Listas Lineares

Siang Wun Song - Universidade de São Paulo - IME/USP

MAC 5710 - Estruturas de Dados - 2008

Referência bibliográfica

Os slides sobre este assunto de listas lineares são parcialmente baseados no segundo capítulo do livro

- D. E. Knuth. The Art of Computer Programming. Volume 1, Addison Wesley, 1973.

Introdução

- Uma estrutura de dado armazena dados na memória do computador a fim de permitir o acesso eficiente dos mesmos.
- A maioria das estruturas de dados consideram a memória primária (a chamada RAM) como pilhas, filas, árvores binárias de busca, árvores AVL e árvores rubro-negras. Outras são especialmente projetadas e adequadas para serem armazenadas em memórias secundárias como o disco rígido, e.g. B-árvores.
- Uma estrutura de dado bem projetada permite a manipulação eficiente, em tempo e em espaço, dos dados armazenados através de operações específicas. Um conceito relacionado com a estrutura de dado é o tipo abstrato de dados, que veremos em breve.

Lista linear

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

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

Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Ter acesso a x_k, k qualquer, a fim de examinar ou alterar o conteúdo de seus campos
- Inserir um elemento novo antes ou depois de x_k
- Remover x_k
- Colocar todos os elementos da lista em ordem.
- Combinar 2 ou mais listas lineares em uma só
- Quebrar uma lista linear em duas ou mais
- Copiar uma lista linear em um outro espaço

Trataremos neste curso as três primeiras operações, para $k = 1$ e $k = n$, casos considerados importantes e as listas lineares recebem nomes como pilha ou fila conforme a maneira essas operações são realizadas.

A operação de ordenação é extensivamente estudada em cursos de Análise de Algoritmos.

Implementação de uma estrutura de dado

A maneira de implementar listas lineares depende da classe de operações mais frequentes. Não existe, em geral, uma única implementação para a qual todas as operações são eficientes. Por exemplo, não existe uma implementação 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

A operação 1 fica eficiente se a lista é implementada em um vetor (*array*) em alocação seqüencial na memória. Já para a operação 2 é mais adequada a alocação encadeada ou ligada, com o uso de apontadores.

Veremos na seção Tipos Abstratos de Dados que devemos separar as preocupações com relação à especificação de uma estrutura de dado e sua implementação.

Pilha (*stack*)

Listas lineares 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 como pilha e fila.

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:

- Verificar se a pilha está vazia
- Inserir um elemento na pilha (empilhar ou “push”), no lado do topo.
- Remover um elemento da pilha (desempilhar ou “pop”), do lado do topo.

Pilha é uma estrutura LIFO)

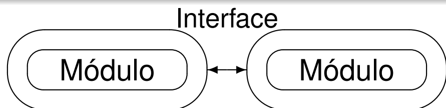
Como o último elemento que entrou na pilha será o primeiro a sair da pilha, a pilha é conhecida como uma estrutura do tipo LIFO (“Last In First Out”). Exemplos:

- Na vida real: pilhas de pratos numa cafeteria (acréscimos e retiradas de pratos sempre feitos num mesmo lado da pilha - lado de cima)
- 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 chamam outros procedimentos, mais e mais endereços de retorno devem ser empilhados. Estes são desempilhados à medida que os procedimentos chegam ao seu fim.
- Na avaliação de expressões aritméticas, a pilha pode ser usada para transformar expressões em notação polonesa ou pós-fixa. A pilha também pode ser usada na avaliação de expressões aritméticas em notação polonesa.

Fila (*queue*) é uma estrutura FIFO

- É uma lista linear em que todas as inserções de novos elements são realizadas numa extremidade da lista e todas as remoções são feitas na outra extremidade.
- Uma fila é uma estrutura do tipo FIFO ("First In First Out"). Elementos novos são inseridos no lado *In* (fim da fila) e a retirada ocorre no lado *Out* (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.
- 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.

