Operações em Estruturas de Dados

Inteligência Artificial

- Nesta aula são descritas algumas operações comuns em estruturas de dados freqüentemente utilizadas em IA
  - Otimização de Recursão no Final (Acumuladores)
  - Ordenação
  - Árvores Binárias
  - Grafos

José Augusto Baranauskas
Departamento de Física e Matemática – FFCLRP-USP
E-mail: augusto@usp.br
URL: http://dfm.fmrp.usp.br/~augusto

Otimização de Recursão no Final

- Em Prolog, não há construções iterativas e o conceito mais geral da recursão é utilizado para especificar ambos algoritmos iterativos e recursivos
- A principal vantagem da iteração sobre a recursão é eficiência (em geral, em espaço)
- A recursão exige que uma pilha seja mantida para armazenar cada chamada recursiva ainda não terminada
- Uma computação recursiva envolvendo n chamadas recursivas requer espaço linear em n, ou seja, O(n)
- Por outro lado, um programa iterativo utiliza uma quantidade fixa de memória, independente do número de iterações, ou seja, O(1)
- Prolog transforma uma cláusula (programa) recursivo com uma única chamada recursiva no final em um programa iterativo

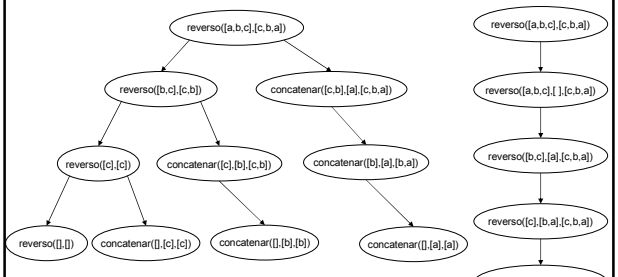
Acumuladores

```
% versão sem acumulador
somaLista([],0).
somaLista([X|T],Soma) :-
  somaLista(T,SomaT),
  Soma is SomaT + X.

% versão com acumulador
somaLista(Lista,Soma) :-
  somaLista(Lista,0,Soma).
somaLista([],Soma,Soma).
somaLista([X|T],SomaParcial,SomaTotal) :-
  NovaParcial is SomaParcial + X,
  somaLista(T,NovaParcial,SomaTotal).
```


Acumuladores

```
% versão sem acumulador
reverso([],[]).
reverso([X|T],Reverso) :-
  reverso(T,Treverso),
  concatenar(Treverso,[X],Reverso).

% versão com acumulador
reverso(Lista,Reverso) :-
  reverso(Lista,[],Reverso).
reverso([],R,R).
reverso([X|T],ParteReversa,TotalReverso) :-
  reverso(T,[X|ParteReversa],TotalReverso).
```


Eficiência



Em geral, o tamanho da árvore de prova da versão sem acumulador da relação **reverso** é quadrático ao número de elementos na lista enquanto a versão com acumulador é linear

Ordenação

- Ordenação é uma operação fundamental em programação e pode ser realizada em listas
- Para tanto, vamos assumir que existe uma relação de ordem gt(X,Y) que é verdadeiro se X é maior do que Y
  - Se os elementos da lista são números, a relação gt/2 pode ser definida como:
    - `gt(X,Y) :- X > Y.`
  - Se os elementos da lista são átomos ou estruturas, a relação gt/2 pode ser definida como:
    - `gt(X,Y) :- X @> Y.`

Inserção Direta

- insertsort(+L,?ListaOrdenada)
- Para ordenar uma lista não vazia L = [X|T]
  - Ordene a cauda T de L
  - Insira a cabeça, X, de L na cauda ordenada na posição tal que a lista resultante seja ordenada. O resultado é a lista toda ordenada

```
insertsort([],[]).
insertsort([X|T],Sorted) :-
  insertsort(T,SortedT),
  insert(X,SortedT,Sorted).

insert(X,[Y|T],[Y|T1]) :-
  gt(X,Y), !,
  insert(X,T,T1).
insert(X,T,[X|T]).
```


