Estruturas de Informação
Aula 8: Listas (parte 2)

10/04/2008

Fontes Bibliográficas

- Livros:
  - Projeto de Algoritmos (Nivio Ziviani): Capítulo 3;
  - Introdução a Estruturas de Dados (Celes, Cerqueira e Rangel): Capítulo 10;
  - Estruturas de Dados e seus Algoritmos (Szwarefiter, et. al): Capítulo 2;
  - Algorithms in C (Sedgewick): Capítulo 3;
- Slides baseados nas transparências disponíveis em: http://www.dcc.ufmg.br/algoritmos/transparencias.php

TAD Lista (1)

/* Faz a lista ficar vazia */

- FLVazia(Lista).
  - Input: L (Lista)
  - Output: L'
  - Pré-condição: L é definida
  - Pós-condição: L' é definida e vazia

/* Insere x após o último elemento da lista */

- Insere(x, Lista). Insere x após o último
  - Input: x (Item da Lista) e L (Lista)
  - Output: L'
  - Pré-condição: L é definida e x é um Item válido da lista
  - Pós-condição: L' é definida e vazia e o elemento item de L' é igual a x

TAD Lista (2)

/*Retorna o item x que está na posição p da lista, retirando-o da lista e deslocando os itens a partir da posição p+1 para as posições anteriores */

- Retira(p, Lista, x)
  - Input: p (posição válida da lista) e L (Lista)
  - Output: x (item da lista da posição p)
  - Pré-condição: L é definida e p é uma posição válida da lista
  - Pós-condição: L' é a lista L sem o item x, com todos os itens deslocados de uma posição

TAD Lista (3)

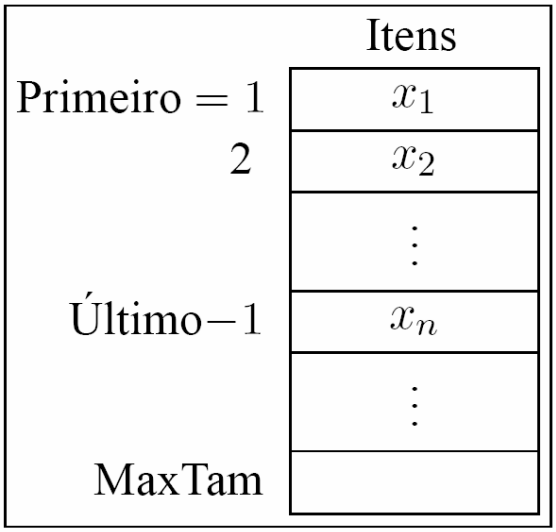
/*Verifica se a lista está vazia*/

- Vazia(Lista)
  - Input: L (Lista)
  - Output: B (*true* se lista vazia; senão retorna *false)*
  - Pré-condição: L é definida
  - Pós-condição: L não é modificada
- /*Imprime os itens da lista na ordem de ocorrência */
- Imprime(Lista)
  - Input: L (Lista)
  - Output:nenhum
  - Pré-condição: L é definida e não está vazia
  - Pós-condição: L não é modificada e seus elementos são impressos

Implementação de Listas Lineares

- Há varias maneiras de implementar listas lineares.
- Cada implementação apresenta vantagens e desvantagens particulares.
- Vamos estudar duas maneiras distintas
  - Usando alocação sequencial e estática (com vetores).
  - Usando alocação não sequencial e dinâmica (com ponteiros): *Estruturas Encadeadas.*

Listas Lineares em Alocação Seqüencial e Estática (2)



Estrutura de Listas com Alocação Seqüencial e Estática (2)

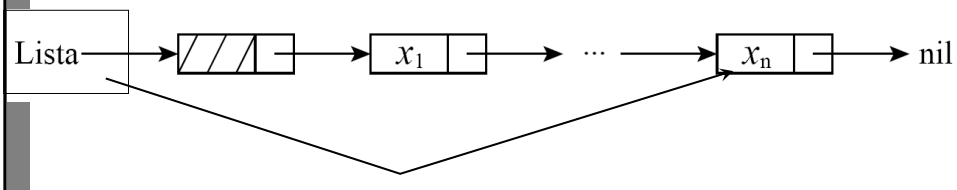
```
#define InicioVetor 1
#define MaxTam  1000

typedef int TipoChave;
typedef int Ponteiro;

typedef struct {
  TipoChave Chave;
  /* outros componentes */
} TipoItem;

typedef struct {
  TipoItem Item[MaxTam];
  Ponteiro Primeiro, Ultimo;
} TipoLista;
```

