MAC5710 - Estruturas de Dados

Notas de Aula - Listas Lineares: pilhas, filas, listas circulares, filas de prioridade

Siang - 2005

1 Introdução

[Material parcialmente baseado no livro de Knuth - *The Art of Computer Programming - Volume I*]

Uma lista linear é um conjunto de n elementos (de informações)

$$x_1, x_2, ..., x_n,$$

cuja propriedade estrutural envolve as posições relativas de seus elementos. Supondo $n > 0$, temos

1. x_1 é o primeiro elemento
2. para $1 < k < n, x_k$ é precedido por x_{k-1} e seguido por x_{k+1}
3. x_n é o último elemento.

Algumas operações que podemos querer realizar sobre listas lineares:

1. Ter acesso a x_k, k qualquer, a fim de examinar ou alterar o conteúdo de seus campos
2. Inserir um elemento novo antes ou depois de x_k
3. Remover x_k
4. Combinar 2 ou mais listas lineares em uma só
5. Quebrar uma lista linear em duas ou mais
6. Copiar uma lista linear em um outro espaço
7. Determinar o no. de elementos de uma lista linear

Operações (1), (2) e (3) para $k = 1$ e $k = n$ são muito importantes. Nomes especiais como pilha ou fila são dados para as listas linears conforme a maneira essas operações são realizadas, como veremos em breve.

Num programa de aplicação, raramente são necessárias todas as operações acima simultâneamente. A maneira de representar listas lineares depende da classe de operações mais frequentes. Não existe, em geral, uma única representação para a qual todas as operações são eficientes. Por exemplo, não existe uma representação para atender às seguintes duas operações de maneira eficiente:

1. ter acesso fácil ao x_k, para k qualquer

2. inserir ou remover elementos em qualquer posição da lista linear

2 Pilhas e Filas

Listas linears em que inserções, remoções e acessos a elementos ocorrem no primeiro ou no último elemento são muito frequentemente encontradas. Tais listas linears recebem nomes especiais.

2.1 Pilha ou "stack"

É uma lista linear em que todas as inserções e remoções são feitas numa mesma extremidade da lista linear. Esta extremidade se denomina *topo* (em inglês "top") ou lado aberto da pilha.

As operações definidas para uma pilha incluem:

1. Verificar se a pilha está vazia

2. Inserir um elemento na pilha (empilhar ou "push"), no lado do topo.

3. Remover um elemento da pilha (desempilhar ou "pop"), do lado do topo.

Como o último elemento que entrou na pilha será o primeiro a sair da pilha, a pilha é conhecida como uma estrutura do tipo LIFO ("Last In First Out").

Exemplos:

1. Na vida real: pilhas de pratos numa cafeteria (acréscimos e retiradas de pratos sempre feitos num mesmo lado da pilha - lado de cima)

2. Na execução de uma programa: uma pilha pode ser usada na chamada de procedimentos, para armazenar o endereço de retorno (e os parâmetros reais). A medida que procedimentos chama outros procedimentos, mais e mais endereços de retorno devem ser empilhados. Estes são desempilhados à medida que os procedimentos chegam ao seu fim.

3. Na avaliação de expressões aritméticas, a pilha pode ser usada para transformar expressões em notação polonesa ou pos-fixa. A pilha também pode ser usada na avaliação de expressões aritméticas em notação polonesa.

2.2 Fila ou "queue"

É uma lista linear em que todas as inserções de novos elements são realizadas numa extremidade da lista e todas as remoções de elementos são feitas na outra extremidade da lista.

As filas são estruturas do tipo FIFO ("First In First Out"). Usando a notação do livro de Knuth, elementos novos são inseridos no lado R ("Rear" ou fim da fila) e a retirada ocorre no lado F ("Front" ou frente ou começo da fila).

Exemplo: Num sistema operacional, os processos prontos para entrar em execução (aguardando apenas a disponibilidade da CPU) são geralmente mantidos numa fila.

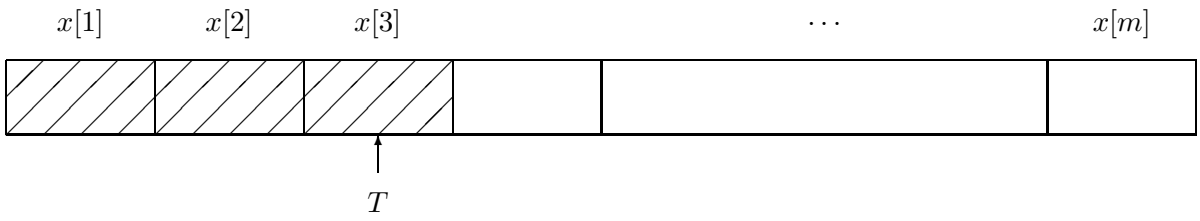
Obs. Existe um tipo de fila em que as retiradas de elementos da fila depende de um valor chamado prioridade de cada elemento. O elemento de maior prioridade entre todos os elementos da fila é o próximo a ser retirado. Tal fila recebe o nome de fila de prioridade.

3 Representação de listas linears

3.1 Alocação sequencial

Os elementos da lista linear ocupam posições consecutivas da memória do computador.

