Universidade Federal do Espírito Santo

Departamento de Computação e Eletrônica - CEUNES
Estrutura de Dados I
Prof. Oberlan Romão

Listas Simplesmente Encadeadas

Atenção: Ao terminar, não se esqueça de enviar as soluções no AVA.

O objetivo dessa aula prática é nos aprofundarmos nos conceitos de lista simplesmente encadeadas. Antes de iniciar, baixe o arquivo `Lista.zip` no AVA.

Considerando o `struct lista` que contém apenas um ponteiro para o primeiro nó da lista (mostrado abaixo), algumas funções (como obter o tamanho da lista, inserir no fim ou pegar o último nó) precisam fazer $O(n)$ operações, uma vez que temos que percorrer todos os n nós da lista para obter a informação desejada.

```
struct lista {
    No *inicio;
};
```

Podemos melhorar a complexidade dessas funções e simplificar outras adicionando novos campos ao `struct lista` (arquivo `Lista.c`). Em especial, podemos ter um campo que armazena o tamanho da lista e outro que aponta para o último nó da lista. Dessa forma, o `struct lista` fica da seguinte forma:

```
struct lista {
    No *inicio;
    No *fim;
    int tamanho;
};
```

Como você pode imaginar, agora temos que ter atenção ao criarmos a lista e ao adicionarmos e/ou removermos algum novo nó, de forma que a estrutura permaneça consistente sempre que uma dessas operações forem executadas. Inicialmente, vamos completar a função `criaLista`. Como devemos inicializar cada um desses novos campos na função?

```
Lista *criaLista() {
    Lista *lista = malloc(sizeof(Lista));
    lista->inicio = NULL;
    lista->fim =     /* Qual valor esse campo deve ser inicializado? */
    lista->tamanho = /* Qual valor esse campo deve ser inicializado? */
    return lista;
}
```

Vamos agora corrigir as funções responsáveis por inserir um novo nó na lista e implementarmos outras considerando esse novo `struct lista`. Começamos pela função que insere no início da lista (função `insereInicioLista`). Devemos modificar o campo `fim`? Se sim, em que caso?

Exercício 1: Compile o código e verifique se o resultado do `/** Exercicio 1 **/` saiu como esperado.

Para inserirmos um nó no **fim** da lista, sem o campo `fim`, devemos percorrer toda a lista até encontrarmos o último nó. Então, inserimos o nó. Dessa forma, essa função precisa fazer $O(n)$ operações. Entretanto, com o auxilio do campo `fim` podemos melhorar essa complexidade.

Exercício 2: Implemente a função `insereFimLista`. Descomente a parte do `/** Exercicio 2 **/` (`main.c`) para verificar a sua implementação.

Exercício 3: Na aula teórica, implementamos uma função que faz uma busca por um determinado valor entre os nós da lista. Entretanto, é comum precisamos procurar um nó em específico (não o valor que está no nó). Implemente a função `No* procuraNoLista(Lista *lista, No *p)`.

Exercício 4: Outra função relacionada a busca é encontrar o n-ésimo elemento da lista. Implemente a função `No* n_esimo(Lista *lista, int n)`.

Exercício 5: Analise a função `void removeElementoLista(Lista *lista, int valor)`. Note que ela não está levando em conta os novos campos do `struct lista`. Faça as alterações necessárias para que a função funcione de forma adequada.

Exercício 6: Novamente, pode ser interessante tentarmos remover um nó em específico, ou seja, ter uma função que recebe o ponteiro do nó a ser removido (ao invés do valor). Nesse caso, devemos procurar tal nó na lista e, caso ele exista, removê-lo. Implemente a função `void removeNoLista(Lista *lista, No*p)`.

Exercício 7: Analise o código da função `void insereMisteriosoLista(Lista *lista, int valor)` e tente adivinhar (sem executar o código) o que ela faz.

a) Qual o objetivo da função?

b) Usando apenas essa função para inserir, seria possível implementar uma busca mais eficiente? Como?

c) Dado uma lista que sempre insere como feito por essa função, seria possível implementar uma busca binária? Por quê?

Exercício 8: Para finalizar, implemente a função `Lista * copiaLista(Lista *lista)` que recebe uma lista e retorna uma cópia dessa lista. Essa cópia deve ser independente da lista passada como parâmetro.

Lembre-se de testar vazamentos de memória com o Valgrind. Precisamos/podemos fazer `free` nos nós criados (que não usam `malloc`) na função `main`? Por quê?