies [SWISH](#), [SWI-Prolog](#) for SHaring. The current version rewrites the Learn Prolog Now! HTML on the fly, recognising source code and example queries. It is not yet good at recognising the relations between source code fragments and queries. Also Learn Prolog Now! needs some updating to be more compatible with SWI-Prolog. All sources are on GitHub:

Learn Prolog Now! Fork 44    LPN SWISH Proxy Fork 7    SWISH Fork 62

# Learn Prolog Now!

by Patrick Blackburn, Johan Bos, and Kristina Striegnitz

- LPN! Home
- Free Online Version
- Paperback English
- Paperback Français
- Teaching Prolog
- Prolog Implementations
- Prolog Manuals
- Prolog Links
- Thanks!
- Contact us

[ next ] [ prev ] [ prev-tail ] [ | tail ] [ up ]

## Chapter 4 Lists

This chapter has three main goals:

1. To introduce lists, an important recursive data structure often used in Prolog programming.
2. To define the `member/2` predicate, a fundamental Prolog tool for manipulating lists.
3. To introduce the idea of recursing down lists.

- [4.1 Lists](#)
- [4.2 Member](#)
- [4.3 Recursing down Lists](#)
- [4.4 Exercises](#)
- [4.5 Practical Session](#)

[ next ] [ prev ] [ prev-tail ] [ | front ] [ up ]

© 2008-2012 [Patrick Blackburn](#), [Johan Bos](#), [Kristina Striegnitz](#)



*Se desejamos ter um procedimento que checa se um dado elemento pertence à lista...*

```
member(X, [X|T]).  
member(X, [_|T]) :- member(X, T).
```



## Exemplo...

The screenshot shows the SWISH Prolog IDE interface. The browser address bar displays `swish.swi-prolog.org/example/examples.swinb`. The SWISH logo and menu items (File, Edit, Examples, Help) are visible at the top. The main editor area contains the text: `1 Your Prolog rules and facts go here ...`. The right-hand pane shows the execution results for two queries:

```
trace, Grupo3=[gabriel,lucas,pedro,samuel], member(pedro,Grupo3).  
Call: _3838=[gabriel,lucas,pedro,samuel]  
Exit: [gabriel,lucas,pedro,samuel]=[gabriel,lucas,pedro,samuel]  
Call: !isomember(pedro,[gabriel,lucas,pedro,samuel])  
Exit: !isomember(pedro,[gabriel,lucas,pedro,samuel])  
Grupo3 = [gabriel,lucas,pedro,samuel]  
  
trace, Grupo3=[gabriel,lucas,pedro,samuel], member(slyson,Grupo3).  
Call: _3838=[gabriel,lucas,pedro,samuel]  
Exit: [gabriel,lucas,pedro,samuel]=[gabriel,lucas,pedro,samuel]  
Call: !isomember(slyson,[gabriel,lucas,pedro,samuel])  
Fail: !isomember(slyson,[gabriel,lucas,pedro,samuel])  
false  
  
?- trace, Grupo3=[gabriel,lucas,pedro,samuel], number(slyson,Grupo3).
```

At the bottom of the results pane, there are buttons for "Examples", "History", and "Solutions", and a checkbox for "table results" with a "Run" button.



## *operador interno append/3*

*Uma operação básica com listas é juntar duas listas em uma terceira lista.  
O operador append/3 pode fazer isso facilmente...*

```
append([], L, L).  
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
```



## Exemplo...

SWISH File Edit Examples Help

Program x prolog\_tutorials x examples x +

1 Your Prolog rules and facts go here ...

trace, Grupo5=[gabriel\_y,vinicius,danilo],append([guilherme,wagner],Grupo5,NewGroup).

Call: \_3884-[gabriel\_y,vinicius,danilo]  
Exit: [gabriel\_y,vinicius,danilo]-[gabriel\_y,vinicius,danilo]  
Call: lists\_append([guilherme,wagner],[gabriel\_y,vinicius,danilo],\_3880)  
Exit: lists\_append([guilherme,wagner],[gabriel\_y,vinicius,danilo],[guilherme,wagner,gabriel\_y,vinicius,danilo])

Grupo5 = [gabriel\_y,vinicius,danilo].  
NewGroup = [guilherme,wagner,gabriel\_y,vinicius,danilo]

?- trace, Grupo5=[gabriel\_y,vinicius,danilo],append([guilherme,wagner],Grupo5,NewGroup).