(a) Pilha (alocação sequencial)



T = topo da pilha
Convenção adotada para pilha vazia: $T = 0$

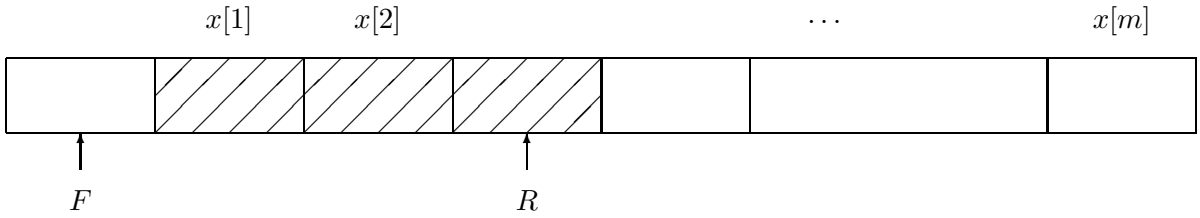
Inserir um novo elemento de valor Y na pilha:

```
T := T + 1;
if T > m then overflow;
x[T] := Y
```

Remover um elemento da pilha, colocando o valor do elemento retirado em Y:

```
if T = 0 then underflow
else
    begin
       Y := x[T];
       T := T - 1
    end
```

(b) Fila (alocação sequencial)



Convenção para fila vazia: $R = F$

Situação inicial $R = F = 0$

Inserir um elemento Y na fila

```
R := R + 1;
if R > m then overflow;
x[R] := Y
```

Remover da fila:

```
if R = F then underflow;
F := F + 1;
Y := x[F];
if R = F then
begin
    F := 0;
    R := 0
    end
```

Obviamente esta solução pode ser melhorada, representando-se a fila como *fila circular*, como se segue.

(c) Fila circular (alocação sequencial)

Na fila circular, tudo se passa como se a posição $x[m]$ seja seguida por $x[1]$, ou $x[1]$ seja precedida por $x[m]$.

Convenção de fila vazia: $R = F$

Situação inicial: $R = F = m$

Inserir Y na fila circular:

```
if R = m then R := 1
    else  R := R + 1;
if R = F then overflow;
x[R] := Y
```

Remover da fila circular:

```
if R = F then underflow;
if F = m then F := 1
 else F := F + 1;
Y := x[F]
```

Crítica sobre o uso da alocaçã sequencial

Por um lado, a alocação sequencial simplifica bastante a implementação dos algoritmos de inserção e remoção.

Por outro lado, pode-se constatar a grande dificuldade quando várias estruturas de dados (digamos 3 pilhas e duas filas) devem ser implementadas num espaço sequencial

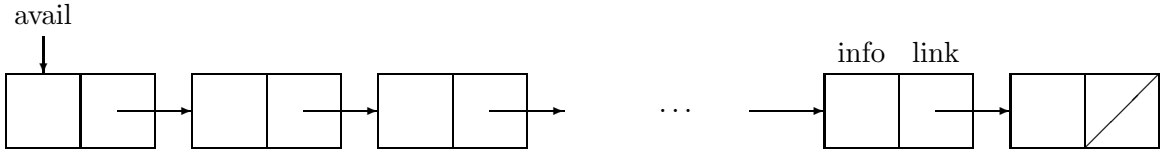
único. Temos que dividir a priori o espaço em três partes para implementar cada estrutura de dado em cada parte do espaço. Esse dimensionamento pode não ser fácil pois podemos desconhecer qual das estruturas irá crescer mais que outras. Assim, podemos chegar a situação em que se esgota o espaço alocado para uma determinada estrutura de dado enquanto há na verdade espaço sobrando naquelas partes reservadas para as outras estruturas de dados. O rearranjo de espaço é possível mas em geral envolve uma grande movimentação de dados e portanto é muito custoso.

Para compartilhar uma mesmo espaço por várias estruturas de dados, uma forma muito elegante é usar uma outra alocação, a chamada alocação ligada ou encadeada.

3.2 Alocação ligada

3.2.1 Lista livre

Para usar a alocação ligada, todo o espaço livre é organizado inicialmente numa lista livre, onde os elementos apresentam dois campos: um chamado info (para armazenar as informações das listas linears) e um campo chamado link (para apontar para o próximo elemento). Se x aponta para um elemento, então os seus campos info e link serão indicados por info(x) e link(x). Por simplicidade, nos exemplos, o campo info será constitu¡do apenas por um valor do tipo inteiro. Evidentemente, ele pode conter informações mais complexas, dependendo do problema. A lista livre é apontada por uma variável apontadora chamada avail. A idéia é que quando uma lista linear precisa crescer de tamanho (inserção), um elemento livre é extra¡do da lista livre e usado pela lista linear. Quando a lista linear não precisa mais de um elemento (remoção), o elemento removido é devolvido à lista livre. A lista livre pode assim ser compartilhada por várias listas lineares. Esse esquema de alocação de memória é também conhecido pelo nome de *alocação dinâmica* de memória.



O in¡cio da lista livre é apontado pela variável apontadora avail.

