Algoritmos e Estrutura de Dados

Listas

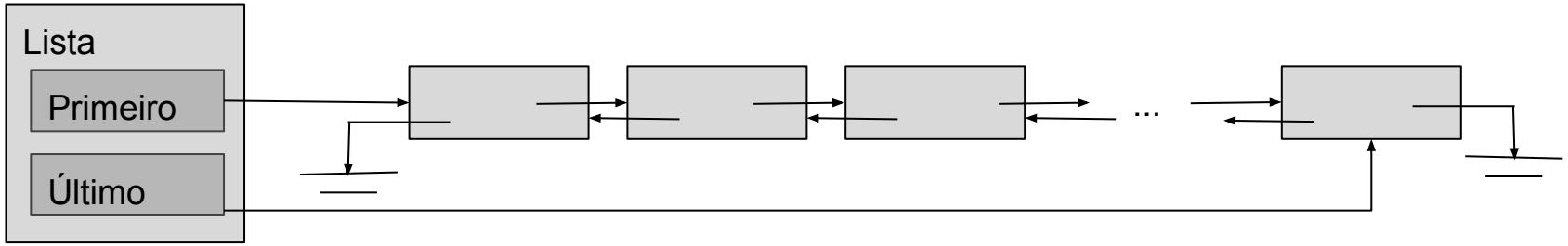
Prof. Nilton Luiz Queiroz Jr.

Listas Ligadas

- Imagine que fosse necessário fazer uma remoção do fim de uma lista ligada;
  - Não seria possível acessar o seu antecessor sem antes varrer a lista toda;
- Para resolver esse problema usamos listas duplamente ligadas;
  - Um nó agora é composto por um ponteiro para o antecessor e um ponteiro para o sucessor;
  - A única desvantagem da utilização deste ponteiro é o espaço ocupado;
  - Porém é possível reduzir o tempo de algumas operações, tal como a remoção no fim da lista;
  - A lista pode ser percorrida facilmente de trás para frente;

Lista Duplamente Ligada

- Como existem ponteiros para o antecessor e o sucessor, tanto o primeiro quanto o último nó seriam aterrados
  - O primeiro não teria antecessor;
  - O último não teria sucessor;