Borbulhamento

- bubblesort(+L,?ListaOrdenada)
- Para ordenar uma lista L
  - Encontre dois elementos adjacentes, X e Y em L tais que gt(X,Y) e troque X em Y em L, obtendo L1; então ordene L1
  - Se não há um par de elementos adjacentes, X e Y, em L tais que gt(X,Y) então L já está ordenada

```
bubblesort(L,Sorted) :-
   swap(L,L1), !,
   bubblesort(L1,Sorted).
bubblesort(L,L).

swap([X,Y|T],[Y,X|T]) :-
   gt(X,Y).
swap([Z|T],[Z|T1]) :-
   swap(T,T1).
```


Quicksort

- quicksort(+L,?ListaOrdenada)
- Para ordenar uma lista L não vazia
  - Retire algum elemento X de L e particione o resto de L em duas listas, chamadas Small e Big de forma que todos os elementos de L maiores que X fiquem em Big e todos os outros em Small
  - Ordene Small, obtendo SortedSmall
  - Ordene Big, obtendo SortedBig
  - A lista toda ordenada é a concatenação de SortedSMall com [X | SortedBig]

```
quicksort([],[]).
quicksort([X|Tail],Sorted) :-
  particao(X,Tail,Small,Big),
  quicksort(Small,SortedSmall),
  quicksort(Big,SortedBig),
  concatenar(SortedSmall,
       [X|SortedBig],Sorted).

particao(X,[],[],[]).
particao(X,[Y|T],[Y|Small],Big):-
  gt(X,Y), !,
  particao(X,T,Small,Big).
particao(X,[Y|T],Small,[Y|Big]):-
  particao(X,T,Small,Big).
```

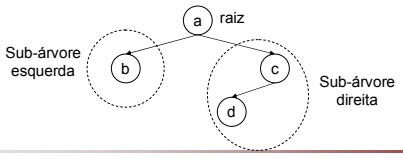

Mergesort

- mergesort(+L,?ListaOrdenada)
- Para ordenar uma lista L
  - Divida L em duas listas L1 e L2 de tamanho aproximadamente igual
  - Ordene L1 e L2 obtendo Sorted1 e Sorted2
  - Intercale Sorted1 e Sorted2 obtendo a lista ordenada

```
mergesort([],[]) :- !.
mergesort([X],[X]) :- !.
mergesort(L,Sorted) :-
  divide(L,L1,L2),
  mergesort(L1,Sorted1),
  mergesort(L2,Sorted2),
  intercala(Sorted1,Sorted2,Sorted).

divide([],[],[]).
divide([X],[X],[]).
divide([X,Y|L],[X|L1],[Y|L2]) :-
  divide(L,L1,L2).

intercala([],L,L) :- !.
intercala(L,[],L) :- !.
intercala([X|T1],[Y|T2],[Y|T3]) :-
  gt(X,Y), !,
  intercala([X|T1],T2,T3).
intercala([X|T1],L,[X|T3]) :-
  intercala(T1,L,T3).
```

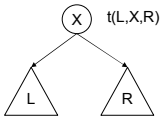

Árvores Binárias

- Uma árvore binária é uma estrutura que é ou vazia ou possui 3 componentes:
  - Uma raiz
  - Uma sub-árvore esquerda
  - Uma sub-árvore direita
- A raiz pode ser qualquer objeto, mas as sub-árvores devem ser árvores binárias também



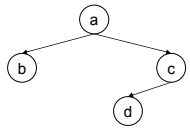
Árvores Binárias

- Podemos representar uma AB da seguinte forma:
  - O átomo **nil** representa uma árvore vazia
  - O functor **t** representa uma árvore com raiz **X**, sub-árvore esquerda **L** e sub-árvore direita **R** da seguinte forma **t(L,X,R)**



- Nesta representação, a árvore abaixo é representada como:

t(t(nil,b,nil), a, t(t(nil, d, nil),c,nil))



