UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS
DEPARTAMENTO DE COMPUTAÇÃO

ESTRUTURAS DE DADOS – TI0140 2022.2 T02A

PROFESSOR: PABLO MAYCKON SILVA FARIAS

3ª PRÁTICA EM LABORATÓRIO (2022-10-21)

Importante:

1. Esta lista se destina à prática individual do conteúdo por cada estudante; as respostas não são para entregar ao professor.
2. Não deixe de tirar as suas dúvidas: procure aproveitar ao máximo a presença do professor no laboratório, mas também não deixe de tirar dúvidas posteriormente.

Questão 1. O objetivo desta questão é fazer uma implementação de Deque vista Lista Duplamente Encadeada, com a seguinte interface, sem alterações:

```
class DequeInt
  {
   ... atributos/campos da classe: ver o item "a" abaixo ...

   public:

   int consultar_dir ()     { ... }
   int consultar_esq ()     { ... }

   DequeInt()               { ... }

   ~DequeInt ()             { ... }

   void inserir_dir (int e) { ... }
   void inserir_esq (int e) { ... }

   void remover_dir ()      { ... }
   void remover_esq ()      { ... }

   bool vazio ()            { ... }
  };
```

Observe que, exceto pelo nome `DequeInt` e pela substituição de `double` por `int`, essa é a mesma interface do exercício da penúltima aula, sobre `DequeVetor`; obviamente, porém, a implementação deve ser bastante diferente.

a) Na verdade, a implementação já foi iniciada na aula passada:

```
class DequeInt
  {
```

```
  struct Noh { int elem;  Noh *ant, *prox; };

  Noh *esq, *dir;

  DequeInt () { esq = dir = nullptr; }

  void inserir_esq (int e) { ... }

  ...
};
```

Partindo então da implementação acima, implemente as funções de inserção:

```
void inserir_esq (int e) { ... }
void inserir_dir (int e) { ... }
```

Observações:

1. Comece atualizando a função `inserir_esq` escrita em sala, identificando trechos comuns ao `if` e ao `else` e que possam ser escritos uma só vez.

2. Observe que o nó alocado pode ser inicializado de maneira bastante sucinta através da sintaxe ilustrada abaixo:

   ```
   #include <iostream>

   using std::cout;

   int main ()
     {
      struct E { int i; double d; char c; };

      E est { 1, 2.3, '4' }, *din = new E { 1, 2.3, '4' };  // <- Aqui

      if (est.i == din->i and est.d == din->d and est.c == din->c)
        { cout << "Tudo certo!\n"; }
      else
        { cout << "Errado!\n"; }

      delete din;
     }
   ```

3. Uma maneira mais idiomática de escrever o construtor teria sido usar uma lista de inicializadores de membros:

   ```
   DequeInt () : esq(nullptr), dir(nullptr) { }
   ```

   A diferença é que esta maneira de fato fornece inicializadores para os atributos da classe. A versão `DequeInt () { esq = dir = nullptr; }` na verdade inicializa os atributos de maneira padrão *e depois* realiza as atribuições em questão de `nullptr`.

4. Após implementar as funções de inserção, mesmo sem ter concluído ainda a implementação do deque, já é possível escrever um pequeno programa que crie um deque e faça algumas inserções. O exercício da aula 17 (realizada na sexta-feira 14/10/2022) dá um exemplo de programa que usa um deque com a interface acima.

b) Implemente as funções de remoção:

```
// Pré-condição: o deque não está vazio.
void remover_dir () { ... }
void remover_esq () { ... }
```

Atenção: assim como na inserção é importante considerar se o deque está vazio no início da chamada da função, no caso da remoção o cenário de deque vazio também é importante; entretanto, conforme escrito acima, é pré-condição da função que o deque não estará vazio quando uma operação de remoção for executada; nesse caso, de que forma o cenário de deque vazio é importante na remoção?

Lembre de desalocar o nó removido. Teste as suas funções.

c) Implemente as funções restantes:

```
int consultar_dir () { ... }  // Pré-condição: deque não vazio.
int consultar_esq () { ... }  // Pré-condição: deque não vazio.

bool vazio ()        { ... }

~DequeInt  ()        { ... }  // Desaloca os nós da lista encadeada.
```

Em seguida, escreva um programa que teste toda a sua implementação de deque, lembrando que nem o construtor nem o destrutor são chamados explicitamente (sendo chamados implicitamente quando da criação e da destruição do objeto do deque).

