Universidade Federal do Ceará - Campus Quixadá
QXD0010 – Estruturas de Dados – Turma 03A
Prof. Atílio Gomes

Projeto – Avaliação Parcial 2

A solução do problema descrito neste documento deve ser entregue até a meia-noite do dia **22/01/2022**.

Leia atentamente as instruções abaixo.

Instruções:

- Este trabalho pode ser feito em **dupla** ou **individualmente** e deve ser implementado usando a linguagem de programação C++
- O seu trabalho deve ser compactado (.zip, .rar, etc.) e enviado via Moodle no link correspondente ao Projeto da disciplina.
- Identifique o código-fonte do projeto colocando o **nome** e a **matrícula** dos integrantes da equipe como comentário no início do código.
- Indente corretamente o seu código para facilitar o entendimento.
- A estrutura de dados deve ser implementada como TAD.
- Os códigos-fonte devem estar devidamente organizados e documentados.
- Observação: Lembre-se de desalocar os endereços de memória alocados quando os mesmos não forem mais ser usados.
- **Obsservação: Qualquer indício de plágio resultará em nota ZERO para todos os envolvidos.**

DICA: COMECE O TRABALHO O QUANTO ANTES.

1 Listas Circulares Duplamente Encadeadas

A estrutura de lista simplesmente encadeada, vista durante a aula, caracteriza-se por formar um encadeamento simples entre os nós: cada nó armazena um ponteiro para o próximo elemento da lista. Dessa forma, não temos como percorrer eficientemente os elementos em ordem inversa. O encadeamento simples também dificulta a retirada de um elemento da lista. Mesmo se tivermos o ponteiro do elemento que desejamos retirar, temos de percorrer a lista, elemento por elemento, para encontrar o elemento anterior, pois, dado o ponteiro para um determinado elemento, não temos como acessar diretamente seu elemento anterior.

Para solucionar esses problemas, podemos formar o que chamamos de **listas duplamente encadeadas**. Nelas, cada elemento tem um ponteiro para o próximo elemento e um ponteiro para o elemento anterior. Assim, dado um elemento, podemos acessar os dois elementos adjacentes: o próximo e o anterior. A lista duplamente encadeada pode ou não ter um nó cabeça e pode ou não ser circular, conforme as conveniências do programador. Uma **lista circular duplamente encadeada** é uma lista duplamente encadeada na qual o último elemento da lista passa a ter como próximo o primeiro elemento, que, por sua vez, passa a ter o último como anterior. A Figura 1 ilustra uma lista duplamente encadeada com estrutura circular e a presença de um nó sentinela. Já a Figura 2 ilustra a lista quando ela está vazia.

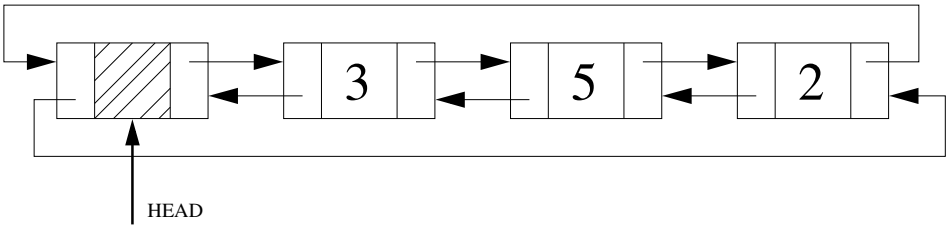

Figura 1: Lista circular duplamente encadeada com nó sentinela.

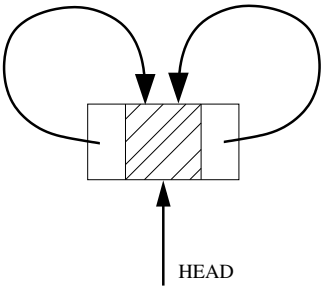

Figura 2: Uma lista circular duplamente encadeada com nó sentinela vazia.