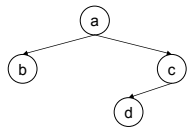
Árvores Binárias

- Vamos definir o predicado in(X,T) que verifica se o elemento X encontra-se na árvore T
- X está na árvore T se:
  - A raiz de T é X ou
  - X está na sub-árvore esquerda de T ou
  - X está na sub-árvore direita de T

```
in(X,t(_,X,_)).
in(X,t(L,_,_)) :-
  in(X,L).
in(X,t(_,_,R)) :-
  in(X,R).
```


Árvores Binárias

```
?- T = t(t(nil,b,nil),a,
         t(t(nil, d, nil),c,nil)
        ),
  in(X,T).
X = a;
X = b;
X = c;
X = d
```

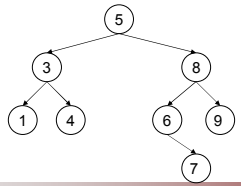


Árvores Binárias

- A meta **in(a,T)** é satisfeita imediatamente pela primeira cláusula do predicado **in/2**
- Por outro lado, a meta **in(d,T)** é satisfeita depois de várias chamadas recursivas antes que **d** seja encontrado
- De modo análogo, a meta **in(e,T)** falhará somente depois de buscar na árvore toda
- Esta ineficiência pode ser solucionada se houver uma relação de ordem entre os dados

Árvores Binárias de Busca

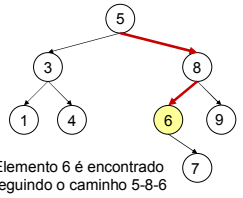
- Uma árvore t(L,X,R) é uma árvore binária de busca (ABB) se:
  - Todos os nós na sub-árvore esquerda L são menores que X
  - Todos os nós na sub-árvore direita R são maiores que X
  - As sub-árvores L e R são também ABB
- ABB também são conhecidas como dicionários binários



Árvores Binárias de Busca

- A vantagem da ABB é que sempre é suficiente realizar a busca em no máximo uma das sub-árvores
- Para encontrar um elemento X na árvore T
  - Se X é a raiz de T então X foi encontrado, caso contrário
  - Se X é menor que a raiz de T então procure X na sub-árvore esquerda de T, caso contrário
  - Procure X na sub-árvore direita de T
  - Se T é vazia então a busca falha

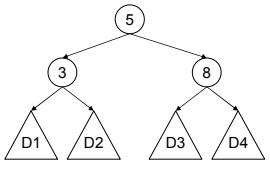
```
% in(+X,+T) procura X na ABB T
in(X,t(_,X,_)).
in(X,t(Left,Root,_)) :-
  gt(Root,X),
  in(X,Left).
in(X,t(_,Root,Right)) :-
  gt(X,Root),
  in(X,Right).
```



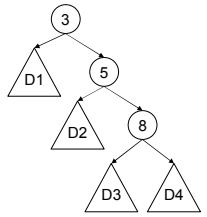
Elemento 6 é encontrado seguindo o caminho 5-8-6

Árvores Binárias de Busca

```
?- in(5,T), in(3,T),
   in(8,T).
T = t(t(D1,3,D2), 5,
      t(D3,8,D4))
```



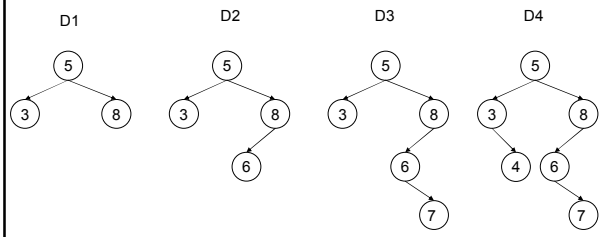

```
?- in(3,T), in(5,T),
   in(8,T).
T = t(D1, 3,
      t(D2,5,t(D3,8,D4)))
```



Inserção em ABB

- Para adicionar um nó X a uma ABB T
  - Adicionar X a uma árvore vazia resulta em **t(nil,X,nil)**
  - Se a raiz de T é maior que X então insira X na sub-árvore esquerda de T, caso contrário insira X na sub-árvore direita de T

Inserção em ABB (nó folha)