Tipos abstratos de dados



- No trabalho pioneiro de D. Parnas – “On the criteria to be used in decomposing systems into modules”, *Communications of the ACM* 15, 2, 1972, pp. 1053–1058, Parnas apresenta princípios para nortear o projetista na estruturação do software.
- O principal argumento é o chamadao *ocultamento de informação* (“information hiding”).
- Um programa é decomposto em móulos. Um módulo deve expor a outros somente a informação necessária para a correta operação, escondendo internamente estruturas de dados, lógica nos procedimentos, etc. Essa é a base do conceito de encapsulamento de linguagens orientadas a objetos.

Especificação formal de tipos abstratos de dados

Um tipo abstrato é o conjunto de dados e de operações sobre esses dados.

Na abstração por meio de tipos abstratos de dados, usa-se especificação algébrica para permitir a definição axiomática de um tipo abstrato (composto por dados e operações) sem impor detalhes de implementação.

A visibilidade da estrutura interna do tipo de dado é limitada às operações que forem explicitamente enumerados como exportáveis, incorporando a idéia de "information hiding" de Parnas.

Há portanto a separação de

- definição ou especificação de um tipo e
- sua implementação

Especificação do tipo abstrato de dado chamado pilha (*stack*)

Como ilustração, damos um exemplo sem maiores detalhes.
Tipo stack
Operações:

1. Push
2. Pop
3. Top
4. Clear
5. Empty

Condições pre e pos: estando **pre** verificada, **pos** é verificada após a operação.

Especificação do tipo pilha

Convenção:
S Valor da pilha antes do procedimento
S' Valor da pilha após o procedimento
$\neg$ não
$\sim$ concatenação de duas cadeias
$<>$ cadeia vazia
length comprimento da cadeia
maxstack tamanho máximo da pilha

RESULT resultado da operação

procedure Push (var S: stack; x: T);
    pre $\neg$ Full (S');
    pos $S' =< x >\sim S$;

function Pop (var S: stack): T;
    pre $\neg$ Empty (S');
    pos $S' =<\text{RESULT}> \sim S$;

Especificação do tipo pilha

Convenção:
S Valor da pilha antes do procedimento
S' Valor da pilha após o procedimento
$\neg$ não
$\sim$ concatenação de duas cadeias
$<>$ cadeia vazia
length comprimento da cadeia
maxstack tamanho máximo da pilha

RESULT resultado da operação

function Top (S: stack): T;
  pre $\neg$ Empty (S);
  pos RESULT = first (S');

proceudre Clear (var S: stack);
  pos $S' =<>$;

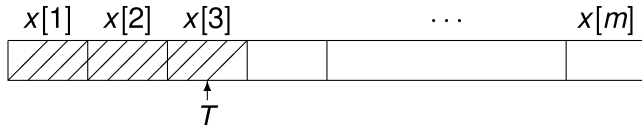
function Empty (S: stack):boolean;
  pos RESULT $\equiv$ ($S' =<>$);

Implementação do tipo abstrato de dado chamado pilha (*stack*)

Os módulos que usam a pilha podem se basear nas condições **pre** e **pos**, mas não podem usar detalhes e informações de implementação.

Implementação: Pilha com alocação seqüencial

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



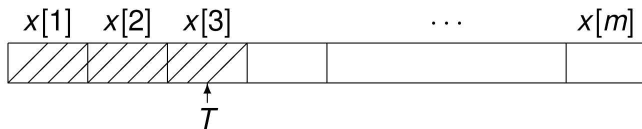
T = topo da pilha

Convenção adotada para pilha vazia: $T = 0$

Inserir um novo elemento de valor Y na pilha:

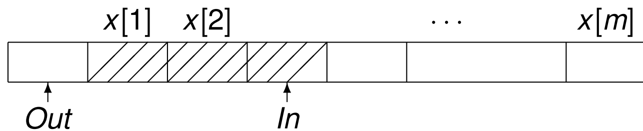
1: $T \leftarrow T + 1$
2: **if** $T > m$ **then**
3: overflow
4: **end if**
5: $x[T] \leftarrow Y$

Implementação: Pilha com alocação seqüencial



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

1: **if** $T = 0$ **then**
2: underflow
3: **else**
4: $Y \leftarrow x[T]$
5: $T \leftarrow T - 1$
6: **end if**

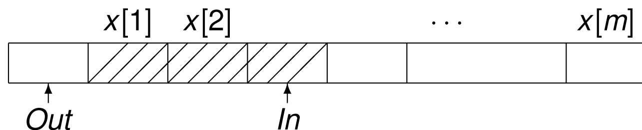


Convenção para fila vazia: $In = Out$

Situação inicial $In = Out = m$

Nessa implementação, tudo se passa como se a posição $x[m]$ seja seguida por $x[1]$, ou $x[1]$ seja precedida por $x[m]$.
Inserir Y na fila circular:

1: **if** $In = m$ **then**
2: $\quad In \leftarrow 1$
3: **else**
4: $\quad In \leftarrow In + 1$
5: **end if**
6: **if** $In = Out$ **then**
7: $\quad$ overflow
8: **end if**
9: $x[In] \leftarrow Y$



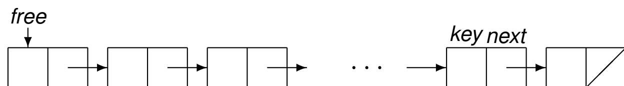
Remover da fila circular:

1: **if** $In = Out$ **then**
2: underflow
3: **else**
4: **if** $Out = m$ **then**
5: $Out \leftarrow 1$
6: **else**
7: $Out \leftarrow Out + 1$
8: **end if**
9: **end if**
10: $Y \leftarrow x[Out]$

Crítica sobre o uso da alocação seqüencial

- A alocação seqüencial simplifica bastante a implementação dos algoritmos de inserção e remoção.
- Entretanto, é difícel implementar várias estruturas de dados (digamos 3 pilhas e duas filas) em um mesmo espaço seqüencial.
  - Temos que particionar a priori o espaço em partes para implementar cada estrutura de dado.
  - Isso 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 espaço sobrando nas partes reservadas para as outras estruturas. Rearranjar o espaço é possível mas pode envolver uma custosa movimentação de dados.
- Para compartilhar uma mesmo espaço por várias estruturas de dados, apresentarmos a alocação ligada ou encadeada.

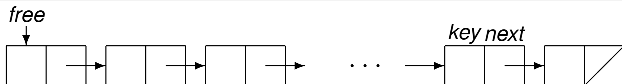
Alocação ligada ou encadeada: Lista livre



Na alocação ligada, todo o espaço livre é organizado inicialmente numa lista livre, apontada pela variável apontadora *free*, onde os elementos apresentam dois campos: um chamado *key* (para armazenar as informações) e um campo chamado *next* (para apontar para o próximo elemento).

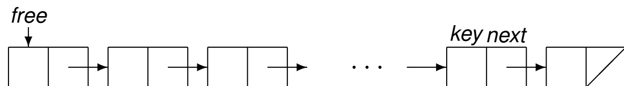
Se x aponta para um elemento, então os seus campos key e next serão indicados por $key(x)$ e $next(x)$.

Alocação ligada ou encadeada: Lista livre



- Nos exemplos, o campo *key* será constituído apenas por um valor digamos do tipo inteiro. Evidentemente, ele pode conter informações mais complexas, dependendo do problema.
- Quando uma estrutura de dado (e.g. uma pilha) precisa crescer de tamanho (inserção), um elemento livre é extraído da lista livre e usado pela estrutura de dado.
- Quando uma estrutura de dado remove um elemento, o elemento removido é devolvido à lista livre.
- A lista livre pode assim ser compartilhada por várias estruturas de dados (por exemplo, várias pilhas e filas). Esse esquema de alocação de memória é também conhecido pelo nome de *alocação dinâmica* de memória.