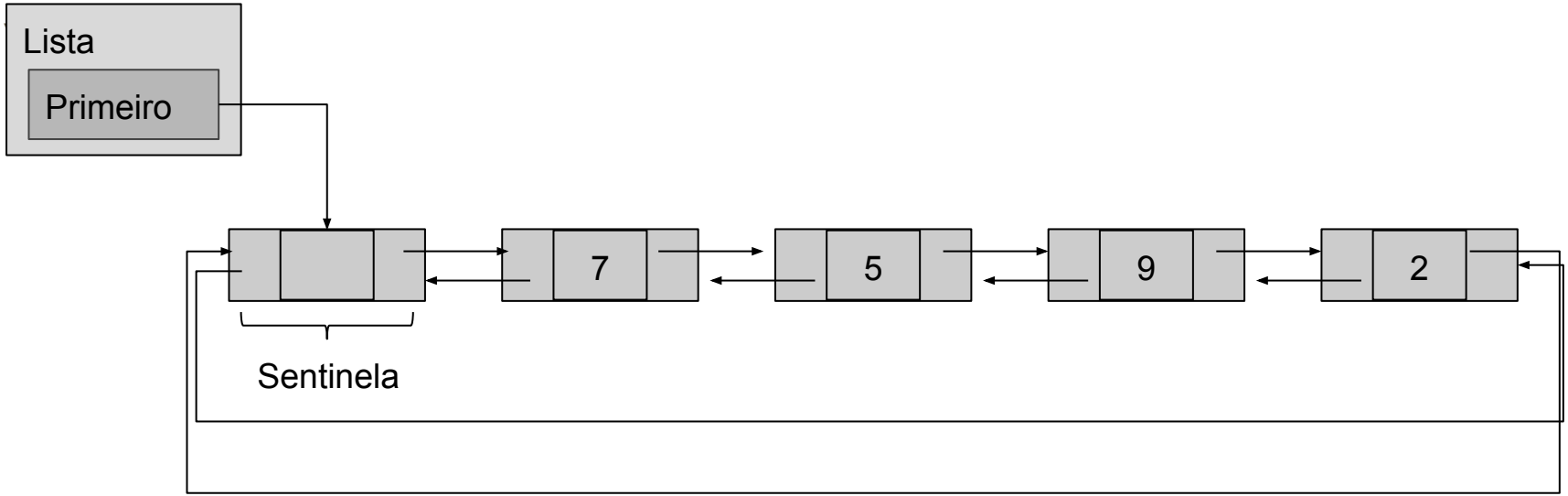


Lista Duplamente Ligada

- As operações de inserção e remoção teriam que manipular ambos ponteiros;
- Além dos ponteiros do próprio elemento a ser inserido seria necessário manipular os ponteiros de seus nós “vizinhos”;

Lista Duplamente Ligada

- As listas duplamente ligadas podem ser lineares ou circulares e podem conter ou não sentinela;



Lista Duplamente Ligada

- Os agregados heterogêneos necessários para uma lista duplamente ligada com sentinela se diferem dos usado em uma lista simplesmente ligada;
  - Precisamos de mais ponteiros;

Implementação

- Agregados heterogêneos

```
struct tipo_item{
        int chave;
        /*outros campos*/
};

struct tipo_celula{
        struct tipo_item item;
        struct tipo_celula *prox;
        struct tipo_celula *ant;
};

struct tipo_lista{
        struct tipo_celula *primeiro;
};
```

Implementação

- A inicialização de uma lista é feita da seguinte maneira:

```
void inicializa(struct tipo_lista *l){
        l->primeiro=(struct tipo_celula *)malloc(sizeof(struct tipo_celula));
        l->primeiro->prox=l->primeiro;
        l->primeiro->ant=l->primeiro;
}
```

Implementação

- Para verificar se uma lista está vazia basta ver se a sentinela aponta para ela mesmo

```
int vazia(struct tipo_lista *l){
        return l->primeiro->prox == l->primeiro;
}
```

Implementação

- A inserção no início é feita de maneira similar a lista ligada com sentinela;
  - Deve-se ajustar o ponteiro de antecessor do elemento inserido e do elemento que estava ocupando a primeira posição
- Com o uso de sentinela não existem casos específicos;
  - Inserções no início são sempre iguais!

Implementação

- Inserção no início

```
void insere_primeiro(struct tipo_lista *l, struct tipo_item x){
        struct tipo_celula *novo;
        novo=(struct tipo_celula *)malloc(sizeof(struct tipo_celula));
        novo->item=x;
        novo->prox = l->primeiro->prox;
        novo->ant = l->primeiro;
        l->primeiro->prox->ant = novo;
        l->primeiro->prox = novo;
}
```

Implementação

- A inserção no fim é muito similar a inserção no início;
  - Os ponteiros ajustados devem ser o antecessor da sentinela e o sucessor do antecessor da sentinela
- Não existem casos específicos

Implementação

- Inserção no fim

```
void insere_ultimo(struct tipo_lista *l, struct tipo_item x){
        /*algoritmo*/
}
```

Implementação

- Inserção no fim

```
void insere_ultimo(struct tipo_lista *l, struct tipo_item x){
        struct tipo_celula *novo;
        novo=(struct tipo_celula *)malloc(sizeof(struct tipo_celula));
        novo->item=x;
        novo->prox=l->primeiro;
        novo->ant=l->primeiro->ant;
        l->primeiro->ant->prox=novo;
        l->primeiro->ant=novo;
}
```

Implementação

- A inserção após a i-ésima posição localiza a posição
  - “Anda” com o ponteiro até que uma variável contadora alcance o valor de p
  - insere a frente do ponteiro;
- O único cuidado deve ser: verificar se a posição foi encontrada
  - Caso tenha se alcançado a sentinela então não se pode inserir após ela

Implementação

- Inserção após a i-ésima posição

```
void insere_apos_posicao(struct tipo_lista *l, struct tipo_item x, int pos){
        /*algoritmo*/
}
```

