

ESTATÍSTICA E DADOS

Clodoaldo A M Lima

15 de Março de 2013

Tipos de Listas

Lista I (Lista Estática – implementação via arranjo)

```
#define MAX 10

typedef int telem;
typedef struct {
    telem v[MAX];
    int n;
} tlista;
```

```
#define MAX 10

typedef struct {
    int chave;
    int valor;
} ITEM;
//representa a lista de itens

typedef struct {
    ITEM itens[TAM];
    int fim;
} tlista;
```

Tipos de Listas

Lista I (Lista Estática – implementação via arranjo)

```
#define MAX 10

typedef int telem;
typedef struct {
    telem v[MAX];
    int n;
} tlista;
```

```
#define MAX 10

typedef struct {
    int chave;
    int valor;
} ITEM;
//representa a lista de itens

typedef struct {
    ITEM itens[TAM];
    int fim;
} tlista;
```

Operações

- Criar lista
 - Pré-condição: nenhuma
 - Pós-condição: inicia a estrutura de dados
- Limpar lista
 - Pré-condição: nenhuma
 - Pós-condição: coloca a estrutura de dados no estado inicial

Operações

- Inserir item (última posição)
 - Pré-condição: a lista não está cheia
 - Pós-condição: insere um item na última posição, retorna true se a operação foi executada com sucesso, false caso contrário
- Inserir item (por posição)
 - Pré-condição: a lista não está cheia
 - Pós-condição: insere um item na posição informada, que deve estar dentro da lista, retorna true se a operação foi realizada com sucesso, false caso contrário

Operações

- Remover item (por posição)
 - Pré-condição: uma posição válida da lista é informada
 - Pós-condição: o item na posição fornecida é removido da lista, retorna true se a operação foi executada com sucesso, false caso contrário.
- Recuperar item (dado uma chave)
 - Pré-condição: nenhuma
 - Pós-condição: recupera o item dado uma chave, retorna true se a operação foi executada com sucesso, false caso contrário.

Operações

- Recuperar item (por posição)
 - Pré-condição: uma posição válida da lista é informada
 - Pós-condição: recupera o item na posição fornecida, retorna true se a operação foi executada com sucesso, false caso contrário.
- Contar número de itens
 - Pré-condição: nenhuma
 - Pós-condição: retorna o número de itens na lista.

Operações

- Verificar se a lista está vazia
 - Pré-condição: nenhuma
 - Pós-condição: retorna true se a lista estiver vazia e false caso-contrário
- Verificar se a lista está cheia
 - Pré-condição: nenhuma
 - Pós-condição: retorna true se a lista estiver cheia e false caso-contrário.
- Imprime lista
 - Pré condição: nenhuma
 - Pós-condição: imprime na tela os itens da lista

Exercício – ListaEstatica.h

- `#define TAM 100`
- `//representa o item armazenado`
- `typedef struct {`
 - `int chave;`
 - `int valor;``} ITEM;`
- `//representa a lista de itens`
- `typedef struct {`
 - `ITEM itens[TAM];`
 - `int fim;``} tlista;`

ListaEstatica.h

- `//funções que manipulam a lista`
- `void criar(tlista *lista);`
- `void imprimir(tlista *lista);`
- `int vazia(tlista *lista);`
- `int cheia(tlista *lista);`
- `int inserir_fim(tlista *lista, ITEM *item);`
- `int insere_posicao (tlista *lista, int pos, ITEM *item);`
- `int remover(tlista *lista, int pos);`
- `int tamanho(tlista *lista);`
- `void limpa(tlista *lista);`

ListaEstatica.h

- **int** verifica_ordem(tlista *lista);
 - 0 não ordenada, 1 - crescente, 2 decrescente
- **void** concatena(tlista *lista1, tlista *lista2);
 - Faz uma copia de L1 em outra L2
- **void** concatena_sem_rep(tlista *lista1, tlista *L2);
 - Faz uma copia de L1 em outra L2 sem repetição
- **void** inverte(tlista *lista);
 - Faz a inversão da lista
- **tlista*** intercalar(tlista *lista1, tlista *lista2);
 - Intercalar L1 com L2, gerando L3 ordenada (considere L1 e L2 ordenadas)
- **void** eliminar_ocorencia(tlista *lista1, item *dado)
 - Elimina da lista todas as ocorrencias de um determinado dado item

Tipo de Listas

Listas Estáticas Ordenadas

Listas Ordenadas

- Características
 - Uma lista pode estar em ordem **crescente/decrescente** segundo o valor de alguma chave
 - Uma lista pode ser mantida em ordem **crescente/decrescente** segundo o valor de **alguma chave**
 - Esta ordem facilita a pesquisa de itens
 - Por outro lado, a **inserção e remoção** são mais complexas pois deve manter os itens ordenados

Tipo Abstrato de Dados

- O TAD listas ordenadas é o mesmo do TAD listas, apenas difere na implementação
- As operações diferentes serão a inclusão e inserção de itens
- Inserir item
 - Pré condição: a lista não está cheia
 - Pós-condição: insere um item em uma posição tal que a lista é mantida ordenada
- Remover item
 - Pré-condição: uma posição válida da lista é informada
 - Pós-condição: o item na posição fornecida é removido da lista, a lista é mantida ordenada