Listas com alocação não sequencial e dinâmica

- Cada item é encadeado com o seguinte mediante uma variável do tipo Ponteiro.
- Permite utilizar posições não contíguas de memória.
- É possível inserir e retirar elementos sem necessidade de deslocar os itens seguintes da lista.
- Há uma célula cabeça para simplificar as operações sobre a lista
- Estrutura Encadeada

Listas com alocação não sequencial e dinâmica

- Cada item é encadeado com o seguinte mediante uma variável do tipo Ponteiro.
- Permite utilizar posições não contíguas de memória.
- É possível inserir e retirar elementos sem necessidade de deslocar os itens seguintes da lista.
- Há uma célula cabeça para simplificar as operações sobre a lista



Estrutura da Lista com Alocação não Sequencial e Dinâmica

- A lista é constituída de células.
- Cada célula contém um item da lista e um ponteiro para a célula seguinte.
- O registro (struct) TipoLista contém um ponteiro para a célula cabeça e um ponteiro para a última célula da lista.

Estrutura da Lista com Alocação não Sequencial e Dinâmica (2)

```
typedef int TipoChave;
typedef struct {
  TipoChave Chave;
  /* outros componentes */
} TipoItem;
typedef struct Celula_str *Ponteiro;
typedef struct Celula_str {
  TipoItem Item;
  Ponteiro Prox;
} Celula;
typedef struct {
  Ponteiro Primeiro, Ultimo;
} TipoLista;
```

Implementação TAD Lista com Ponteiros

```
void FLVazia (TipoLista *Lista)
{
  Lista->Primeiro = (Ponteiro) malloc
  (sizeof (Celula));
  Lista->Ultimo = Lista->Primeiro;
  Lista->Primeiro->Prox = NULL;
}

int Vazia (TipoLista Lista)
{
 return (Lista.Primeiro == Lista.Ultimo);
}
```

Implementação TAD Lista com Ponteiros (2)

```
void Insere (TipoItem x, TipoLista *Lista)
{
  Lista->Ultimo->Prox = (Ponteiro)
  malloc(sizeof(Celula));
  Lista->Ultimo = Lista->Ultimo->Prox;
  Lista->Ultimo->Item =x;
  Lista->Ultimo->Prox = NULL;
}
```

Implementação TAD Lista com Ponteiros (3)

```
/* O item retirado é o seguinte apontado por p */
void Retira (Ponteiro p, TipoLista *Lista, TipoItem
  *Item)
{
  Ponteiro q;
  if (Vazia(*Lista) || p == NULL || p->Prox == NULL)
  {   printf ("ERRO: Lista vazia ou posicao
      nao existe\n");
      return;
  }
  q = p->Prox;
  *Item = q->Item;
  p->Prox = q -> Prox;
  if (p->Prox == NULL) Lista->Ultimo = p;
  free (q);
}
```

Implementação TAD Lista com Ponteiros(4)

```
void Imprime (TipoLista Lista)
{
  Ponteiro Aux;
  Aux = Lista.Primeiro->Prox;
  while (Aux != NULL)
  {
     printf ("%d\n", Aux->Item.Chave);
     Aux = Aux->Prox;
  }
}
```

Lista com alocação não sequencial e dinâmica: vantagens e desvantagens

- Vantagens:
  - Permite inserir ou retirar itens do meio da lista a um custo constante (importante quando a lista tem de ser mantida em ordem).
  - Bom para aplicações em que não existe previsão sobre o crescimento da lista (o tamanho máximo da lista não precisa ser definido *a priori*).
- Desvantagem: utilização de memória extra para armazenar os apontadores.

Exercício Vestibular

- Num vestibular, cada candidato tem direito a três opções para tentar uma vaga em um dos sete cursos oferecidos.
- Para cada candidato é lido um registro:
  - Chave: número de inscrição do candidato.
  - NotaFinal: média das notas do candidato.
  - Opção: vetor contendo a primeira, a segunda e a terceira opções de curso do candidato.

```
int Chave; /* variando de 1 a 999 */
int NotaFinal; /* variando de 0 a 10 */
int Opcao[3]; /* variando de 1 a 7 */
```

- Problema: distribuir os candidatos entre os cursos, segundo a nota final e as opções apresentadas por candidato.
- Em caso de empate, os candidatos serão atendidos na ordem de inscrição para os exames.

Exercício Vestibular: Solução Possível

- Ordenar registros pelo campo NotaFinal, respeitando a ordem de inscrição;
- Percorrer cada conjunto de registros com mesma NotaFinal, começando pelo conjunto de NotaFinal 10, seguido pelo de NotaFinal 9, e assim por diante.
  - Para um conjunto de mesma NotaFinal tenta-se encaixar cada registro desse conjunto em um dos cursos, na primeira das três opções em que houver vaga (se houver).