Implementação

- Inserção após a i-ésima posição

```
void insere_apos_posicao(struct tipo_lista *l, struct tipo_item x, int pos){
        struct tipo_celula *novo,*ptr;
        int i=0;
        ptr=l->primeiro->prox;
        while((ptr!=l->primeiro) && (i<pos)){
                ptr=ptr->prox;
                i++;
        }
        if(ptr!=l->primeiro){
                novo=(struct tipo_celula *)malloc(sizeof(struct tipo_celula));
                novo->item=x;
                novo->ant=ptr;
                novo->prox=ptr->prox;
                ptr->prox->ant=novo;
                ptr->prox=novo;
        }
}
```

Implementação

- Remoção no início deve ajustar os ponteiros do sucessor do primeiro e da sentinela;
  - O antecessor deve apontar para a sentinela, e a sentinela deve apontar para o sucessor do primeiro;
- Não existem casos específicos;

Implementação

- Remoção no início

```
int remove_primeiro(struct tipo_lista *l, struct tipo_item *x){
        /*algoritmo*/
}
```

Implementação

- Remoção no início

```
int remove_primeiro(struct tipo_lista *l, struct tipo_item *x){
        struct tipo_celula *ptr;
        if(!vazia(l)){
                ptr=l->primeiro->prox;
                *x=ptr->item;
                ptr->prox->ant = ptr->ant;
                ptr->ant->prox = ptr->prox;
                free(ptr);
                return 1;
        }else{
                return 0;
        }
}
```

Implementação

- A busca usa um ponteiro que para quando acha o elemento ou chega na sentinela;
  - Se chegou na sentinela então não foi encontrado;
    - Retorna NULL;
  - Se não chegou então achou
    - Retorna o ponteiro usado para percorrer

Implementação

- Busca

```
struct tipo_celula* buscar(struct tipo_lista *l, int chave){
        /*algoritmo*/
}
```

Implementação

- Busca

```
struct tipo_celula* buscar(struct tipo_lista *l, int chave){
        struct tipo_celula *ptr;
        ptr=l->primeiro->prox;
        while((ptr!=l->primeiro) &&  (chave != ptr->item.chave)){
                ptr=ptr->prox;
        }
        if(ptr!=l->primeiro){
                return ptr;
        }else{
                return NULL;
        }
}
```

Implementação

- Para remover por chave basta buscar o elemento:
  - Caso encontre ajuste os ponteiros e libere a memória do elemento apontado;
- Não existem casos específicos;

Implementação

- Remoção por chave

```
int remove_chave(struct tipo_lista *l, int chave, struct tipo_item *x){
        /*algoritmo*/
}
```

Implementação

- Remoção por chave

```
int remove_chave(struct tipo_lista *l, int chave, struct tipo_item *x){
        struct tipo_celula *ptr;
        ptr=buscar(l,chave);
        if(ptr==NULL){
                return 0;
        }else{
                *x=ptr->item;
                ptr->prox->ant = ptr->ant;
                ptr->ant->prox = ptr->prox;
                free(ptr);
                return 1;
        }
}
```

Exercícios

1. Faça um procedimento que imprima todos valores e chaves em uma lista duplamente ligada com sentinela;
2. Faça uma função que informe o tamanho de uma lista duplamente ligada com sentinela

   Obs: A sentinela não deve contar como elemento, ou seja, uma lista somente com a sentinela deve ter tamanho 0, e uma lista com sentinela e mais um elemento o tamanho será 1;

3. Implemente a função de remoção na posição em uma lista duplamente ligada circular com sentinela

Exercícios

4. Faça um subprograma de rotação para a direita em lista duplamente ligada circular com sentinela. Seu subprograma deve receber a lista e um valor inteiro de quantas posições é a rotação:
   Exemplos:
   Dada a lista:
   L = [5, 1, 3, 9, 2, 7]
   após aplicar rotaciona_direita(L, 2) teremos
   L = [2, 7, 5, 1, 3, 9]
   Dada a lista:
   L2 = [1, 7, 2, 6, 4, 9, 0]
   após aplicar rotaciona_direita(L, 5) teremos
   L2= [2, 6, 4, 9, 0, 1, 7]