Examples History Solutions

table results Run



## *Voltando ao projeto dos grupos...*

*Não devemos de forma alguma antecipar a programação do algoritmo para resolver o jogo de tiles antes de seguir o processo (aqui colocado como uma “boa prática”, e mostrado na transparência de no. 3 esta aula.*

*Entretanto, para que não pareça a “missão impossível”, bem acima do “nível do curso”, vamos já “baixar à terra”, e colocar alguns procedimentos que deverão ser utilizados e que não constituem o núcleo do projeto.*

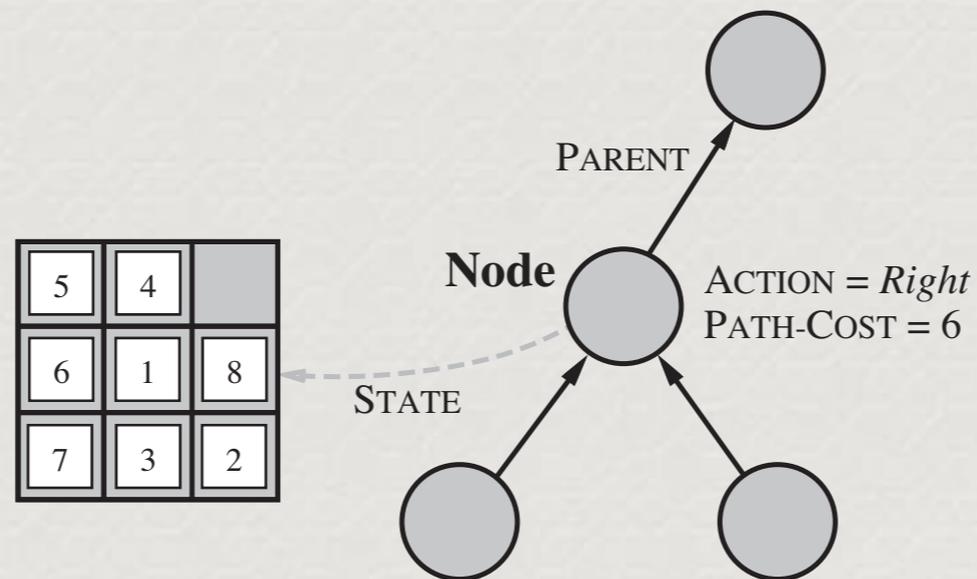
*Um destes procedimentos é justamente a entrada de dados: como poderemos fazer um sistema Prolog (especificamente o Swich) ler uma tabela de tiles.*

*Utilizaremos um procedimento simples, baseado em listas como vimos anteriormente.*



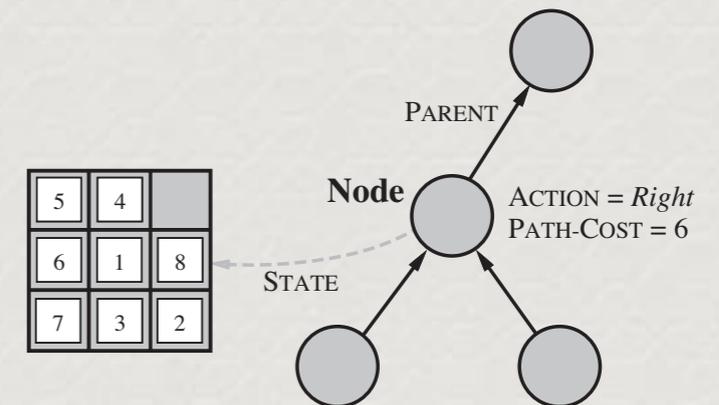
## A tabela de tiles

*A tabela de tiles tem NOVE posições e o conteúdo de cada uma destas posições é um inteiro de zero a oito, onde o zero significa o tile para onde se pode mover os demais vizinhos.*





*O programa em prolog a seguir insere uma tabela de tiles de entrada. Não é exatamente uma “interface”, não tem correção de entradas erradas, somente preserva o número de dados de entrada.*



```
add_tile(X,Y,K) :- X < 10, read(Z), append(Y, [(X,Z)], L), N is X + 1,
    add_tile(N, L, K).
add_tile(10, X, X).
in_data(Index, Board) :- Index=1, Tile_board=[], add_tile(Index, Tile_board, Board).
```



swish.swi-prolog.org/example/iris.swinb

Bookmarks Bar (Chro) | Bookmarks | Artificial Intelligence | Notícias | Popular | Save to Mendeley

