Instituto Federal de Educação, Ciência e Tecnologia da Paraíba
Campus Campina Grande
Coordenação de Informática

Laboratório de Estruturas de Dados – Engenharia de Computação
Prof. Ruan Delgado Gomes

Exercício - Listas Estáticas e Encadeadas

1 Operações Especiais – Lista Estática

Com base na implementação de lista estática fornecida em anexo com esta atividade, implemente um conjunto de operações especiais para manipular a lista. As operações a serem implementadas são:

- Verificar se um determinado elemento existe dentro da lista (*hasElement*);
  - Deve-se receber como parâmetro o elemento a ser buscado e deve-se retornar a posição do elemento dentro da lista ou -1 se o elemento não estiver dentro da lista.
- Inserir um elemento em qualquer local da lista (*insertPosition*);
  - Deve-se receber como parâmetro o valor a ser inserido e a posição (inicia em 0) onde o elemento deve ser inserido;
  - Deve-se retornar -1 se a posição for inválida (menor que 0 ou maior que a quantidade de elementos da lista), caso contrário deve-se retornar 0.
  - Se o vetor estiver todo ocupado, deve-se fazer a realocação dobrando o tamanho do vetor.
- Retira um elemento, dado o valor do elemento como parâmetro (*removeElement*);
  - Deve-se receber como parâmetro o valor do elemento a ser retirado e deve-se retornar a posição onde o elemento se encontrava (ou -1 caso o elemento não seja encontrado);
  - Pode-se usar a função *hasElement*, previamente implementada neste exercício, para descobrir onde o elemento a ser removido se encontra e a função *remove_pos*, já implementada no código em anexo, para remover o elemento da lista de acordo com a sua posição.

As assinaturas das funções especiais que manipulam a lista são definidas a seguir:

```
1
2 int hasElement(const lista *l, int v);
3
4 int insertPosition(lista *l, int v, int pos);
5
6 int removeElement (lista *l, int v);
```

Note que nas funções em que não é necessário realizar modificações na lista (funções apenas para consulta de informações sobre a lista ou seus elementos) a lista é passada como um ponteiro constante, para que não seja possível modificar a lista dentro da função. Nas funções em que a lista é alterada (ex: inserção e remoção de elementos), a lista é passada como parâmetro por referência normalmente; ou seja, é possível modificar a lista dentro da função. Após implementar as funções, escreva um programa simples que usa todas as funções escritas, de modo que seja possível testar se todas estão funcionando corretamente.

2 Implementando a Lista Encadeada

Implemente uma lista encadeada de inteiros que possua as operações básicas para manipulação da lista. Para implementar a lista encadeada, coloque a definição da estrutura e as assinaturas das funções em um arquivo chamado listaencadeada.h e a implementação das funções em um arquivo chamado listaencadeada.c, como no exemplo de lista estática enviado em anexo. Para implementar a lista encadeada utilize as seguintes definições de estrutura:

```
1  . . .
2  struct node {
3     int data;
4     struct node *next;
5  };
6  typedef struct node node;
7
8  typedef struct {
9     node *begin;
10 } list;
11 . . .
```

A Figura 2 mostra uma representação gráfica da estrutura definida pelo código mostrado. O tipo *list* contém apenas um ponteiro para o primeiro elemento (do tipo *node*) da lista, denominado *begin*. No entanto, em outras implementações, informações adicionais sobre a lista poderiam ser adicionadas à *struct list*. Como esse exemplo mostra uma lista de inteiros, cada *node* possui um campo do tipo inteiro, e um ponteiro para o próximo elemento da lista (*next*). O último elemento da lista deve ter o valor NULL atribuído ao campo *next*, uma vez que o último elemento não aponta para qualquer outro elemento (obs: note que não se trata de uma lista circular).

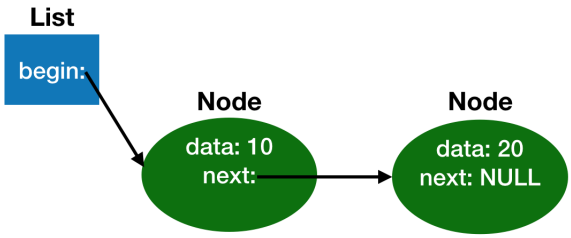

Figura 1: Representação gráfica da lista encadeada.