Questão 2. O objetivo nesta questão é fazer uma implementação do tipo abstrato de dados *conjunto (dinâmico)* usando lista encadeada como estrutura de dados. Um conjunto dinâmico é como o conjunto que conhecemos na matemática, exceto que não é constante, podendo ser modificado por inclusão e remoção de elementos; é portanto um contêiner que se diferencia da pilha, da fila e do deque por ter operações de consulta e remoção que recebem como argumento o próprio elemento a ser consultado ou removido.

a) Você deverá completar a seguinte interface de conjunto:

```
class ConjuntoDouble
  {
   ... atributos ...

   public:

   // Cria um conjunto vazio representado através de lista encadeada.
   ConjuntoDouble ()                  { ... }

   // Retorna "true" se e somente se "e" pertence ao conjunto.
   bool pertence (double e)           { ... }

   // Insere "e" no conjunto sem testar se "e" está no conjunto.
   void inserir_sem_testar (double e) { ... }

   // Insere "e" no conjunto apenas se "e" ainda não pertence ao conjunto.
   void inserir (double e)            { ... }

   // Remove "e" do conjunto, caso "e" pertença a ele;
   // se "e" não pertence ao conjunto, mantém-no inalterado.
   void remover (double e)            { ... }

   // Desaloca a lista encadeada.
   ~ConjuntoDouble ()                 { ... }

  };
```

Observações e orientações:

1. Defina os métodos na ordem acima e vá escrevendo um programa que teste a sua classe à medida em que ela for ficando pronta; esse programa pode ser iniciado assim que você tiver definido o construtor.

2. Observe que há duas funções de inserção: uma que realiza a inserção sem antes testar se o elemento em questão já pertence ao conjunto, e outra que faz esse teste e somente insere quando o elemento ainda não pertence ao conjunto. Naturalmente, a primeira versão é mais rápida, mas, se usada de forma inadequada, pode levar a um conjunto com elementos repetidos e portanto inconsistente.

   A intenção é que o usuário do conjunto use a primeira versão quando puder garantir que o elemento sendo inserido não pertence ao conjunto, e que use a segunda quando não tiver essa garantia.

   Por fim, observe que não precisa haver repetição de código entre essas funções, uma vez

que a segunda pode chamar a primeira (depois do teste de pertinência, que por sua vez também pode ser feito chamando a função `pertence`).

b) (Antes de iniciar este item, complete o item anterior, ou seja, finalize a implementação das funções de conjunto.)

Uma implementação simples da função `remover` divide o código em 3 partes: o caso do conjunto vazio, o caso da remoção do primeiro nó da lista e o caso da remoção de um nó a partir da segunda posição.

Entretanto, é possível escrever um único código para lidar com todos esses casos: basta criar um ponteiro `p` que *aponte para o ponteiro* que aponta para o próximo nó:

1. No início, `p` deve apontar para o ponteiro que aponta para o primeiro nó da lista (observe que isso é diferente de fazer `p` também apontar para esse nó).
2. Em cada momento, o que se deve testar é se o nó `n` apontado pelo ponteiro apontado por `p` possui ou não o elemento procurado:
   - Em caso positivo, a remoção pode ser realizada fazendo o ponteiro apontado por `p` passar a apontar para o nó seguinte a `n`.
   - Em caso negativo, então `p` deve ser reapontado para que o processo seja repetido para o próximo nó (ou seja encerrado, caso não haja outro nó). Para onde `p` deve passar a apontar?

   O código abaixo ilustra o uso de ponteiros (que apontam) para ponteiros:

```
#include <iostream>
using std::cout;

int main ()
  {
   int i = 0, j = 1;  int *p = &i;  // p aponta para i.

   int **pp = &p;                   // pp aponta para p.

   if (*pp == p)
     cout << "Claro: *pp denota o objeto apontado por pp.\n";

   *pp = &j;                        // Agora p aponta para j.

   if (**pp == j)
     cout << "Claro: **pp, ou *(*pp), denota o objeto apontado"
             " pelo objeto apontado por pp.\n";
  }
```

c) Observe que tanto a função `pertence` quanto a função `remover` procuram pelo argumento `e`; existe então uma espécie de duplicação de código, que idealmente deveria ser removida. Porém, essa duplicação não é direta, uma vez que a função `pertence` apenas precisa analisar cada nó individualmente, enquanto que a função `remover` precisa acompanhar também o nó (ou o ponteiro) que aponta para o nó atual, de forma a viabilizar a remoção. Ainda assim, porém,

essa duplicação pode ser removida, usando a técnica de ponteiro (que aponta) para ponteiro do item anterior.