Extração de um elemento da lista livre

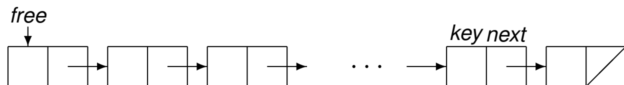


O início da lista livre é apontado pela variável apontadora free.
Extração de um elemento livre da lista livre, que será apontado por P

ExtraiLivre(P):

1: **if** $free = nil$ **then**
2: CollectGarbage
3: **end if**
4: $P \leftarrow free$
5: $free \leftarrow next(free)$
6: $next(P) \leftarrow nil$

Devolução de um elemento à lista livre

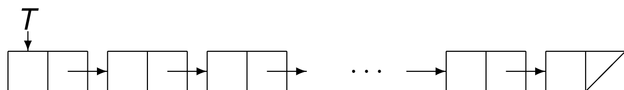


Devolver o elemento apontado por P à lista livre.
DevolveLivre(P):

1: $next(P) \leftarrow free$
2: $free \leftarrow P$
3: $P \leftarrow nil$

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

Pilha em alocação encadeada - inserção

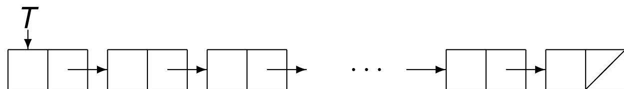


Considere uma pilha cujo topo é apontado pela variável T.
Pilha vazia: T = nil

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

1: $\mathit{ExtraiLivre}(P)$
2: $\mathit{key}(P) \leftarrow Y$
3: $\mathit{next}(P) \leftarrow T$
4: $T \leftarrow P$

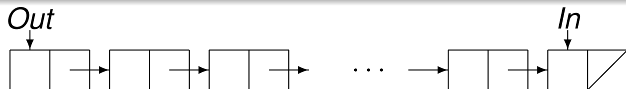
Pilha em alocação encadeada - remoção



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

1: **if** $T = nil$ **then**
2: underflow
3: **else**
4: $P \leftarrow T$
5: $T \leftarrow next(P)$
6: $Y \leftarrow key(P)$
7: $DevolveLivre(P)$
8: **end if**

Fila em alocação encadeada - inserção



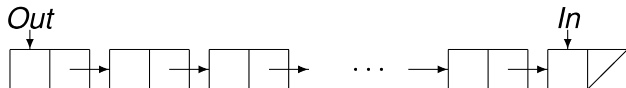
Variáveis apontadoras da fila: *Out* e *In*

Fila vazia: $Out = In$ = nil

InsereFila(Out, In, Y): Insere um elemento novo com informação Y na fila

1: $ExtraiLivre(P)$
2: $key(P) \leftarrow Y$
3: $next(P) \leftarrow nil$
4: **if** In not $= nil$ **then**
5: $\quad next(In) \leftarrow P$
6: $\quad In \leftarrow P$
7: **else**
8: $\quad In \leftarrow P$
9: $\quad Out \leftarrow P$
10: **end if**

Fila em alocação encadeada - remoção



RemoveFila(Out, In, Y): Remove um elemento da fila, colocando a informação do elemento removido em Y

1: **if** $Out = nil$ **then**
2: underflow
3: **else**
4: $P \leftarrow Out$
5: $Out \leftarrow next(P)$
6: $Y \leftarrow key(P)$
7: $DevolveLivre(P)$
8: **if** $Out = nil$ **then**
9: $In \leftarrow nil$
10: **end if**
11: **end if**

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 construídas pelo programador.
- Em certas linguagens, como Pascal, Linguagem C e Modula 2, já existem funções ou rotinas pré-definidas da linguagem para efetuar operações do tipo ExtraiLivre e DevolveLivre.
- Mostramos a seguir como construir a sua própria lista livre e as rotinas para a sua manipulação, para a linguagem Pascal. Embora em Pascal essas rotinas já fazem parte da linguagem, o exemplo pode ser útil para quem quiser implementar as mesmas funções por exemplo em Assembler ou em FORTRAN.