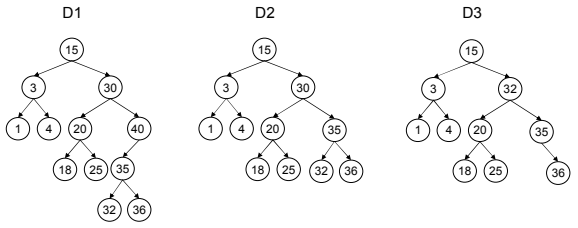


add(D1,6,D2) add(D2,7,D3) add(D3,4,D4)

Inserção em ABB

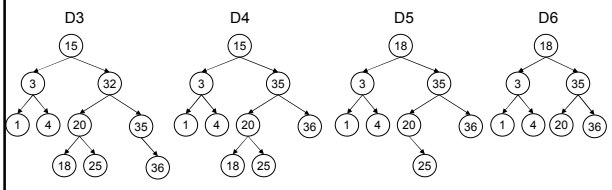
```
% add(+Tree,+X,?NewTree) insere X na árvore Tree
% gerando árvore NewTree

add(nil,X,t(nil,X,nil)).       % insere X numa árvore vazia
add(t(L,Y,R),X,t(Ll,Y,R)) :-  % insere X na sub-árvore esquerda
  gt(Y,X),
  add(L,X,Ll).
add(t(L,Y,R),X,t(L,Y,Rl)) :-  % insere X na sub-árvore direita
  gt(X,Y),
  add(R,X,Rl).
```


Remoção em ABB



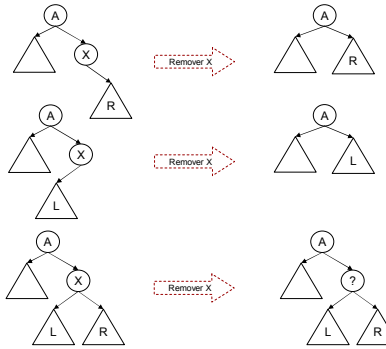
del(D1,40,D2) del(D2,30,D3)

Remoção em ABB

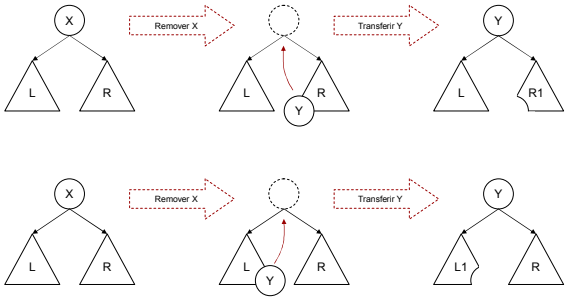


del(D3,32,D4) del(D4,15,D5) del(D5,25,D6)

Remoção em ABB



Remoção em ABB

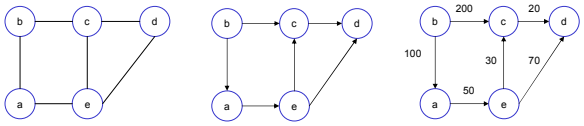


Remoção em ABB

```
% del(Tree,X,NewTree) remove X na árvore Tree gerando árvore NewTree
del(t(nil,X,R),X,R) :- !.
del(t(L,X,nil),X,L) :- !.
del(t(L,X,R),X,t(L,Y,R1)) :-
  delmin(R,Y,R1), !.
del(t(L,Root,R),X,t(L1,Root,R)) :-
  gt(Root,X),
  del(L,X,L1).
del(t(L,Root,R),X,t(L,Root,R1)) :-
  gt(X,Root),
  del(R,X,R1).

% delmin(Tree,Y,NewTree) remove elemento mínimo Y da árvore Tree gerando
% árvore NewTree
delmin(t(nil,Y,R),Y,R).
delmin(t(L,Root,R),Y,t(L1,Root,R)) :-
  delmin(L,Y,L1).
```