Escreva então uma função-membro (privada)

```
Noh** localizar (double e) { ... }
```

que:

1. Retorne um ponteiro (apontando) para o ponteiro da lista que aponta para o nó que armazena o elemento `e`, caso ele pertença ao conjunto.
2. Caso `e` não pertença ao conjunto, então a função deve retornar um ponteiro (apontando) para o ponteiro da lista que aponta para nulo (este último pode ser o ponteiro da cabeça da lista, se ela estiver vazia, ou então o campo `prox` do último nó, em caso contrário).

Em seguida, reescreva as funções `pertence` e `remover` usando a função `localizar`, removido a duplicação de código que existia entre elas.

Questão 3. Nesta questão você deve criar variações das implementações anteriores usando *nós sentinelas*.

Um nó sentinela é um nó de uma lista encadeada que está sempre presente mas não se destina a armazenar um elemento de fato da lista (a não ser em momentos específicos, para fins de otimização). A presença de um nó sentinela pode ser útil de diferentes maneiras:

1. Para tornar a busca mais rápida: se o nó sentinela ficar no final da lista, então uma operação de busca na lista pode ser agilizada da seguinte maneira: ao invés de fazer um laço e, em cada iteração, testar primeiramente se se atingiu nulo, e, em caso contrário, testar se o nó atual armazena o elemento procurado, pode-se:
   - Começar armazenando na sentinela o elemento procurado.
   - Realizar um laço que simplesmente passa para o próximo nó enquanto o nó atual não possuir o elemento procurado. Observe que não é necessário testar se se atingiu um ponteiro nulo, pois, como o elemento a ser procurado foi inicialmente armazenado na sentinela, então algum nó possuindo esse elemento será necessariamente atingido. Assim, ao invés de 2 testes por iteração, faz-se apenas um teste.
   - Ao fim do laço, basta testar se o nó encontrado é ou não a sentinela: em caso negativo, então é porque o elemento procurado já estava presente na lista e foi encontrado no num nó anterior à sentinela; em caso positivo, então é porque o elemento não estava presente na lista e foi encontrado apenas na sentinela.
2. Para simplificar o código: se, numa lista duplamente encadeada, há sentinela(s) no início e no final da lista, então todos os nós da lista terão anterior e próximo, incluindo o primeiro e o último nós. Isso simplifica a implementação da função de remoção, que não precisa analisar se os ponteiros para anterior e para próximo são nulos ou não.

Assim sendo:

a) Escreva uma variação da implementação de conjunto, utilizando um nó sentinela da seguinte maneira:

1. A sentinela deve ficar no início da lista: isso facilitará a implementação da função de remoção, que não precisará mais considerar o caso especial de remoção no início da lista (embora, como vimos na questão anterior, isso já pudesse ser contornado pelo uso de ponteiros para ponteiros).
2. A lista deve ser circular, ou seja, o último nó deve apontar para o primeiro, que no caso será a sentinela. Isso permitirá a otimização mencionada acima para a operação de busca.
3. Você pode alocar a sentinela dinamicamente no construtor e desalocá-lo no destrutor, mas é tanto mais simples quanto mais eficiente se a sentinela for um nó "fixo", parte da classe do conjunto. Nesse caso, o atributo da classe não seria um ponteiro para nó, mas um nó propriamente dito.
4. Atenção: a função `vazio` do conjunto deve ser adaptada de forma a levar em consideração a presença da sentinela.

b) Escreva uma variação da implementação de deque, utilizando um nó sentinela da seguinte maneira:

1. A classe do deque não deve mais possuir os atributos `esq` e `dir`, mas apenas um nó sentinela (que, assim como no item anterior, não deve ser um ponteiro, mas um nó propriamente dito). O campo `prox` da sentinela servirá para apontar para a extremidade direita do deque, e o campo `ant` apontará para a extremidade esquerda. Novamente, então, teremos uma lista circular (desta vez, uma lista duplamente encadeada, circular e com sentinela).
2. A sentinela servirá para evitar os testes de lista vazia, simplificando tanto as operações de inserção quanto as de remoção.
3. Assim como no item anterior, será necessário adaptar a função `vazio`.

Boa prática!