Extração de um elemento livre da lista livre, que será apontado por P

ExtraiLivre(P):

```
if avail = nil then CollectGarbage;
P := avail;
avail := link(avail);
link(P) := nil
```

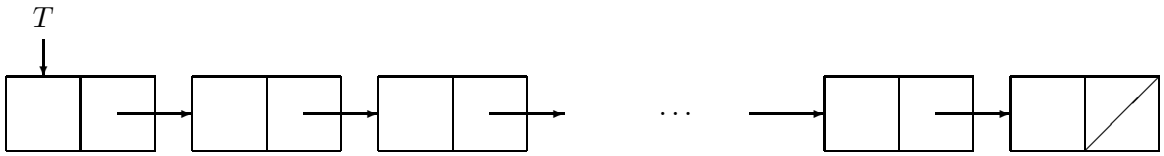
Devolução de um elemento (apontado por P) à lista livre

DevolveLivre(P):

```
link(P) := avail;
avail := P;
P := nil
```

Na execução de ExtraiLivre, pode ser que a lista livre já esteja vazia (condição avail = nil). Neste caso, é chamada uma rotina CollectGarbage, que tenta identificar e recuperar elementos não mais ativos, devolvendo-os à lista livre.

3.2.2 Pilha



Considere uma pilha cujo topo é apontado pela variável T.
Pilha vazia: $T = \text{nil}$

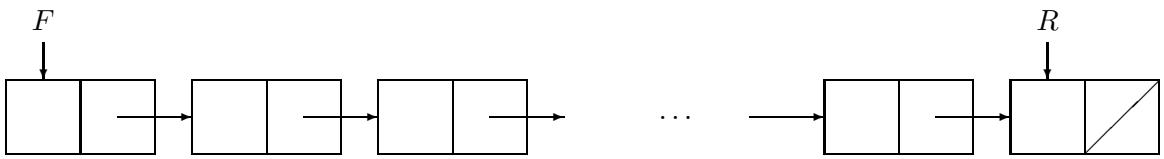
push(T, Y): Inserir um novo elemento, a conter a informação Y, na pilha apontada por T

```
ExtraiLivre(P);
info(P) := Y;
link(P) := T;
T := P
```

pop(T, Y): Remover um elemento da pilha apontada por T, colocando a informação do elemento removido em Y

```
if T = nil then
    underflow else
begin
    P := T;
    T := link(P);
    Y := info(P);
    DevolveLivre(P)
end
```

3.2.3 Fila



Variáveis apontadoras da fila: F e R
Fila vazia: $F = R = \text{nil}$

InsereFile(F, R, Y): Insere um elemento novo com informação Y na fila

```
ExtraiLivre(P);
info(P) := Y;
link(P) := nil;
if R not = nil then
begin
    link(R) := P;
    R := P
end
else
begin
    R := P;
    F := P
end
```

RemoveFile(F, R, Y): Remove um elemento da fila, colocando a informação do elemento removido em Y

```
if F = nil the underflow
else
begin
    P := F;
    F := link(P);
    Y := info(P);
    DevolveLivre(P);
    if F = nil then R := nil
end
```

4 Implementação da alocação dinâmica de memória

Em linguagens como Fortran e Assembler, a organização da lista livre, assim como as operações para a sua manipulação, têm que ser explicitamente construı¡das pelo programador.

Em certas linguagens, como Pascal, Linguagem C e Modula 2, já existem funções pré-definidas da linguagem para efetuar operações do tipo ExtraiLivre e DevolveLivre.

4.1 Construção da lista livre e funções para sua manipulação

Mostramos abaixo como tais funções podem ser construı¡das em Pascal. Embora em Pascal essas funções já fazem parte da linguagem, o exemplo pode ser útil para quem quiser implementar as mesmas funções por exemplo em Assembler.

Usamos um array info e um array link, alocando um total de m elementos para cada array. nil será representado por 0. Variáveis apontadoras contêm realmente ¡ndices desses arrays.

```
const m = 1024; {qualquer outro valor serve}
var info,link: array[1..m] of integer;
avail: integer;
procedure initialize;
```

```
{criação da lista livre inicial}
var i: integer;
begin
   for i:=1 to m-1 do
   begin
      info[i] := 0;
      link[i] := i+1
   end;
   info[m] := 0;
   link[m] := 0;
   avail := 1
end;

procedure extrai(var P: integer);
{Extrai um elemento, a ser apontado por P, da lista livre}
begin
   if avail = 0 then CollectGarbage;
   P := avail;
   avail := link[avail]
end;

procedure devolve(P: integer);
{Devolve um elemento apontado por P à lista livre}
begin
   link[P] := avail;
   avail := P
end;
```

No programa principal, criamos a lista livre chamando inicialize. Podemos escrever outros procedimentos que chamam extrai e devolve para retirar e devolver elementos da lista livre. Alguns exemplos se seguem.

```
procedure push(var T: integer; Y: integer);
{Insere Y numa pilha apontada por T}
var P: integer;
begin
   extrai(P);
   info[P] := Y;
   link[P] := T;
   T := P
end;

procedure pop(var T,Y: integer);
{Retira um elemento da pilha apontada por T}
var P: integer;
begin
   if T = 0 then
      underflow
```

```
    else
    begin
       Y := info[T];
       P := T;
       T := link[T];
       devolve(P)
    end
end
```