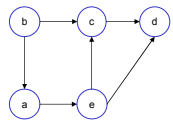

Grafos

- Um grafo G(V,E) é definido por um conjunto V de vértices (ou nós) e um conjunto E de arestas, onde cada aresta é um par de nós
- As arestas podem ser orientadas bem como possuírem um custo ou rótulo associado

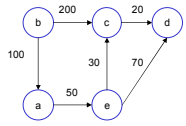


Grafos

- Grafos podem ser representados de diversas formas, por exemplo, considerado somente as arestas (orientadas):
  - `aresta(b,a).`
  - `aresta(a,e).`
  - `aresta(b,c).`
  - `aresta(c,d).`
  - `aresta(e,c).`
  - `aresta(e,d).`

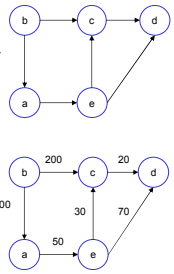


- Ou considerando informações adicionais
  - `aresta(b,a,100).`
  - `aresta(a,e,50).`
  - `aresta(b,c,200).`
  - `aresta(c,d,20).`
  - `aresta(e,c,30).`
  - `aresta(e,d,70).`



Grafos

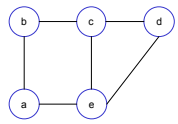
- Outro método é representar todo o grafo como um objeto composto por dois conjuntos: nós e arestas
- Cada conjunto é representado como uma lista
- Vamos escolher o functor **grafo** para combinar ambos conjuntos e o functor **a** para representar uma aresta
  - `G1 = grafo([a,b,c,d,e], [a(b,a), a(a,e), a(b,c), a(c,d), a(e,c), a(e,d)] )`
  - `G2 = grafo([a,b,c,d,e], [a(b,a,100), a(a,e,50), a(b,c,200), a(c,d,20), a(e,c,30), a(e,d,70)] )`



Caminhos em Grafos

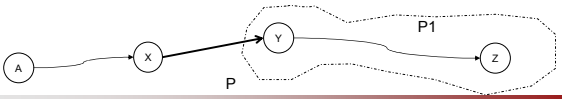
- Seja G é um grafo e A e Z dois nós de G
- Um **caminho** acíclico entre A e Z de G será encontrado pela relação **path(A,Z,G,P)** onde P é uma lista de nós no caminho
- Exemplo

```
?- path(a,d,G,P).
P = [a,e,d];
P = [a,e,c,d];
P = [a,b,c,d];
P = [a,b,c,e,d]
```



Caminhos em Grafos

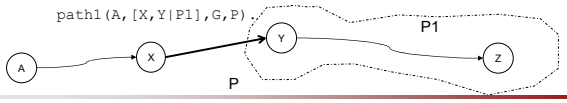
- Para encontrar um caminho acíclico P entre A e Z de G (sem custos)
  - Se A = Z então P = [A], caso contrário
  - Encontre um caminho acíclico P1 de algum Y até Z e encontre um caminho de A até Y evitando os nós em P1



Caminhos em Grafos

- Para encontrar um caminho acíclico P entre A e Z de G (sem custos)
  - Se A = Z então P = [A], caso contrário
  - Encontre um caminho acíclico P1 de algum Y até Z e encontre um caminho de A até Y evitando os nós em P1
- Isto requer outra relação **path1(A,P1,G,P)**

```
path(A,Z,G,P) :-
  path1(A,[Z],G,P).
path1(A,[A|P1],_,[A|P1]).
path1(A,[Y|P1],G,P) :-
  adjacente(X,Y,G),
  \+ pertence(X,P1),
  path1(A,[X,Y|P1],G,P).
```



Caminhos em Grafos

- A definição da relação adjacente/3 depende da representação de grafos
- Se G é um grafo representado como grafo(Nós,Arestas)
  - G é não orientado

```
adjacente(X,Y,grafo(_,Arestas)) :-
  pertence(a(X,Y),Arestas) ;
  pertence(a(Y,X),Arestas).
```

  - G é orientado

```
adjacente(X,Y,grafo(_,Arestas)) :-
  pertence(a(X,Y),Arestas).
```