**SWISH** File Edit Examples Help

124 Users online

Search

Program

```

1
2 add_tile(X,Y,S):- X < 10, read(S), append(Y, [(X,S)],L), X is X + 1,
3   add_tile(N, L, X).
4 add_tile(10, X, X).
5 in_data(Index, Board):- Index=1, Tile_board=[], add_tile(Index, Tile_board, Board).
6

```

```

Exit: read(-)
Call: !ns:append![(1,5), (2,4), (3,0), (4,6), (5,1), (6,8), (7,7), (8,3)], ![(9,2)], ![_B574]
Exit: !ns:append![(1,5), (2,4), (3,0), (4,6), (5,1), (6,8), (7,7), (8,3)], ![(9,2)], ![(1,5), (2,4), (3,0), (4,6), (5,1), (6,8), (7,7), (8,3), (9,2)]
Call: !_6624 is 9+1
Exit: 10 is 9+1
Call: add_tile(10, [(1,5), (2,4), (3,0), (4,6), (5,1), (6,8), (7,7), (8,3), (9,2)], !_6416)
Call: 10<10
Fail: 10<10
Redo: add_tile(10, [(1,5), (2,4), (3,0), (4,6), (5,1), (6,8), (7,7), (8,3), (9,2)], !_6416)
Exit: add_tile(10, [(1,5), (2,4), (3,0), (4,6), (5,1), (6,8), (7,7), (8,3), (9,2)], [(1,5), (2,4), (3,0), (4,6), (5,1), (6,8), (7,7), (8,3), (9,2)])
Exit: add_tile(9, [(1,5), (2,4), (3,0), (4,6), (5,1), (6,8), (7,7), (8,3)], [(1,5), (2,4), (3,0), (4,6), (5,1), (6,8), (7,7), (8,3), (9,2)])
Exit: add_tile(8, [(1,5), (2,4), (3,0), (4,6), (5,1), (6,8), (7,7)], [(1,5), (2,4), (3,0), (4,6), (5,1), (6,8), (7,7), (8,3), (9,2)])
Exit: add_tile(7, [(1,5), (2,4), (3,0), (4,6), (5,1), (6,8)], [(1,5), (2,4), (3,0), (4,6), (5,1), (6,8), (7,7), (8,3), (9,2)])
Exit: add_tile(6, [(1,5), (2,4), (3,0), (4,6), (5,1)], [(1,5), (2,4), (3,0), (4,6), (5,1), (6,8), (7,7), (8,3), (9,2)])
Exit: add_tile(5, [(1,5), (2,4), (3,0), (4,6)], [(1,5), (2,4), (3,0), (4,6), (5,1), (6,8), (7,7), (8,3), (9,2)])
Exit: add_tile(4, [(1,5), (2,4), (3,0)], [(1,5), (2,4), (3,0), (4,6), (5,1), (6,8), (7,7), (8,3), (9,2)])
Exit: add_tile(3, [(1,5), (2,4)], [(1,5), (2,4), (3,0), (4,6), (5,1), (6,8), (7,7), (8,3), (9,2)])
Exit: add_tile(2, [(1,5)], [(1,5), (2,4), (3,0), (4,6), (5,1), (6,8), (7,7), (8,3), (9,2)])
Exit: add_tile(1, [], [(1,5), (2,4), (3,0), (4,6), (5,1), (6,8), (7,7), (8,3), (9,2)])
Exit: in_data(1, [(1,5), (2,4), (3,0), (4,6), (5,1), (6,8), (7,7), (8,3), (9,2)])
X = 1,
Y = [(1,5), (2,4), (3,0), (4,6), (5,1), (6,8), (7,7), (8,3), (9,2)]
0.870 seconds cpu time

```

7- trace, in\_data(X,Y).