4.2 Alocação dinâmica usando funções embutidas

Em Pascal, existe um tipo chamado *pointer* ou apontador que é usado para apontar para outras estruturas. Por exemplo, podemos definir

```
type elemento =
   record
      info: integer;
      link: ^elemento
   end
```

Aqui o record elemento possui um campo link cujo tipo é um apontador (indicado por $\wedge$) a um elemento.

```
var x,y : ^elemento;
```

Declaramos duas variáveis x e y do tipo apontador a elemento. Para podermos usar x, devemos ter algum elemento para ser apontado por x.

```
new(x);
```

A função *new*, com o parâmetro x que é uma variável do tipo apontador para elemento, tem o papel de criar um registro do tipo elemento, alocado de maneira dinâmica. A função new obtém espaço de um espaço semelhante à nossa lista livre.

```
x^.info := 40;
x^.link := nil;
```

O registro apontado por x é indicado por $x\wedge$. Assim $x\wedge$.info denota o campo info do registro apontado por x. nil é usado para indicar o apontador nulo.

```
y := x;
```

O comando de atribuição vale também para variáveis do tipo apontador, se ambas são do mesmo tipo.

```
dispose(x);
```

A função *dispose* é usada para devolver memória não mais útil para poder ser usada novamente.

```
procedure push(var T: ^elemento; Y: integer);
var P: ^elemento;
begin
   new(P);
   P^.info := Y;
   P^.link := T;
   T := P
end;

procedure pop(var T: ^elemento; var Y: integer);
var P: ^elemento;
begin
   if T = nil then
      underflow
   else
   begin
      Y := T^.info;
      P := T;
      T := T^.link;
      dispose(P)
   end
end
```

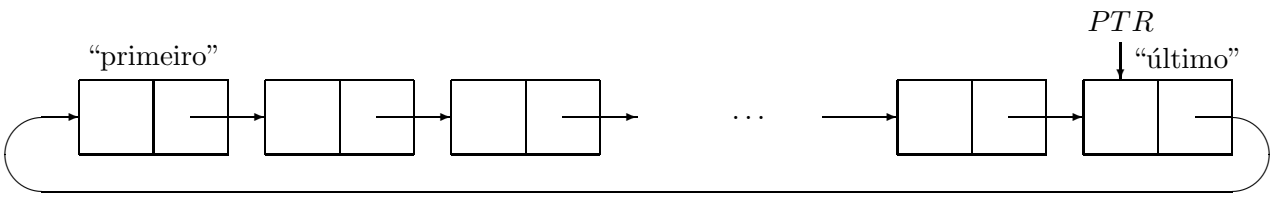
Tendo declarado

```
var topo: ^elemento;
    valor: integer;
```

podemos escrever no programa principal

```
topo := nil; {pilha começa vazia}
push(topo,102);
push(topo,304);
pop(topo,valor);
```

5 Listas circulares ligadas

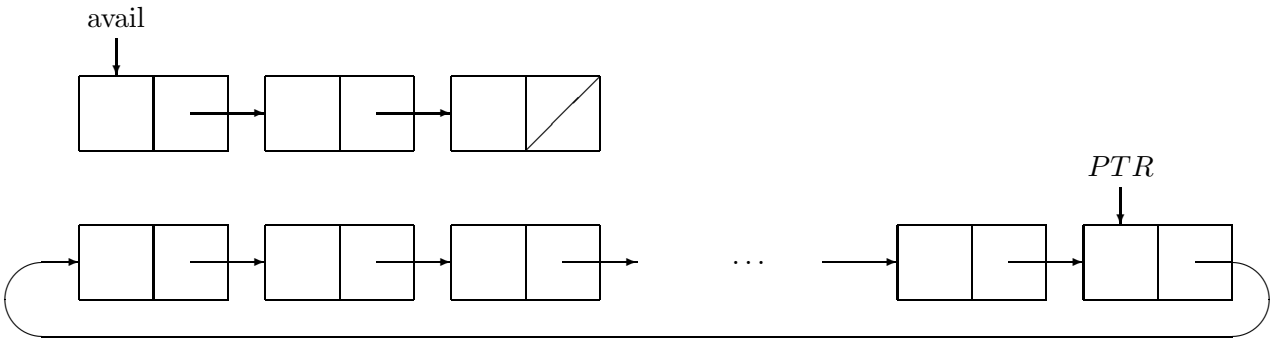


O último nó aponta de volta para o primeiro nó. O apontador PTR aponta para o último nó; assim sendo, link(PTR) dará acesso ao primeiro nó.