- Se G é um grafo representado por um conjunto de fatos aresta(X,Y)
  Nota: parâmetro G pode ser removido de **path/4** e **path1/4**
  - G é não orientado

```
adjacente(X,Y) :-
  aresta(X,Y) ;
  aresta(Y,X).
```

  - G é orientado

```
adjacente(X,Y) :-
  aresta(X,Y).
```

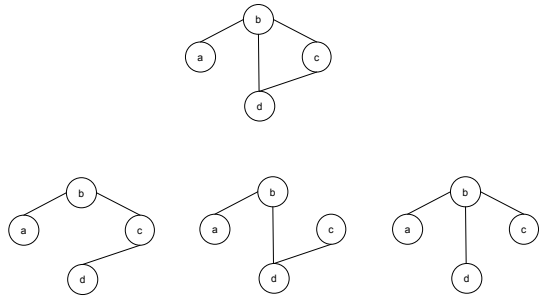

Caminhos em Grafos com Custos

```
% path(A,Z,G,P,C) P é um caminho acíclico de A até
% Z no grafo G com custo C
path(A,Z,G,P,C) :-
  path1(A,[Z],0,G,P,C).
path1(A,[A|P1],C1,_,[A|P1],C1).
path1(A,[Y|P1],C1,G,P,C) :-
  adjacente(X,Y,CustoXY,G),
  \+ pertence(X,P1),
  C2 is C1 + CustoXY,
  path1(A,[X,Y|P1],C2,G,P,C).
```


Árvore de Espalhamento de um Grafo

- Um grafo é dito **conexo** se há um caminho entre cada par de nós no grafo
- Uma **árvore de espalhamento** (*spanning tree*) de um grafo G(V,E) conexo é um grafo T(V,E'), onde E' é um subconjunto de E tal que
  - T é conexo
  - Não há ciclos em T
- Essas duas condições garantem que T é uma árvore

Árvore de Espalhamento de um Grafo



Árvore de Espalhamento de um Grafo

- Vamos definir a relação stree(G,T) onde T é a árvore de espalhamento de G, assumindo que G é conexo
  - Inicie com o conjunto vazio de arestas e gradualmente adicione novas arestas de G, com o cuidado que um ciclo nunca seja criado até que nenhuma aresta possa ser adicionada (pois isto criaria um ciclo)
  - O conjunto resultante define uma árvore de espalhamento
- A condição de não haver ciclo pode ser mantida considerando que uma aresta pode ser adicionada somente se um dos seus nós já está na árvore de espalhamento mas o outro nó não

Árvore de Espalhamento de um Grafo

```
% stree(G,Tree) gera árvore de espalhamento Tree do grafo G
stree(grafo(V,E),Tree) :-
  pertence(Aresta,E),
  spread(grafo(V,[Aresta]),Tree,grafo(V,E)).

spread(Tree1,Tree,G) :-          % cresce de Tree1 para Tree
  addedge(Tree1,Tree2,G), !,
  spread(Tree2,Tree,G).
spread(Tree,Tree,_).             % nenhuma aresta pode ser adicionada
                                 % sem formar ciclo

addedge(grafo(V,Tree),grafo(V,[a(A,B)|Tree]),G) :-
  adjacente(A,B,G),              % A e B adjacentes em G
  vertice(A,grafo(V,Tree)),      % A está na arvore
  \+ vertice(B,grafo(V,Tree)).   % aresta A-B não forma ciclo

adjacente(X,Y,grafo(_,E)) :-
  pertence(a(X,Y),E) ;
  pertence(a(Y,X),E).

vertice(V,G) :-                  % V é nó do grafo G se
  adjacente(V,_,G).              % V é adjacente a algum nó em G
```


Slides baseados nos livros:

Bratko, I.;
Prolog Programming for Artificial Intelligence,
3rd Edition, Pearson Education, 2001.

Clocksin, W.F.; Mellish, C.S.;
Programming in Prolog,
5th Edition, Springer-Verlag, 2003.

Material elaborado por
José Augusto Baranauskas
Revisão 2007