Como construir sua própria lista livre e as rotinas

Usamos um array key e um array next, alocando um total de m elementos para cada array. nil será representado por 0. Variáveis apontadoras contêm realmente índices desses arrays.

```
const m = 1024;qualquer outro valor serve
var key,next: array[1..m] of integer;
free: integer;
procedure initialize;
criação da lista livre inicial
var i: integer;
begin
    for i:=1 to m-1 do
    begin
        key[i] := 0;
        next[i] := i+1
    end;
    key[m] := 0;
    next[m] := 0;
    free := 1
end;
```

Rotinas para extrair e devolver um elemento

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

procedure devolve(P: integer);
Devolve um elemento apontado por P à lista livre
begin
        next[P] := free;
        free := P
end;
```

Usando a lista livre assim criada para implementar uma pilha

```
procedure push(var T: integer; Y: integer);
Insere Y numa pilha apontada por T
var P: integer;
begin
      extrai(P);
      key[P] := Y;
      next[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 := key[T];
          P := T;
          T := next[T];
          devolve(P)
      end
end
```

Uso de apontadores em Pascal ou C

Em Pascal, existe um tipo chamado *pointer* ou apontador que é usado para apontar para outras estruturas. Abaixo o record elemento possui um campo next cujo tipo é um apontador (indicado por $\wedge$) a um elemento.

```
type elemento =
    record
        key: integer;
        next: ∧elemento
    end
```

Em C, podemos escrever:

```
typedef
        struct elemento *lista
typedef
        struct elemento {
        int key;
        lista next;
} elemento;
```

Variáveis do tipo apontador

Voltando a Pascal, podemos declarar:

```
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. Usamos a rotina *new*.

Rotina new (Pascal) para criar um novo elemento

```
var x,y : ∧elemento;
new(x);

x∧.key := 40;
x∧.next :=nil;
y := 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.
- O registro apontado por *x* é indicado por $x\wedge$. Assim $x\wedge$.key denota o campo key do registro apontado por *x*. nil é usado para indicar o apontador nulo.
- O comando de atribuição vale para variáveis do tipo apontador, se ambas são do mesmo tipo. Assim podemos escrever $y := x$.

Função malloc (C) para criar um novo elemento

```
lista x = (lista) malloc(sizeof(elemento));
x -> key = 40;
x -> next = null;
```

Rotina dispose para devolver um elemento inútil

```
dispose(x);
```

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

Em C, usamos free:

```
free(x);
```

Mostramos a seguir o uso dessas rotinas para implementar estruturas de dados.

Rotinas push e pop para pilha T

```
procedure push(var T: ∧elemento; Y: integer);
var P: ∧elemento;
begin
      new(P);
      P∧.key := Y;
      P∧.next := 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∧.key;
            P := T;
            T := T∧.next;
            dispose(P)
      end
end
```

Uso das rotinas push e pop para pilha T

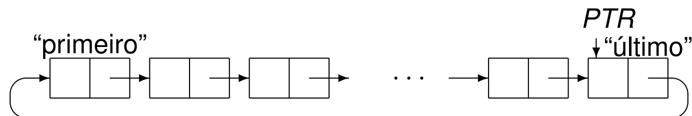
Tendo declarado

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

podemos escrever no programa principal

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

Listas circulares encadeadas

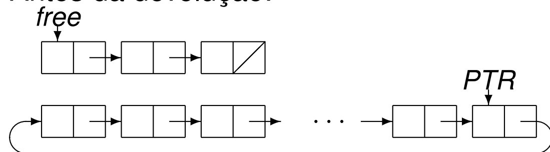


O último nó aponta de volta para o primeiro nó. O apontador PTR aponta para o último nó; assim sendo, next(PTR) dará acesso ao primeiro nó.

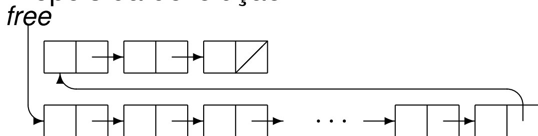
Lista vazia: PTR = nil

Devolução de uma lista circular à lista livre free

Antes da devolução:

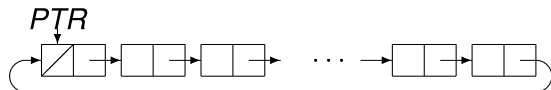


Depois da devolução:

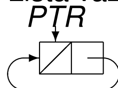