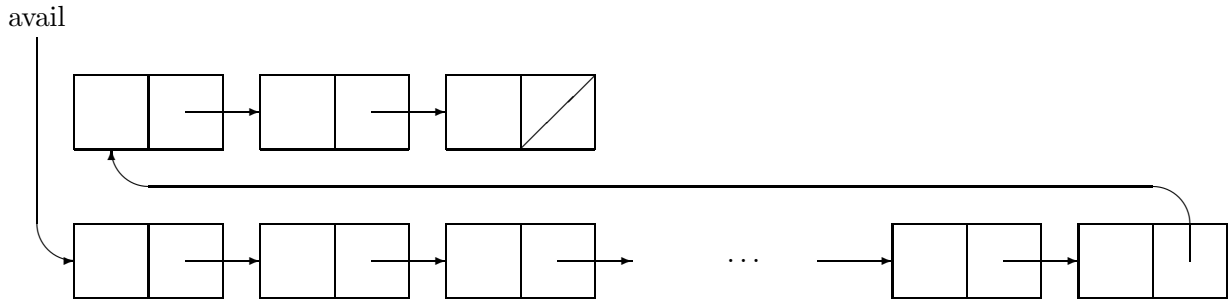
Lista vazia: PTR = nil

5.1 Devolução de uma lista circular à lista livre avail

Antes da devolução:



Depois da devolução:




```
if PTR not = nil then
    avail <-> link(PTR)
```

onde $\leftrightarrow$ significa trocar entre si, isto é,

```
    P := avail;
    avail := link(PTR);
    link(PTR) := P
```

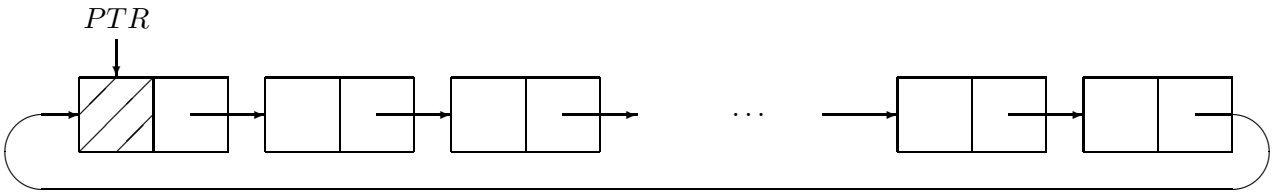
Se estamos percorrendo uma lista circular para procurar um nó com determinado valor contido, é bom sabermos quando parar a busca, caso tal valor não esteja presente. Por exemplo, para determinar se algum nó contém Y no seu campo info, podemos escrever:

```
P := PTR;
achou := false;
repeat
    if info(P) = Y then
```

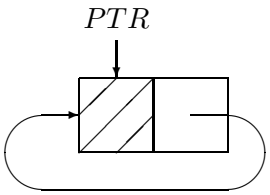
```
        achou := true
    else
        P := link(P)
until
    achou or P = PTR
```

Sabemos que "demos uma volta" pelo valor do apontador. Uma outra maneira é usar um nó especial chamado "cabeça" da lista, cuja informação é algum valor especial usado só para tal finalidade.

5.2 Lista circular com cabeça de lista



Lista vazia: só tem a cabeça



6 Listas duplamente ligadas

Um nó possui além do campo info, campos para dois apontadores llink e rlink.
Lista vazia: Tanto llink como rlink apontam para a cabeça.

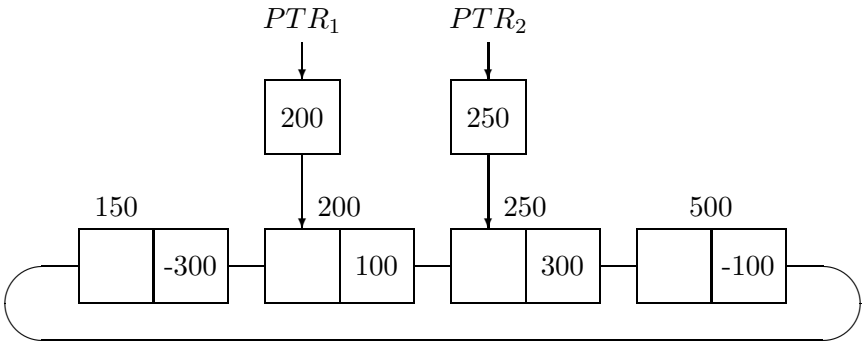
Uma vantagem óbvia é a possibilidade de percorrer a lista nos dois sentidos (tanto para a direita como para a esquerda). Uma outra caracter¡stica é a possibilidade de poder remover um elemento qualquer da lista, conhecendo-se apenas um apontador ao mesmo. Por exemplo, para remover o elemento apontado por x, basta fazer:

```
rlink(llink(x)) := rlink(x);
llink(rlink(x)) := llink(x);
devolve(x)
```

Analogamente, podemos inserir um elemento novo, apontado por P, à direita ou esquerda de um elemento apontado por x.

O preço das caracter¡sticas ou vantagens acima é o uso de dois campos em cada nó para guarda rlink e llink. Usando apenas um campo, será poss¡vel percorrer a lista facilmente nos dois sentidos?