Problema: Implemente em C++ o Tipo Abstrato de Dados Lista Linear usando como base a estrutura de dados Lista Circular Duplamente Encadeada Com Nó Sentinela. A sua estrutura de dados deve ser encapsulada por uma classe chamada `List`, que deve suportar as seguintes operações:

(1) `List();`
Construtor vazio da classe `List`. Cria uma lista vazia. Deve iniciar todos os atributos da classe com valores válidos.

(2) `List(const List& lst);`
Construtor de cópia da classe `List`. Esse construtor recebe como parâmetro uma referência para um objeto `lst` do tipo `List` e inicializa a sua lista com os mesmos elementos da lista `lst`.

(3) `~List();`
Destrutor da classe. Libera toda a memória que for alocada dinamicamente pela estrutura.

(4) `bool empty() const;`
Retorna se a lista está vazia.

(5) `size_t size() const;`
Retorna o número de nós da lista. O tipo de retorno size_t já encontra-se definido por padrão no C++ e é a mesma coisa que um unsigned int.

(6) `void clear();`
Esvazia a lista.

(7) `Item& front();`
Retorna uma referência para o primeiro elemento na lista.

(8) `Item& back();`
Retorna uma referência para o último elemento na lista. Se a lista estiver vazia, essa função deve lançar uma exceção (exception).

(9) `void push_front(const Item& data);`
Adiciona um `Item` no início da lista.

(10) `void push_back(const Item& data);`
Adiciona um `Item` ao final da lista.

(11) `void pop_front();`
Remove o elemento no início da lista. Se a lista estiver vazia, essa função deve lançar uma exceção (exception).

(12) `void pop_back();`
Remove o elemento no final da lista. Se a lista estiver vazia, essa função deve lançar uma exceção (exception).

(13) `void insertAt(const Item& data, int index);`
Insere um novo `Item` na posição `index` da lista. Dada uma lista com n elementos, esta função só deve inserir o novo elemento se e somente se $0 \leq index \leq n$. Caso contrário, uma exceção deve ser lançada. Esta função deve ter complexidade $O(n)$ no pior caso.

(14) `Item removeAt(int index);`
Remove o elemento na posição `index` da lista e retorna o seu valor. Dada uma lista com n elementos, esta função só deve remover o elemento se e somente se $0 \leq index \leq n-1$. Caso contrário, uma exceção deve ser lançada. Esta função deve ter complexidade $O(n)$ no pior caso.

(15) `void removeAll(const Item& data);`
Remove da lista todas as ocorrências do elemento `data` passado como parâmetro. Esta função deve ter complexidade $O(n)$ no pior caso.

(16) `void swap(List& lst);`
Troca o conteúdo da lista com o conteúdo da lista `lst`. Após a chamada a esta função-membro, os elementos nesta lista são aqueles que estavam em `lst` antes da chamada e os elementos de `lst` são aqueles que estavam nesta lista.

(17) `void concat(List& lst);`
Concatena a lista atual com a lista `lst` passada por parâmetro. Após essa operação ser executada, `lst` será uma lista vazia, ou seja, o único nó de `lst` será o nó sentinela. Esta função deve ter complexidade $O(n)$ no pior caso.

(18) `List *copy();`
Retorna um ponteiro para uma cópia desta lista. A cópia desta lista deve ser uma outra lista, que deve ser criada dinamicamente dentro da função. Esta função deve ter complexidade $O(n)$ no pior caso.

(19) `void append(Item vec[], int n);`
Esta função recebe um array de `Item` e o seu tamanho n como entrada e copia os elementos do array para a lista. Todos os elementos anteriores da lista são mantidos e os elementos do array devem ser adicionados após os elementos originais. Esta função deve ter complexidade $O(n)$ no pior caso.