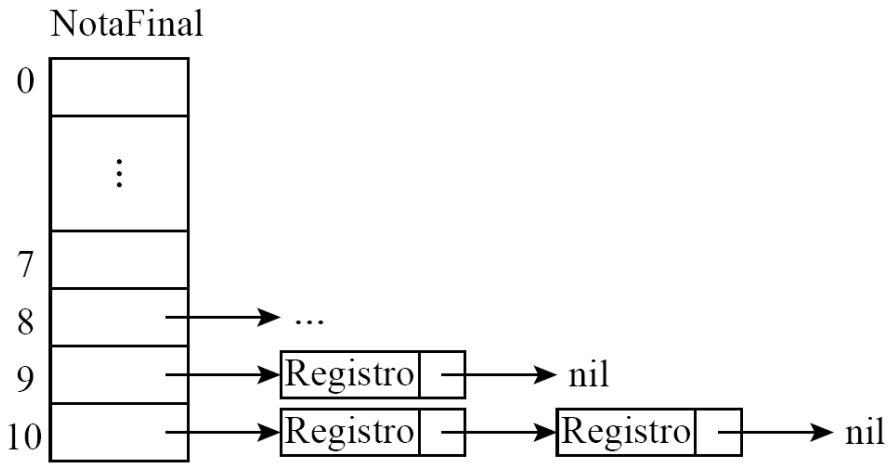
Exercício Vestibular: algoritmo

```
/*programa Vestibular */
void main()
{
   int Nota;
   ordena os registros pelo campo NotaFinal;
   for (Nota=10; nota >= 0 ; nota --)
   { while houver registro com mesma nota do
     { if existe vaga em um dos cursos de opcao do
     candidato
       { insere registro no conjunto de aprovados}
       else insere registro no conjunto de reprovados;
     }
   }
   imprime aprovados por curso ; imprime reprovados;
}
```

Exercício Vestibular: Classificação dos Alunos

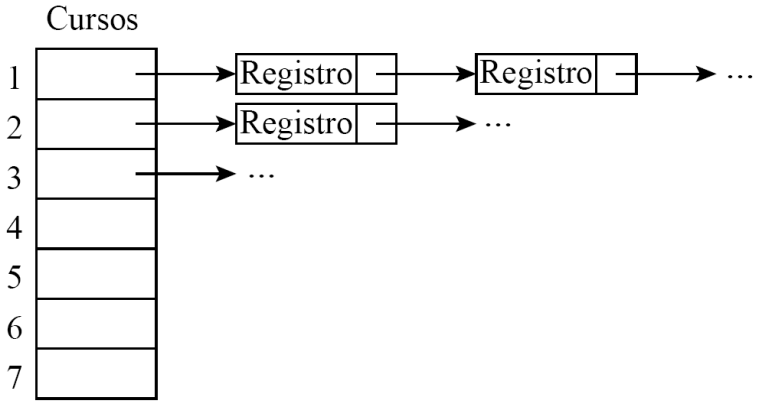
- Uma boa maneira de representar um conjunto de registros é com o uso de listas.
- Ao serem lidos, os registros são armazenados em listas para cada nota.
- Após a leitura do último registro os candidatos estão automaticamente ordenados por NotaFinal.
- Dentro de cada lista, os registros estão ordenados por ordem de inscrição, desde que os registros sejam lidos na ordem de inscrição de cada candidato e inseridos nesta ordem

Exercício Vestibular: Classificação dos Alunos (2)

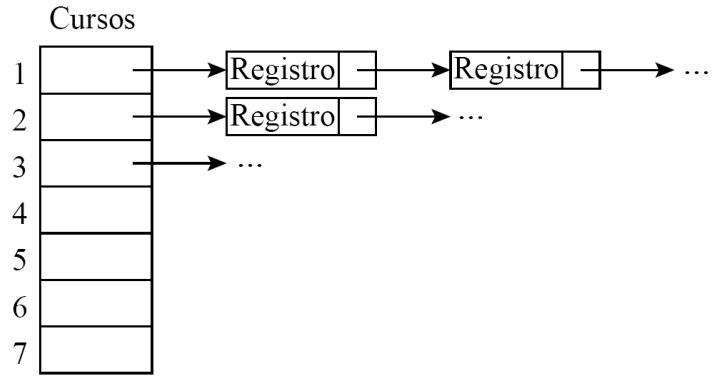


Exercício Vestibular: Classificação dos Alunos (3)

- As listas de registros são percorridas, iniciando-se pela de NotaFinal 10, seguida pela de NotaFinal 9, e assim sucessivamente.
- Cada registro é retirado e colocado em uma das listas abaixo, na primeira das três opções em que houver vaga.