A resposta é SIM, como mostra o seguinte exerc¡cio extra¡do do livro de Knuth. Basta colocar no campo link de cada elemento a diferença do endereço do próximo elemento com o do elemento anterior. Dois apontadores PTR1 e PTR2 são usados para apontar a dois elementos vizinhos da lista.




```
Andar um passo para esquerda:     Andar um passo para direita:
P := PTR2 - link(PTR1);          P := PTR1 + link(PTR2);
PTR2 := PTR1;                    PTR1 := PTR2;
PTR1 := P                        PTR2 := P
```

7 Ordenação topológica

Uma ordenação parcial de um conjunto S é uma relação entre os objetos de S, indicada pelo s¡mbolo $\preceq$ (leia-se "precede ou igual"), satisfazendo as seguintes propriedades para quaisquer objetos x, y, z de S (não necessariamente distintos):

(i) Transitividade: se $x \preceq y$ e $y \preceq z$ então $x \preceq z$.

(ii) Anti-simétrica: se $x \preceq y$ e $y \preceq x$ então $x = y$.

(iii) Reflexividade: $x \preceq x$.

Se $x \preceq y$ e x not $\preceq y$, então escreveremos $x \prec y$ e diremos que "x precede y".

De (i), (ii) e (iii) temos:

(i') se $x \prec y$ e $y \prec z$ então $x \prec z$.

(ii') se $x \prec y$ então y not $\prec x$.

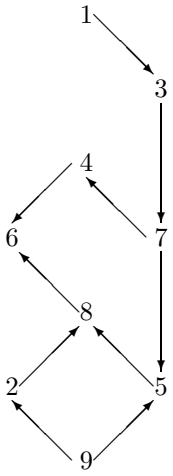
(iii') x not $\prec x$.

Exemplos de ordenações parciais:

1. Relação $\leq$ (menor ou igual) entre números.
2. Relação $\subseteq$ (contido em) entre conjuntos.
3. Relação "x deve ser executado antes de y" em um conjunto de atividades.

7.1 Uma representação em diagrama

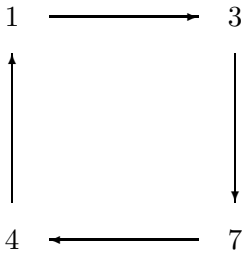
Vamos representar $x \prec y$ por $x \to y$. Assim o diagrama abaixo



representa as relações

$9 \prec 2 \quad 4 \prec 6$
$3 \prec 7 \quad 1 \prec 3$
$7 \prec 5 \quad 7 \prec 4$
$5 \prec 8 \quad 9 \prec 5$
$8 \prec 6 \quad 2 \prec 8$

A propriedade (ii) significa que não existem ciclos fechados. Assim, o seguinte não é ordenação parcial.



7.2 O problema da ordenação topológica

Dada uma ordenação parcial, uma ordenação topológica é uma sequência

$$a_1 a_2 \ldots a_n$$

tal que para $a_j \prec a_k$, temos $j < k$.

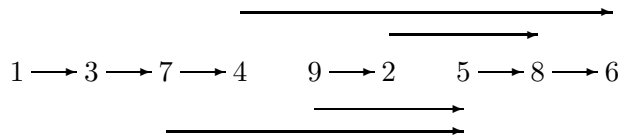
Isto é, se um elemento a_j precede a_k, ele irá aparecer antes de a_k na ordenação topológica.

Um exemplo de uma ordenação topológica do exemplo acima é

Usando o diagrama,

Todas as "flechas" apontam para a direita.

1 3 7 4 9 2 5 8 6



7.3 Um algoritmo de ordenação topológica

Sejam n objetos numerados de 1 a n. Seja dado o valor n como a primeira entrada. Os dados seguintes de entrada são pares da forma

$$j \quad k$$

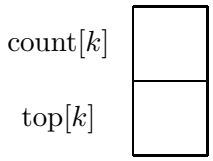
onde cada par significando $j \prec k$.

O último par contém

$$0 \quad 0$$

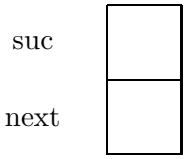
indicando o fim dos dados.

Usamos uma tabela sequencial $x[1], x[2], \ldots, x[n]$ onde cada $x[k]$ tem a forma



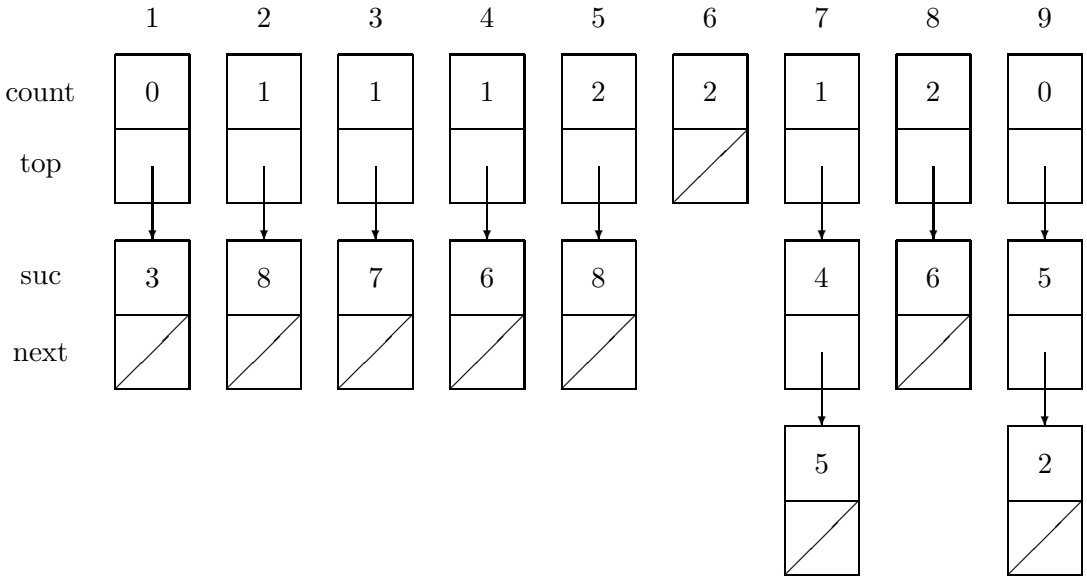
count[k] contém o número de predecessores diretos do objeto k, isto é, o no. de pares $(j \prec k)$ que aparecem na entrada.