(20) `bool equals(const List& lst) const;`
Determina se a lista `lst` passada por parâmetro é igual à lista em questão. Duas listas são iguais se elas possuem o mesmo tamanho e o valor do k-ésimo elemento da primeira lista é igual ao k-ésimo elemento da segunda lista. Esta função deve ter complexidade $O(n)$. Esta função deve ter complexidade $O(n)$ no pior caso.

(21) `void reverse();`
Inverte a ordem dos elementos na lista. Esta função deve ter complexidade $O(1)$ no pior caso.

(22) `void merge(List& lst);`
Recebe uma lista `lst` como parâmetro e constrói uma nova lista que será a intercalação dos elementos da lista original com os elementos da lista passada por parâmetro. Ao final desta operação, `lst` deve ficar vazia. Como um exemplo, imagine duas listas de inteiros L1 = [1 2 3 4] e L2 = [7 8 9 0 5 6]. Então, o resultado da operação `L1.merge(L2)` nos dá as listas L1 = [1 7 2 8 3 9 4 0 5 6] e L2 = [].

(23) `std::ostream& operator<<(std::ostream& out, const List& lst);`
Esta função sobrecarrega o operador de inserção `<<`. Esta função deve ser implementada como uma função global *friend* da classe `List`. Esta função recebe como entrada um fluxo de saída de dados `out` e uma lista `lst` e insere no fluxo `out` os elementos da lista a fim de que eles sejam impressos no terminal.

(24) `Item& operator[](int index);`
Esta função sobrecarrega o operador de indexação [ ]. Ela retorna uma referência

para o elemento no índice `index`. Se esse índice não for válido, uma excessão deve ser lançada por esta função.

(25) `List& operator=(const List& lst);`
Esta função sobrecarrega o operador de atribuição. Esta função adiciona à lista os mesmos elementos que estão na lista `lst`. Ao fazer isso, ela apaga todos os elementos da lista original para que ela possa receber os novos elementos. Por exemplo, considere duas listas P = [2 3 4] e Q = [6 7 8 9]. Após a operação:

$$P = Q;$$

temos que as listas serão P = [6 7 8 9] e Q = [6 7 8 9], onde P e Q são duas listas distintas.

2 Arquivo main.cpp

Escreva um programa principal (`main.cpp`) com um **menu de comandos** para que o usuário possa testar TODAS as operações da estrutura `List` que você implementou. Você e sua equipe devem pensar e projetar os comandos. **Um requisito obrigatório do menu de comandos é que ele deve fornecer um comando que possibilite criar várias listas no seu programa. Além disso, deve ser possível referenciar essas listas de modo que possamos testar as funções nelas.** A fim de construir esse arquivo main.cpp, você pode tomar como base o arquivo main.cpp que eu disponibilizei nos exercícios de listas simplesmente encadeadas.

3 Relatório

- **Deverá ser submetido:**
  - Um **relatório do trabalho** realizado, contendo: (1) uma descrição completa da estrutura de dados que foi programada; (2) uma descrição dos algoritmos das funções mais complicadas; e as decisões tomadas relativas aos casos e detalhes de especificação que porventura estejam omissos no enunciado; (3) uma seção descrevendo como o trabalho foi dividido entre a dupla, se for o caso; (4) dificuldades encontradas; (5) Referências bibliográficas, tutoriais, vídeos ou outros materiais consultados.
  - O código-fonte devidamente organizado e documentado.

- Um dos parâmetros utilizados na avaliação da qualidade de uma implementação consiste na constatação da presença ou ausência de comentários. Comente o seu código. Mas também não comente por comentar, forneça bons comentários.

- Outro parâmetro de avaliação de código é a *portabilidade*. Dentre as diversas preocupações da portabilidade, existe a tentativa de codificar programas que sejam compiláveis em qualquer sistema operacional. Como testarei o seu código em uma máquina que roda Linux, não use bibliotecas que só existem para o sistema Windows como, por exemplo, a biblioteca `conio.h` e outras tantas.

Informações adicionais para este trabalho:

- Este projeto vale de 0 a 10 pontos.