Examples History Solutions

table results **Run!**

5	4	
6	1	8
7	3	2



## *Algoritmos de busca*

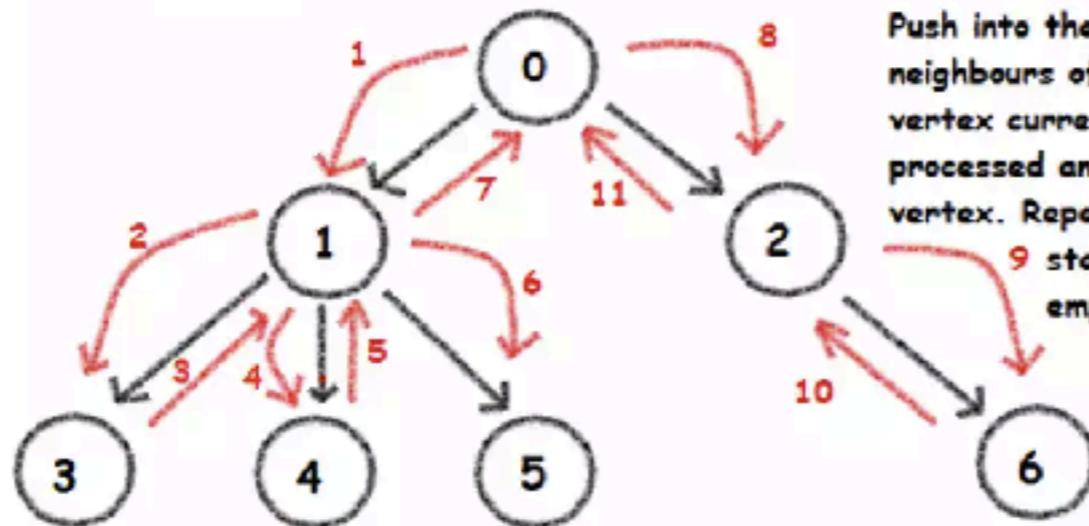
*Busca não informada - quando todos os nós gerados são igualmente promissores, ou não se tem informação sobre o seu potencial: busca em profundidade, busca em largura, busca de custo uniforme*

*Busca informada - quando conhecimento heurístico pode ser levantado que distingue entre os nós gerados em um mesmo nível da árvore.*



# Busca em profundidade

Red arrows indicate the order of search.



Push into the stack the neighbours of the vertex currently being processed and Pop the vertex. Repeat until stack is not empty.

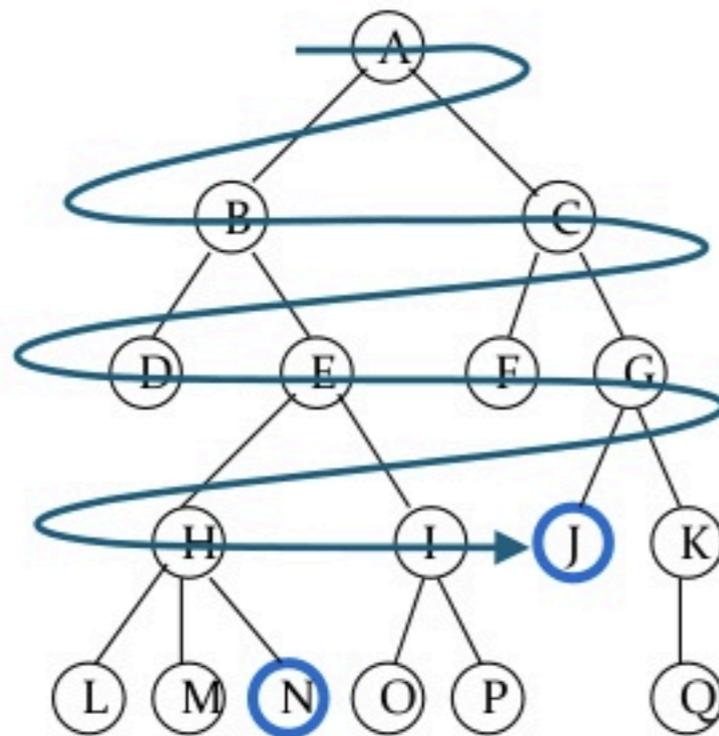
Vertex	Stack
	0
0	1, 2
1	3, 4, 5, 2
3	4, 5, 2
4	5, 2
5	2
2	6
6	

Depth First Search



## Busca em largura

### Breadth-first searching[1]



- A breadth-first search (BFS) explores nodes nearest the root before exploring nodes further away
- For example, after searching A, then B, then C, the search proceeds with D, E, F, G
- Node are explored in the order A B C D E F G H I J K L M N O P Q
- J will be found before N



*Nas próximas aulas veremos*

- i) busca em profundidade limitada;
- ii) busca de aprofundamento interativo;
- iii) busca bi-direcional;



*Até a próxima aula!*