top[k] contém um apontador a uma lista de sucessores diretos do objeto k. Cada elemento dessa lista tem a forma



onde suc indicando o sucessor direto e next apontando para um outro sucessor.

Para o exemplo, temos



O algoritmo de ordenação topológica fica então

```
I. [inicialização]

   read(n); AindaSobrou := n;
   for i := 1 step 1 until n do
       begin
           count[i] := 0;
           top[i] := nil
       end;
```

II. [Leitura dos pares $j \prec k$ e construção das listas]

```
   read (j,k);
   while j not = 0 do
       begin
           count[k] := count[k] + 1;
           extrai(P);
           suc(P) := k;
           next(P) := top[j];
           top[j] := P;
           read (j,k)
        end;
```

III. [Liga todos os nós com count $= 0$ numa fila para facilitar a procura pelo próximo elemento com count nulo. Usamos o mesmo campo tanto para conter count como para qlink, que guarda apontadores para os elementos dessa fila.]

```
   R := 0;
   qlink[0] := 0;
   for i := 1 step 1 until n do
       if count[i] = 0 then
           begin
               qlink[R] := i;
               R := i
           end;
   F := qlink[0];
```

IV. [Produção de uma ordenação topológica.]

```
   while F not = 0 do
       begin
           write(F);
           P := top[F];
           AindaSobrou := AindaSobrou - 1;
           while P not = nil do
               begin
                   count[suc(P)] := count[suc(P)] - 1;
```

```
                if count[suc(P)] = 0 then
                    begin
                        qlink[R] := suc(P);
                        R := suc(P)
                    end;
                P := next(P)
            end;
        F := qlink[F]
    end;
```

V. [Verificação final.]

```
if AindaSobrou > 0 then
    error;
    {existe um ciclo fechado com AindaSobrou elementos, não dá
     para produzir uma ordenação topológica.}
```

8 Fila de prioridade

[Material baseado no livro de A. N. Habermann - Principles of Operating Systems]

Fila de prioridade é uma estrutura de dado que mantém uma coleção de elementos, cada um com uma prioridade associada. Valem as operações seguintes.

- Inserir um elemento novo na fila de prioridade.
- Remover o elemento de maior prioridade da fila de prioridade.

8.1 Implementação com ordenação total

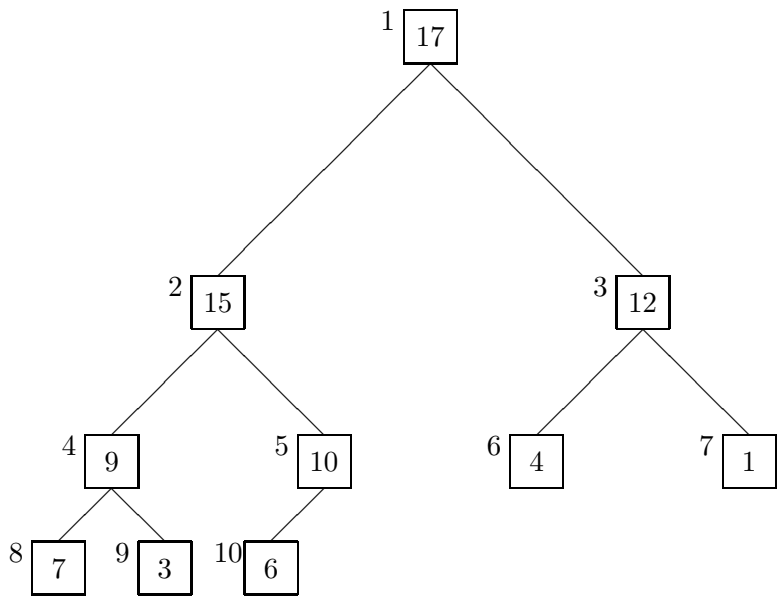
Uma maneira de representar uma fila de prioridade é manter uma lista linear ligada ou encadeada em que os elementos estão sempre ordenados por prioridades decrescentes. Assim,

- Para remover um elemento da fila de prioridade: tempo constante.
- Para inserir um novo elemento: tempo $O(n)$, onde n é o no. de elementos na fila.

