UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS
DEPARTAMENTO DE COMPUTAÇÃO

ESTRUTURAS DE DADOS – TI0140 2022.2 T02A

PROFESSOR: PABLO MAYCKON SILVA FARIAS

LISTA PARA A 4ª PRÁTICA (2022-12-09)

PARTE 1: IMPLEMENTAÇÃO DE MONTE BINÁRIO

Questão 1. Nesta questão você iniciará uma implementação de monte ("heap") binário de máximo para `doubles`. Mais especificamente, você deve definir os atributos da classe e algumas funções-membro triviais, conforme segue:

```
class Monte
  {
   ... defina aqui os dados-membro (atributos) ...

   public:

   // Pré-condição: tam_max >= 1.

   Monte (int tam_max) { ... deve alocar o vetor e inicializar atributos ... }

   ~Monte () { ... deve desalocar o vetor utilizado ... }

   int capacidade () { ... deve retornar tam_max ... }

   int num_elementos () { ... deve retornar o atual número de elementos ... }
  };
```

Observe que o construtor possui um parâmetro `tam_max`: como, na prática, vários usos de filas de prioridades são tais que, de início, é conhecido um limite máximo para o tamanho da lista, então nós abordaremos uma implementação de fila de prioridades limitada, sem redimensionamento; nesse contexto, o parâmetro `tam_max` informa o número máximo de elementos que o monte binário será capaz de armazenar.

A sua implementação já deve viabilizar, por exemplo, o programa abaixo, que deve retornar zero:

```
int main ()
  {
   Monte M(10);  if (M.capacidade() != 10 or M.num_elementos() != 0) return 1;
  }
```

Questão 2. Continue a implementação, escrevendo agora a função-membro para inserção:

```
void inserir (double elem)
  {
   ... Deve inserir "elem" no monte. ...

   ... Se o monte já estiver cheio, execute (incluindo <stdexcept>):

         throw runtime_error ("Tentativa de inserção com monte cheio!");
  }
```

A implementação desse método deve viabilizar, por exemplo, a inclusão do trecho abaixo na função `main` iniciada acima:

```
for (int i = 0; i < 10; ++i) M.inserir (i);
```

Questão 3. Escreva agora a função-membro para consulta:

```
double consultar ()
  {
   ... Deve retornar o valor do elemento do topo do monte ...

   ... Em caso de monte vazio, execute:

         throw runtime_error ("Tentativa de consulta com monte vazio!");
  }
```

A implementação desse método deve viabilizar, por exemplo, a inclusão do trecho abaixo na função `main` anterior, que deve imprimir `Máximo: 9`:

```
cout << "Máximo: " << M.consultar() << '\n';
```

Questão 4. Escreva agora a função-membro para remoção:

```
void remover ()
  {
   ... Deve remover o elemento do topo do monte ...

   ... Em caso de monte vazio, execute:

         throw runtime_error ("Tentativa de remoção com monte vazio!");
  }
```

A implementação desse método deve viabilizar, por exemplo, a inclusão do trecho abaixo na função `main` anterior:

```
while (M.num_elementos() != 0)
  {
   cout << "Remoção de " << M.consultar() << '\n';
   M.remover ();
  }
```

que deve imprimir:

```
Remoção de 9
Remoção de 8
Remoção de 7
Remoção de 6
Remoção de 5
Remoção de 4
Remoção de 3
Remoção de 2
Remoção de 1
Remoção de 0
```

Parte 2: Implementação da Ordenação por Monte

Questão 5. Nesta questão você iniciará uma modificação do monte implementado anteriormente, de forma a viabilizar uma implementação da ordenação por monte ("heapsort").

Mais especificamente, para começar, você deve implementar um segundo construtor, que já receba um vetor para transformar em monte:

```
Monte (double *vetor, int tam_vetor)
  {
   ... deve inicializar os atributos já utilizando o vetor recebido,

       e deve inserir de vetor[0] a vetor[tam_vetor - 1] no monte ...
  }
```

Importante: se você simplesmente escrever o construtor acima, então o destrutor escrito originalmente vai tentar desalocar o vetor ao final, possivelmente com resultados desastrosos, pois não é possível antecipar sequer se o vetor recebido pelo novo construtor foi alocado dinamicamente. É necessário, então, distinguir entre (1) o caso em que o vetor é alocado pelo primeiro construtor e (2) o caso em que o vetor já é recebido pronto pelo segundo construtor. **Sugestão:** adicione então um novo atributo, de tipo `bool`, que sirva para o destrutor saber se precisa ou não desalocar o vetor no final; naturalmente, esse atributo deve ser inicializado pelos dois construtores (e de forma diferente em cada um).

Depois que você fizer as modificações indicadas acima, tanto este programa

```
int main ()
  {
   Monte M(10);

   if (M.capacidade() != 10 or M.num_elementos() != 0) return 1;

   for (int i = 0; i < 10; ++i) M.inserir (i);

   cout << "Máximo: " << M.consultar() << '\n';

   while (M.num_elementos() != 0)
     {
      cout << "Remoção de " << M.consultar() << '\n';
      M.remover ();
     }

  }
```

quanto este

```
int main ()
  {
   double v[10] = { 0, 4, 9, 6, 1, 2, 5, 8, 7, 3 };  Monte M(v,10);

   if (M.capacidade() != 10 or M.num_elementos() != 10) return 1;

   cout << "Máximo: " << M.consultar() << '\n';
```

```
  while (M.num_elementos() != 0)
    {
     cout << "Remoção de " << M.consultar() << '\n';
     M.remover ();
    }
}
```

devem executar de forma bem-sucedida e retornar zero.

Questão 6. Escreva agora uma função que esvazie o monte, mas que, no decorrer das remoções, vá deixando o vetor subjacente ordenado:

```
void esvaziar_ordenando ()
  {
   ... deve esvaziar o monte, mas deixando o vetor ordenado ...
  }
```

Com a implementação desse método, o programa abaixo

```
int main ()
  {
   double v[10] = { 0, 4, 9, 6, 1, 2, 5, 8, 7, 3 };  Monte M(v,10);

   M.esvaziar_ordenando();

   if (M.capacidade() != 10 or M.num_elementos() != 0) return 1;

   for (int i = 0; i < 10; ++i) cout << ' ' << v[i];

   cout << '\n';
  }
```

deve imprimir

```
 0 1 2 3 4 5 6 7 8 9
```

Boa prática!