Tipo Abstrato de Dados

- Exemplo de inserção ordenada

1	
2	
3	
4	
5	

Inserir valor 6



1	6
2	
3	
4	
5	

Tipo Abstrato de Dados

- Exemplo de inserção ordenada

1	6
2	
3	
4	
5	

Inserir valor 8



1	6
2	8
3	
4	
5	

Tipo Abstrato de Dados

- Exemplo de inserção ordenada

1	6
2	8
3	
4	
5	

Inserir valor 7



1	6
2	7
3	8
4	
5	

Tipo Abstrato de Dados

- Exemplo de inserção ordenada

1	6
2	7
3	8
4	
5	

Inserir valor 5



1	5
2	6
3	7
4	8
5	

Tipo Abstrato de Dados

- Exemplo de inserção ordenada

1	5
2	6
3	7
4	8
5	

Inserir valor 3



1	3
2	5
3	6
4	7
5	8

Implementação – Inserção Ordenada

```
int inserir_ordenado(LISTA *lista, ITEM *item){
    if (!cheia(lista)) { //verifica se existe espaço
        int pos = lista->fim;
        //move os itens até encontrar a posição de inserção
        while (pos>0&&lista->itens[pos-1].chave > item->chave){
            lista->itens[pos] = lista->itens[pos-1];
            pos--;
        }
        lista->itens[pos] = *item; //insere novo item

        lista->fim++; //incrementa tamanho da lista

        return 1;
    }
    return 0;
}
```

Implementação – Remoção Ordenada

```
int remover(tlista *lista, int pos){  
    /* posição inválida */  
    if ( (pos > lista->fim) || (pos < 1) ) return 0;  
  
    //desloca os itens depois da posição de remoção  
    while (pos<=lista->fim){  
        lista->itens[pos-1] = lista->itens[pos];  
        pos++;  
    }  
  
    lista->fim--; //decrementa o tamanho da lista  
  
    return 1;  
}
```

Tipos

Listas

Busca em Lista

Busca em Listas Estáticas

- Uma tarefa comum a ser executada sobre listas é a busca de itens **dado uma chave**
- **No caso da lista não ordenada**, a busca será sequencial (consultar todos os elementos)

Porém, com uma lista ordenada, diferentes estratégias podem ser aplicadas que aceleram essa busca

- **Busca sequencial “otimizada”**
- **Busca binária**

Busca Sequencial

- Na busca sequencia, a ideia é procurar um elemento que tenha uma determinada chave, começando do início da lista, e parar quando a lista terminar ou quando o elemento for encontrado.

Busca Sequencial

- Quando os elementos estão ordenados, a busca sequencial pode ser “otimizada” (acelerada)
- Vejamos o seguinte exemplo

Busca Sequencial Ordenada

- Exemplo 1
 - Procure pelo valor 4

1	3
2	5
3	6
4	7
5	8



3 é diferente de 4

Busca Sequencial Ordenada

- Exemplo 1
 - Procure pelo valor 4

1	3
2	5
3	6
4	7
5	8

→ 5 é diferente de 4
e
é maior que 4

Busca Sequencial Ordenada

- Exemplo 1
 - Procure pelo valor 8

1	3
2	5
3	6
4	7
5	8

→ 3 é diferente de 8

Busca Sequencial Ordenada

- Exemplo 1
 - Procure pelo valor 8

1	3
2	5
3	6
4	7
5	8

→ 5 é diferente de 8

Busca Sequencial Ordenada

- Exemplo 1
 - Procure pelo valor 8

1	3
2	5
3	6
4	7
5	8

→ 6 é diferente de 8

Busca Sequencial Ordenada

- Exemplo 1
 - Procure pelo valor 8

1	3
2	5
3	6
4	7
5	8

→ 7 é diferente de 8

Busca Sequencial Ordenada

- Exemplo 1
 - Procure pelo valor 8

1	3
2	5
3	6
4	7
5	8

► Chave encontrada

Busca Sequencial

- A lista ordenada permite realizar busca sequencias **mais rápidas** uma vez que, caso a chave procurada não exista, pode-se parar a busca tão logo se encontre um elemento com chave maior que a procurada
- Entretanto, essa melhora não altera a complexidade da busca sequencia, que ainda é $O(n)$. Por que?