```
if PTRnot = nil then
        free <-> next(PTR)
```

onde <-> significa trocar os valores dos dois apontadores entre si.



Lista vazia: só tem a cabeça



A cabeça possui um valor especial no campo *key*.
Sabemos que demos uma volta pelo valor de *key*.

Listas duplamente ligadas

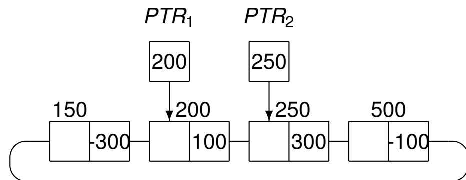
- Um nó possui além do campo key, campos para dois apontadores left e right.
- Na lista vazia: Tanto left como right apontam para a cabeça.
- Uma vantagem óbvia é a possibilidade de percorrer a lista nos dois sentidos (tanto para a direita como para a esquerda).
- Uma outra vantagem é a possibilidade 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:

  1: $right(left(x)) \leftarrow right(x)$
  2: $left(right(x)) \leftarrow left(x)$
  3: $devolve(x)$

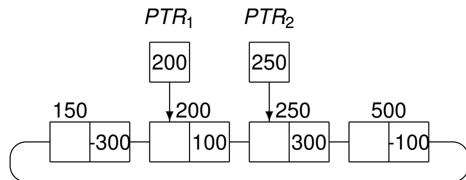
- Analogamente, podemos inserir um elemento novo, apontado por P, à direita ou esquerda de um elemento apontado por x.

Como percorrer nos 2 sentidos sem usar 2 ponteiros

- Usando de dois campos em cada nó (right e left) nos permite percorrer nos dois sentidos.
- Usando apenas um campo, será ainda possível percorrer a lista nos dois sentidos?
- SIM: como se mostra no livro de Knuth. Basta colocar no campo next 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.



Percorrer para esquerda e para direita

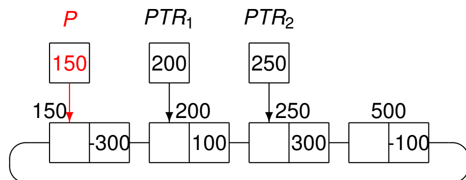


Andar um passo para esquerda:

1: $P \leftarrow PTR2 - next(PTR1)$
2: $PTR2 \leftarrow PTR1$
3: $PTR1 \leftarrow P$

Andar um passo para direita:

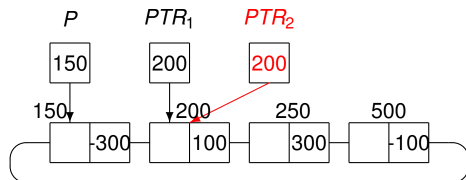
1: $P \leftarrow PTR1 + next(PTR2)$
2: $PTR1 \leftarrow PTR2$
3: $PTR2 \leftarrow P$



Andar um passo para esquerda:

1: $P \leftarrow PTR2 - next(PTR1)$
2: $PTR2 \leftarrow PTR1$
3: $PTR1 \leftarrow P$

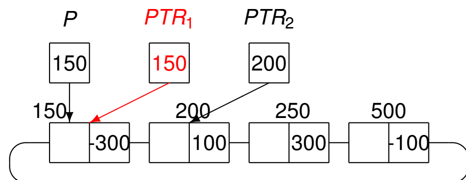
Simulação: percorrer para esquerda



Andar um passo para esquerda:

1: $P \leftarrow PTR2 - next(PTR1)$
2: $PTR2 \leftarrow PTR1$
3: $PTR1 \leftarrow P$

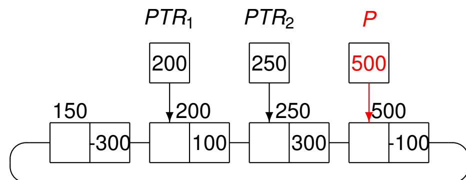
Simulação: percorrer para esquerda



Andar um passo para esquerda:

1: $P \leftarrow PTR2 - next(PTR1)$
2: $PTR2 \leftarrow PTR1$
3: $PTR1 \leftarrow P$

