Universidade Federal do Ceará - Campus Quixadá
QXD0010 – Estruturas de Dados – Turma 02A
Prof. Atílio Gomes

Primeiro Trabalho

As soluções das questões descritas neste documento devem ser entregues até a meia-noite do dia **01/10/2019** pelo SIPPA.

Leia atentamente as instruções abaixo.

Instruções:

- Este trabalho é **individual** e deve ser implementado usando a linguagem de programação C++
- Coloque a solução de cada questão em uma pasta específica. O seu trabalho deve ser compactado (.zip, .rar, etc.) e enviado para o SIPPA na atividade correspondente ao Trabalho 3 da disciplina. (A numeração é esta mesmo, pois as duas atividades que passei no início da disciplina foram cadastradas como Trabalho 1 e Trabalho 2. O SIPPA não ainda não possui a opção Cadastrar Atividade.)
- Identifique o seu código-fonte colocando o seu **nome** e **matrícula** como comentário no início de seu código.
- Indente corretamente o seu código para facilitar o entendimento.
- As estruturas de dados devem ser implementadas como TAD.
- Os programas-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.

Questão 1: [Lista sequencial de tamanho redimensionável] Uma *lista linear* L é um conjunto de $n \geq 0$ *nós* $L_0, L_1, \ldots, L_{n-1}$ tais que suas propriedades estruturais decorrem, unicamente, da posição relativa dos nós dentro da sequência linear:

- Se $n > 0$, L_0 é o primeiro nó,
- Para $0 < k \leq n - 1$, o nó L_k é precedido por L_{k-1}.

As listas lineares estão entre os tipos abstratos de dados de manipulação mais simples. Como vimos em sala, o tipo de armazenamento de uma lista linear pode ser classificado de acordo com a posição relativa na memória de dois nós consecutivos na lista. O caso em que cada dois nós da lista estão em posições contíguas de memória corresponde à **alocação sequencial** de memória. Já o caso em que não é garantido que cada dois nós consecutivos estejam em posições contíguas de memória corresponde à **alocação encadeada** ou **alocação dinâmica**. A escolha de um ou outro tipo depende essencialmente das operações que serão executadas sobre a lista, do número de listas envolvidas na operação, bem como das características particulares dessas listas.

A maneira mais simples de manter uma lista linear na memória do computador é alocar seus nós em posições contíguas (alocação sequencial). Nesse caso, o endereço real do $(j + 1)$-ésimo nó da lista se encontra c unidades adiante daquele correspondente ao j-ésimo elemento. A constante c é o número de bytes de memória que cada nó ocupa. A correspondência entre o índice do array e o endereço real é feita automaticamente pela linguagem de programação quando da tradução do programa.

Em sala de aula, estudamos uma implementação de lista linear usando alocação sequencial (**lista sequencial**). Na implementação que estudamos, a estrutura de dados (o vetor) é encapsulada na classe `QX_SeqList` por meio da utilização do modificador **private**, enquanto as interfaces dos operadores tornam-se visíveis por meio do modificador **public**. Deste modo, graças ao encapsulamento, o programador pode modificar tanto a estrutura de dados quanto a implementação das operações sem provocar alterações nos programas que utilizam a classe `QX_SeqList`, desde que as interfaces dos operadores sejam preservadas.

A implementação de listas por meio de vetores tem como vantagem a economia de memória, já que não gasta-se bytes com ponteiros. Uma segunda vantagem consiste no tempo constante para acessar um nó da lista, dado que sua posição seja conhecida. Porém, como desvantagem citamos o custo para inserir ou retirar itens da lista, que pode causar um deslocamento de todos os itens, no pior caso. Além disso, em aplicações em que não existe previsão sobre o crescimento da lista, a utilização de vetores pode exigir realocação de memória. Essa é uma operação de alto custo em termos de tempo e memória, pois é preciso alocar uma nova área com mais posições do que a atual e copiar todos os itens para ela. Apesar dessa desvantagem, listas sequenciais apresentam melhor performance no caso em que operações de acesso a um nó são frequentemente executadas.

Problema: Uma característica limitante da estrutura `QX_SeqList` é que a capacidade total da lista é fixa uma vez que ela foi criada. Com o intuito de eliminar essa limitação, reimplemente a estrutura `QX_SeqList` vista em sala de aula para sempre permitir a inserção de novos itens na lista. Para isso é preciso modificar a operação de inserção `push_back()` da seguinte forma: toda vez que a inserção de um novo item esgotar a memória disponível no vetor, uma nova área de memória com capacidade maior deve ser alocada e o conteúdo do vetor anterior deve ser copiado para ela. Após sucessivas operações de retirada `remove()`, `removeAll()` ou `pop_back()`, a razão do número

de itens no vetor pela sua capacidade pode se tornar muito pequena. Nesse caso, uma operação para diminuir a quantidade de memória utilizada pelo vetor também deve ser implementada.