Implementa de Busca Sequencial

```
int busca_sequencial(LISTA *lista, int chave, ITEM *item) {
    int i;
    for (i = 0; i <= lista->fim;i++) //percorre a lista
        if (lista->itens[i].chave == chave) {
            //se encontrar a chave
            *item = lista->itens[i]; //armazena o elemento
            return 1; //indica que encontrou
        }
        else if (lista->itens[i].chave > chave){
            //se a chave é maior
            return 0; //indica que não encontrou
        }
    }

    return 0; //indica que não encontrou
}
```

Busca Binária

- A busca binária é um algoritmo de busca mais sofisticado e bem mais eficiente que a busca sequencial
- Entretanto, a busca binária somente pode ser aplicada em estruturas que permite cada elemento em tempo constante, tais como os vetores
- A ideia é, a cada iteração, dividir o vetor ao meio e descartar metade do vetor
- Essa ideia é melhor ilustrada utilizando uma figura

Busca Binária

- Exemplo 1
 - Para procurar pelo valor 30

3	4	6	8	11	15	18	25	30	32
0	1	2	3	4	5	6	7	8	9

Busca Binária

- Exemplo 1
 - Para procurar pelo valor 30

esq									dir
↓									↓
3	4	6	8	11	15	18	25	30	32
0	1	2	3	4	5	6	7	8	9

$$\text{Meio} = (\text{esq} + \text{dir}) / 2 = (0 + 9) / 2 = 4$$

Busca Binária

- Exemplo 1
 - Para procurar pelo valor 30

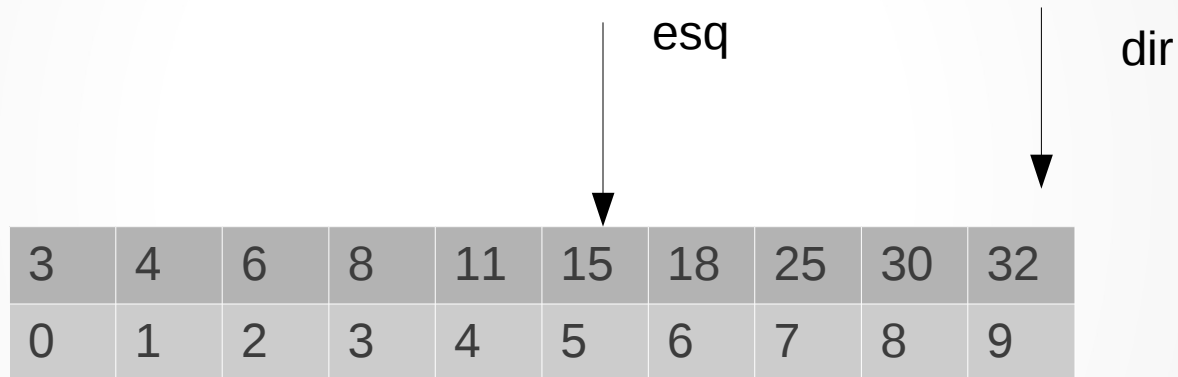
esq				meio					dir
↓				↓					↓
3	4	6	8	11	15	18	25	30	32
0	1	2	3	4	5	6	7	8	9

Itens[meio].chave menor do que 30

esq = meio +1

Busca Binária

- Exemplo 1
 - Para procurar pelo valor 30



3	4	6	8	11	15	18	25	30	32
0	1	2	3	4	5	6	7	8	9

$$\text{Meio} = (\text{esq} + \text{dir}) / 2 = (5 + 9) / 2 = 7$$

Busca Binária

- Exemplo 1
 - Para procurar pelo valor 30

					esq		meio		dir
					↓		↓		↓
3	4	6	8	11	15	18	25	30	32
0	1	2	3	4	5	6	7	8	9

Itens[meio].chave menor do que 30

esq = meio + 1

Busca Binária

- Exemplo 1
 - Para procurar pelo valor 30



3	4	6	8	11	15	18	25	30	32
0	1	2	3	4	5	6	7	8	9

Itens[meio].chave igual 30

Busca Binária

- É importante lembra que
 - Busca binária somente funciona em vetores ordenados
 - Busca sequencial funciona com vetores ordenados ou não
- A binária é muito eficiente. Ela é $O(\log_2 n)$. Para $n = 1000.00$, aproximadamente 20 comparações são necessárias.

Implementação Busca Binária

```
int busca_binaria(LISTA *lista, int chave, ITEM *item){
    int esq = 0;
    int dir = lista->fim;
    while (esq<=dir) {
        int meio (esq+dir)/2; //calcula meio do vetor
        if (lista->itens[meio].chave == chave) {
            //encontrou a chave
            *item = lista -> itens[meio];
            return 1;
        }
        if (lista->itens[meio].chave > chave) {
            //o meio é maior que a chave
            dir = meio -1;
            //desconsidero os itens maiores que o meio
        }else {
            esq = meio +1;
            //desconsidero os itens menores que o meio
        }
    }
    return 0;
}
```

Conclusão

- Pontos fortes
 - Tempo constante de acesso aos dados
- Pontos fracos
 - Custo para inserir e retirar elementos da lista, dada uma posição fornecida pelo
 - Tamanho máximo da lista é (dependendo da linguagem) definido em tempo de compilação

Conclusão

- Quando utilizar
 - Essa implementação simples é mais comumente utilizadas em certas situações
 - Listas pequenas
 - Tamanho máximo da lista é conhecido
 - Poucas ocorrências de utilização dos métodos de inserção e remoção, dada uma posição definida pelo usuário