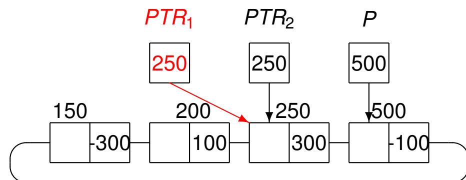
Simulação: percorrer para direita



Andar um passo para direita:

1: $P \leftarrow PTR1 + next(PTR2)$
2: $PTR1 \leftarrow PTR2$
3: $PTR2 \leftarrow P$

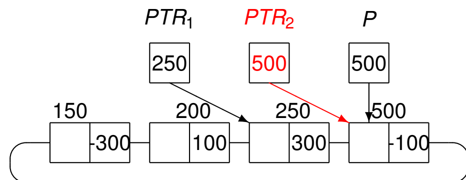
Simulação: percorrer para direita



Andar um passo para direita:

1: $P \leftarrow PTR1 + next(PTR2)$
2: $PTR1 \leftarrow PTR2$
3: $PTR2 \leftarrow P$

Simulação: percorrer para direita



Andar um passo para direita:

1: $P \leftarrow PTR1 + next(PTR2)$
2: $PTR1 \leftarrow PTR2$
3: $PTR2 \leftarrow P$

Este é um bom exercício que usa várias estruturas de dados.

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

1. Transitividade: se $x \preceq y$ e $y \preceq z$ então $x \preceq z$.
2. Anti-simétrica: se $x \preceq y$ e $y \preceq x$ então $x = y$.
3. Reflexividade: $x \preceq x$.

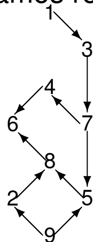
Se $x \preceq y$ e x not $\preceq y$, então escreveremos $x \prec y$ e diremos que “x precede y”.
Das propriedades (1), (2) e (3) acima temos:

1. Se $x \prec y$ e $y \prec z$ então $x \prec z$.
2. Se $x \prec y$ então y not $\prec x$.
3. x not $\prec x$.

Exemplos de ordenações parciais

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

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

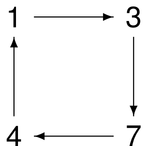


representa as relações

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

O diagrama é acíclico

A propriedade (2) Anti-simétrica: se $x \preceq y$ e $y \preceq x$ então $x = y$ significa que não existem ciclos fechados. Assim, o seguinte não é ordenação parcial.



O problema da ordenação topológica

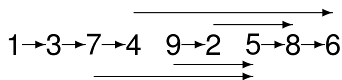
Dada uma ordenação parcial, uma ordenação topológica é uma seqüência

$$a_1 a_2 \ldots a_n$$

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

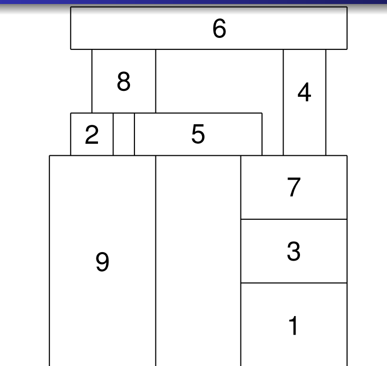
1 3 7 4 9 2 5 8 6

Usando o diagrama,



Todas as "flechas" apontam para a direita.

Construção do monumento de Stonehenge :-)



As precdências podem representar qual pedra suporta qual outra pedra.

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

E a ordenação topológica dá uma possível ordem de colocação das pedras:
1 3 7 4 9 2 5 8 6