Implemente em C++ o Tipo Abstrato de Dados Lista Linear usando como base a estrutura de dados Lista Sequencial. A sua estrutura de dados deve ser encapsulada pela classe chamada `QX_SeqList`, que deve suportar as seguintes operações e deve ter a complexidade de tempo de pior caso listada ao lado de cada uma delas:

- `QX_SeqList()`: Construtor da classe. Note que a lista não tem capacidade máxima determinada, dado que, agora, ela será redimensionável.
  Complexidade de pior caso desta operação: $O(1)$
- `~QX_SeqList()`: Destrutor: libera memória alocada. $O(n)$
- `int size()`: Devolve o tamanho da lista. $O(1)$
- `bool isEmpty()`: Devolve `true` se lista está vazia, e `false` caso contrário. $O(1)$
- `int search(int x)`: Busca chave x e retorna índice do nó caso x esteja presente. Ou devolve `INT_MIN` caso contrário. A constante `INT_MIN` está definida na biblioteca <climits>. Complexidade de pior caso: $O(n)$
- `int at(int k)`: Devolve o k-ésimo elemento da lista. Ou devolve `INT_MIN` não exista. $O(1)$
- `void push_back(int x)`: Adiciona chave x ao final da lista. Complexidade de pior caso: $O(n)$. Complexidade de caso médio: $O(1)$
- `void clear()`: Deixa a lista vazia. $O(1)$
- `void print()`: Imprime elementos. $O(n)$
- `void printReverse()`: Imprime os elementos da lista na ordem reversa. $O(n)$
- `void remove(int x)`: Remove o primeiro inteiro x da lista. $O(n)$
- `void removeAll(int x)`: Remove todas as ocorrências do inteiro x da lista. $O(n)$
- `int pop_back()`: Remove elemento do final da lista. Complexidade de pior caso: $O(n)$. Complexidade de caso médio: $O(1)$

Escreva um programa principal (`main.cpp`) com um menu de opções para que o usuário possa utilizar e testar TODAS as operações da estrutura `QX_SeqList` que você implementou.

Questão 2: [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 cabeça.

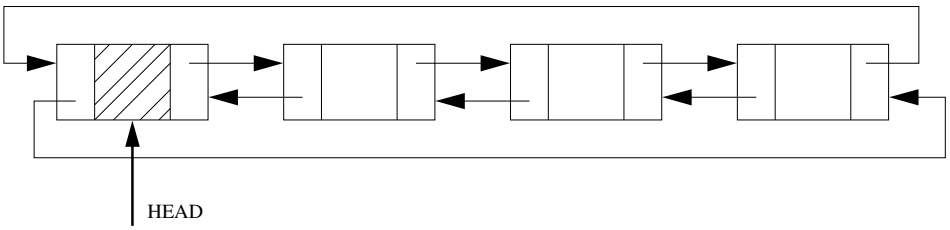

Figura 1: Lista circular duplamente encadeada com nó cabeça.

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

- `QX_List()`: Construtor da classe. Deve iniciar todos os atributos da classe com valores válidos.
- `~QX_List()`: Destrutor da classe. Libera memória previamente alocada.
- `void push_back(int key)`: Insere um inteiro *key* ao final da lista.
- `int pop_back()`: Remove elemento do final da lista e retorna seu valor.
- `void insertAfter(int key, int k)`: Insere um novo nó com valor *key* após o k-ésimo nó da lista.
- `void remove(int key)`: Remove da lista a primeira ocorrência do inteiro *key*
- `void removeAll(int key)`: Remove da lista todas as ocorrências do inteiro *key*.
- `void removeNode(Node *p)`: Remove da lista o nó apontado pelo ponteiro p.

- `int removeNodeAt(int k)`: Remove o k-ésimo nó da lista encadeada e retorna o seu valor. Caso o k-ésimo nó não exista, o programa libera memória alocada e finaliza.
- `void print()`: Imprime os elementos da lista.
- `void printReverse()`: Imprime os elementos da lista em ordem reversa.
- `bool isEmpty()`: Retorna `true` se a lista estiver vazia e `false` caso contrário.
- `int size()`: Retorna o número de nós da lista.
- `void clear()`: Remove todos os elementos da lista e deixa apenas o nó cabeça.
- `void concat(QX_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 cabeça.
- `QX_List *copy()`: Retorna um ponteiro para uma cópia desta lista.
- `void copyArray(int *arr, int n)`: Copia os elementos do array *arr* para a lista. O array *arr* tem n elementos. Todos os elementos anteriores da lista são mantidos e os elementos do array *arr* devem ser adicionados após os elementos originais.
- `bool equal(QX_List *lst)`: Determina se a lista 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.
- `QX_List* separate(int n)`: Recebe como parâmetro um valor inteiro n e divide a lista em duas, de forma à segunda lista começar no primeiro nó logo após a primeira ocorrência de n na lista original. A função deve retornar um ponteiro para a segunda subdivisão da lista original, enquanto a cabeça da lista original deve continuar apontando para o primeiro elemento da primeira lista, caso ele não tenha sido o primeiro a ter valor n.
- `void merge_lists(QX_List *list2)`: Recebe uma `QX_List` como parâmetro e constrói uma nova lista com a intercalação dos nós da lista original com os nós da lista passada por parâmetro. Ao final desta operação, `list2` deve ficar vazia.

Escreva um programa principal (`main.cpp`) com um menu de opções para que o usuário possa utilizar e testar TODAS as operações da estrutura `QX_List` que você implementou.

Informações adicionais para este trabalho:

- 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.
- Este trabalho vale 10 pontos.
- Trabalhos submetidos após o prazo final sofrerão uma penalização de 25% a menos na nota. Por exemplo, se você enviar o trabalho atrasado e ele receber a nota 8, a nota real final será 8-(8*0,25) = 6,0. Só serão aceitos trabalhos com no máximo 5 dias de atraso. Após esse período, nenhum trabalho será mais aceito.