8.2 Implementação com ordenação parcial, usando um "heap"

"Heap" é uma estrutura de árvore binária em que cada nó terminal ou não-folha tem uma prioridade maior ou igual à prioridade de seus filhos. Em particular, vamos exigir que apenas o último n¡vel da árvore pode ser incompleto e, nesse n¡vel, se incompleto, os nós devem estar todos "encostados à esquerda".

Vamos numerar os nós do "heap" em ordem de n¡veis crescentes, indo da esquerda para a direita em cada n¡vel (ordem chamada "breadth-first"). A seguinte propriedade útil se verifica.



- O pai do nó k é o nó k div 2 (isto é, quociente inteiro de k / 2).
- Se nó k tem um filho esquerdo, este será o nó 2k; se k também tem um filho da direita, este será o nó $2k + 1$.

Podemos portanto usar uma alocação sequencial (por exemplo usando um array) para representar um "heap". A estrutura de árvore está impl¡cita nas posições dos nós. Por exemplo, usando um array H:

H	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
	17	15	12	9	10	4	1	7	3	6				

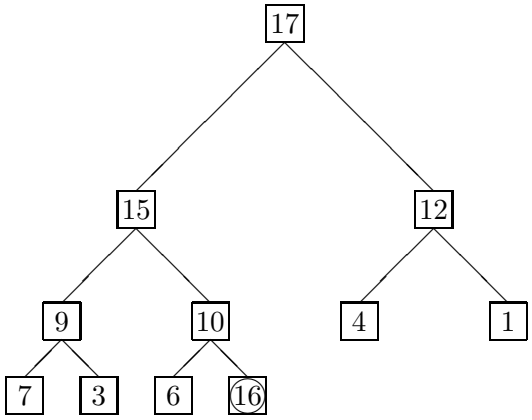
8.2.1 Inserir um elemento novo

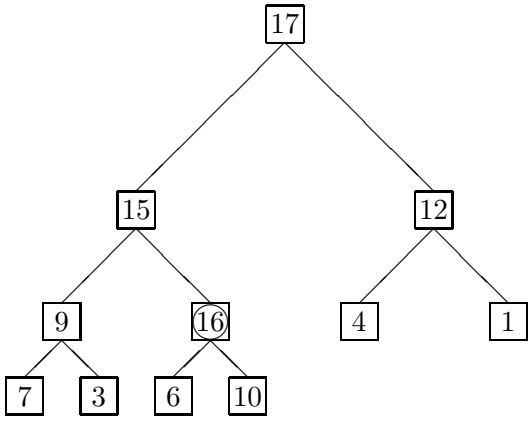
Para inserir um novo elemento com prioridade x, cria-se um novo elemento no fim do array H para receber x. Isso pode perturbar a propriedade do "heap". Para consertar isso, fazemos o seguinte. Se x for maior que seu pai, então os dois trocam de lugar. Essa operação é repetida até que x encontre o seu lugar correto na árvore. Por exemplo, para inserir o elemento 16, fazemos as seguintes trocas.

Seja ult o ¡ndice indicando o último elemento do "heap" (array H) antes da inserção. No exemplo, ult valia 10. Após a inserção, ele passará a valer 11.

```
insere(x)

ult := ult + 1;
k := ult;
```





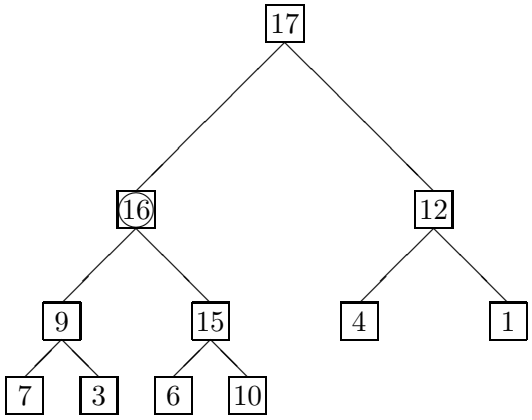

```
while (k div 2) and x > H[k div 2] do
    begin
        H[k] := H[k div 2];
        k := k div 2
    end;
H[k] := x
```

Note que nesse algoritmo, o novo elemento não é colocado dentro do "heap" até que o lugar apropriado tenho sido obtido. O algoritmo é O(log n), onde log denota logaritmo na base 2.

8.2.2 Remover um elemento da fila

A remoção em si é muito simples, já que o elemento de maior prioridade é $H[1]$. Após a remoção, entretanto, precisamos re-arranjar os elementos do "heap". Colocamos em $H[1]$ o elemento $H[\text{ult}]$, liberando assim a última posição. Se o elemento colocado em $H[1]$ for menor que seus filhos, então ele trocado com o maior dos filhos. Isso é repetido até tal elemento ocupar a posição correta. O algoritmo também é O(log n).

```
remove(Y)
```




```
    Y := H[1];
    x := H[ult];
%    parece que nao precisa disso:
%    H[ult] := - infinity;
    ult := ult - 1;
    k := 1;
    while 2k <= ult and (x < H[2k] or x < H[2k + 1]) do
        if H[2k] > H[2k + 1] then
            begin
                H[k] := H[2k];
                k := 2k
            end
        else
            begin
                H[k] := H[2k + 1];
                k := 2k + 1
            end;
    H[k] := x
```