Um algoritmo de ordenação topológica

Sejam n objetos numerados de 1 a n.
A entrada é n e pares da forma $j \ \ k$ onde cada par significando $j \prec k$. O último par contém 0 0 indicando o fim dos dados.

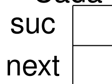
Usamos uma tabela seqüencial $x[1], x[2], \ldots, x[n]$ onde cada $x[k]$ tem a forma

count[k]	
top[k]	

- 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.

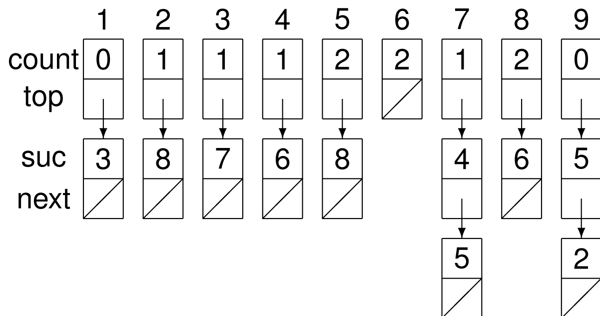
Lista de sucessores

Cada elemento da lista de sucessores diretos de k 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

I. [inicialização]

1: read(n)
2: $AindaSobrou \leftarrow n$
3: **for** $i \leftarrow 1$ **until** n **do**
4: $\quad count[i] \leftarrow 0$
5: $\quad top[i] \leftarrow nil$
6: **end for**

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

1: read(j, k)
2: **while** j not = 0 **do**
3: $\quad count[k] \leftarrow count[k] + 1$
4: $\quad extrai(P)$
5: $\quad suc(P) \leftarrow k$
6: $\quad next(P) \leftarrow top[j]$
7: $\quad top[j] \leftarrow P$
8: $\quad$ read (j, k)
9: **end while**

O algoritmo de ordenação topológica

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 nextf, que guarda apontadores para os elementos dessa fila.]

1: $ln \leftarrow 0$
2: $nextf[0] \leftarrow 0$
3: **for** $i \leftarrow 1$ **until** n **do**
4:     **if** $count[i] = 0$ **then**
5:         $nextf[ln] \leftarrow i$
6:         $ln \leftarrow i$
7:     **end if**
8:     $Out \leftarrow nextf[0]$
9: **end for**

O algoritmo de ordenação topológica

IV. [Produção de uma ordenação topológica.]

```
1:  while Out not = 0 do
2:      write(Out)
3:      P ← top[Out]
4:      AindaSobrou ← AindaSobrou − 1
5:      while P not = nil do
6:          count[suc(P)] ← count[suc(P)] − 1
7:          if count[suc(P)] = 0 then
8:              nextf[In] ← suc(P)
9:              In ← suc(P)
10:         end if
11:         P ← next(P)
12:     end while
13:     Out ← nextf[Out]
14: end while
```

O algoritmo de ordenação topológica

V. [Verificação final.]

1: **if** *AindaSobrou* > 0 **then**
2: erro: existe um ciclo fechado com AindaSobrou elementos, não dá para produzir uma ordenação topológica.
3: **end if**