Exercício Vestibular: Classificação dos Alunos (4)



- Se não houver vaga, o registro é colocado em uma lista de reprovados.
- Ao final, a estrutura acima conterá a relação de candidatos aprovados em cada curso.

Exercício Vestibular: algoritmo mais detalhado

```
/* programa Vestibular */
void main()
{
  int Nota;
  TipoChave Chave;
  lê o número de vagas para cada curso;
  inicializa listas de classificação, de aprovados e
  de reprovados;
  lê registro; /*formato definido anteriormente*/
  while (Chave != 0)
  { insere registro nas listas de classificação,
            conforme nota final; lê registro;
  }
```

Exercício Vestibular: algoritmo mais detalhado (2)

```
…
  for (Nota=10; nota >= 0 ; nota --)
   { while houver próximo registro com mesma NotaFinal
     { retira registro da lista;
       if existe vaga em um dos cursos de opcao do
     candidato
       {insere registro no conjunto de aprovados;
        decrementa o número de vagas para aquele curso;}
       else insere registro no conjunto de reprovados;
       obtém próximo registro;
     }
   }
   imprime aprovados por curso ; imprime reprovados;
}
```

Estrutura Final da Lista

```
#define NOpcoes   3
#define NCursos   7

typedef int TipoChave;
typedef struct {
  TipoChave Chave;
  int NotaFinal;
  int Opcao [NOpcoes];
} TipoItem;
typedef struct Celula_str* Ponteiro;
typedef struct Celula_str {
  TipoItem Item;
  Ponteiro Prox;
}Celula;
typedef struct {
  Ponteiro Primeiro, Ultimo;
}TipoLista;
```

Ex. Vestibular: Detalhamento Final

```
void LeRegistro (TipoItem* Registro)
{ /* valores lidos sao separados por brancos */
  int i;
  scanf ("%d%d, &Registro->Chave, &Registro->NotaFinal);
  for (i=0; i<NOpcoes; i++) scanf ("%d", &Registro-
  >Opcao[i]);
}
```

- Redirecionamento de entrada padrão
  - nome_programa < nome_arquivo
- Redirecionamento de saída padrão
  - nome_programa > nome_arquivo

Ex. Vestibular: Detalhamento Final (2)

```
#define NOpcoes   3
#define NCursos   7
#define FALSE     0
#define TRUE      1

/* estruturas definidas anteriormente */
TipoItem Registro;
TipoLista Classificacao[11];
TipoLista Aprovados[NCursos];
TipoLista Reprovados;
int Vagas [NCursos];
short Passou;
int i, Nota;
```

Ex. Vestibular: Detalhamento Final (3)

```
/*Considere as funcoes do TAD lista e a funcao LeRegitro*/

for (i=1;i<=NCursos; i++) scanf ("%d", &Vagas[i-1]);
for (i=0; i<=10; i++) FLVazia(&Classificacao[i]);
for (i=1; i<=NCursos; i++) FLVazia(&Aprovados[i-1]);
FLVazia(&Reprovados); LeRegistro(&Registro);
while (Registro.Chave != 0)
{
  Insere (Registro, &Classificacao[Registro.NotaFinal]);
  LeRegistro(&Registro);
}
for (Nota = 10; Nota >= 0; Nota --)
{while (!Vazia (Classificacao[Nota]))
 { Retira (Classificacao[Nota].Primeiro,
   &Classificacao[Nota], &Registro);
```

Ex. Vestibular: Detalhamento Final (4)

```
  i = 1; Passou = FALSE;
  while (i<= NOpcoes && !Passou)
  {
   if (Vagas[Registro.Opcao[i-1]-1]>0)
   {Insere (Registro, &Aprovados[Registro.Opcao[i-1]–1]);
    Vagas[Registro.Opcao[i-1]-1]--; Passou = TRUE;
   } /*end do if*/
   i++;
  } /*end do while*/
  if (!Passou) Insere(Registro, &Reprovados);
  } /*end do while*/
 } /*end do for*/
 for (i=1; i<=NCurso; i++)
 {printf ("Relacao dos aprovados no Curso %d\n", i);
  Imprime (Aprovados [i-1];
 }
 printf ("Relacao dos reprovados\n");
 Imprime(Reprovados);
 return 0;
}
```

Exercício Vestibular: observações

- Observe que o programa é completamente independente da implementação do tipo abstrato de dados Lista.
- O exemplo mostra a importância de utilizar **tipos abstratos de dados** para escrever programas, em vez de utilizar detalhes particulares de implementação.
- Altera-se a implementação rapidamente. Não é necessário procurar as referências diretas às estruturas de dados por todo o código.
- Este aspecto é particularmente importante em programas de grande porte.