Na estrutura de dados definida neste documento, por definição, os elementos podem ser inseridos ou removidos de qualquer posição, mas à princípio iremos apenas implementar as funções para adicionar no final da lista ou remover o último elemento da lista.

As operações básicas a serem implementadas são:

- Criação da lista (*createList*);
- Inserção no fim da lista (*add*)
  - Deve-se receber como parâmetro o valor a ser inserido;
- Impressão na tela dos valores armazenados por todos os elementos da lista (*printList*);
- Verificação se a lista está vazia (*isEmpty*);
  - Deve-se retornar 1 caso a lista esteja vazia e 0 caso contrário.

- Remoção no fim da lista (*removeBack*);
- Verificar a quantidade de elementos presentes na lista (*size*).

As assinaturas das funções básicas que manipulam a lista são definidas a seguir:

```
1  . . .
2  void createList(list *l);
3
4  void add(list *l, int v);
5
6  void printList(const list *l);
7
8  int isEmpty(const list *l);
9
10 void removeBack(list *l);
11
12 int size(const list *l);
13 . . .
```

Após implementar as funções, escreva um programa simples que usa todas as funções escritas, de modo que seja possível testar se todas estão funcionando corretamente. Lembre de testar os casos de borda; ou seja, situações em que se realizam modificações no início ou no final da lista, bem como testar situações em que os parâmetros passados são inválidos.

3 Operações Especiais

Uma vez implementada a lista dinâmica, implemente um conjunto de operações especiais para manipular a lista. As operações a serem implementadas são:

- Verificar se um determinado elemento existe dentro da lista (*hasElement*);
  - Deve-se receber como parâmetro o elemento a ser buscado e deve-se retornar a posição do elemento dentro da lista ou -1 se o elemento não estiver dentro da lista.
- Inserir um elemento em qualquer local da lista (*insertPosition*);
  - Deve-se receber como parâmetro o valor a ser inserido e a posição (inicia em 0) onde o elemento deve ser inserido;
  - Deve-se retornar -1 se a posição for inválida (menor que 0 ou maior que a quantidade de elementos da lista – pode-se usar a função *size*), caso contrário deve-se retornar 0.
- Remover um elemento de qualquer local da lista (*removePosition*);
  - Deve-se receber como parâmetro a posição (inicia em 0) do valor a ser removido;
  - Se a posição for inválida nada é feito na lista e deve-se retornar -1, caso contrário deve-se retornar 0.
- Retira um elemento, dado o valor do elemento como parâmetro (*removeElement*);
  - Deve-se receber como parâmetro o valor do elemento a ser retirado e deve-se retornar a posição onde o elemento se encontrava (ou -1 caso o elemento não seja encontrado);
- Obter o valor de um elemento da lista (*get*).
  - Deve-se receber como parâmetro a posição do elemento a ser consultado, bem como o endereço (*vret*) de uma variável do tipo inteiro. A variável referenciada pelo ponteiro *vret* vai armazenar o valor do elemento que foi consultado;

– Deve-se retornar -1 caso a posição consultada seja inválida, caso contrário deve-se retornar 0.

As assinaturas das funções especiais que manipulam a lista são definidas a seguir:

```
1
2  int hasElement(const list *l, int v);
3
4  int insertPosition(list *l, int v, int pos);
5
6  int removePosition(list *l, int pos);
7
8  int removeElement (list *l, int v);
9
10 int get(const list *l, int pos, int *vret);
```

Após implementar as funções, escreva um programa simples que usa todas as funções escritas, de modo que seja possível testar se todas estão funcionando corretamente.

4 Desafio Final

A função para saber a quantidade de elementos presentes na lista (*size*), da forma como implementada neste exercício (percorrendo a lista para contar a quantidade de elementos) possui complexidade $O(N)$. Note que a função *size* também pode ser usada para verificar se uma posição é válida, o que é útil na implementação das funções *insert*, *remove* e *get*. Além disso, usualmente a função para saber a quantidade de elementos dentro de uma lista é chamada com muita frequência por programas que usam uma lista para armazenar dados. Portanto, seria interessante que a função *size* tivesse uma complexidade menor. Pense em uma modificação que pode ser feita na estrutura de dados definida neste exercício, de modo que a função *size* tenha complexidade $O(1)$ (ou seja, sem precisar percorrer toda a lista para saber a quantidade de elementos). Quais modificações seriam necessárias para essa nova implementação? A modificação causa impacto apenas na função *size* ou outras funções também precisariam ser modificadas?