UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO
INE5408 - ESTRUTURAS DE DADOS

LEONARDO DE SOUSA MARQUES

PROJETO I:
Verificação de cenários e determinação de área limpa por um robô aspirador

Docente:

Dr. Alexandre Gonçalves Silva

Florianópolis
2024

Sumário

Lista de tabelas

1 INTRODUÇÃO

O seguinte relatório tem como objetivo analisar e resolver dois problemas relacionados a um arquivo XML (*Extensible Markup Language*). No primeiro, vamos verificar se o aninhamento das chaves (< e >) e fechamento das marcações (tags) está correto. Já no segundo, vamos calcular a área total que um robô pode limpar partindo de uma posição (x, y) em cada uma das matrizes fornecidas nos arquivos.

1.1 Modelo de Arquivo

Antes de começar, vamos entender o modelo do arquivo XML, conforme o bloco abaixo.

```
<cenarios>
<cenario>
<nome>cenario-01</nome>
<dimensoes><altura>20</altura><largura>30</largura></dimensoes>
<robo><x>10</x><y>20</y></robo>
<matriz>
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000111000000000000000000
00000000001100000000000000000000
00110010011111001111000111100
00110010001100001100000111000
00110010001100001111100110000
00110010001100000111101100000
00110110001100001001110111000
00011110001100001111100011100
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
</matriz>
</cenario>
... % Continue com os outros cenarios da mesma forma
</cenarios>
```

Com isso, conseguimos extrair as informações de que o robô começará na posição P, tal que P(x, y) = (10, 20), além de que altura = 20 e largura = 30, por exemplo. Essas informações serão úteis para a questão 2. Em relação aos aninhamentos, percebemos que para o trecho mostrado, ele se demonstra correto. Tendo visto o modelo dos arquivos que estamos trabalhando, podemos dar início a resolução dos problemas apresentados.

2 PROBLEMA 1

Para verificar os aninhamentos de uma string que representa todo o arquivo XML lido, é necessário realizar essa conversão. Por isso, o docente forneceu o seguinte bloco de código, na função main, que lê o conteúdo do arquivo com nome passado por linha de comando no terminal - ex: `cenarios1.xml`.

```
1  int main() {
2
3      string filename;
4      std::cin >> filename;
5
6      // Abertura do arquivo
7      ifstream filexml(filename);
8      if (!filexml.is_open()) {
9          cerr << "Erro ao abrir o arquivo " << filename << endl;
10         throw runtime_error("Erro no arquivo XML");
11     }
12
13     // Leitura do XML completo para 'texto'
14     string texto;
15     char character;
16     while (filexml.get(character)) {
17         texto += character;
18     }
19
20     ...
```

Tendo a string texto, podemos criar a função `bool verifica_aninhamentos(string texto)`, que retorna um valor booleano (verdadeiro ou falso). Para implementar essa função, vamos utilizar a estrutura de dados linear de **Pilha** (*Array Stack*), que segue a lógica LIFO (*Last in, first out*), que vai armazenar as tags que estão sendo abertas e fechadas. (O código completo da pilha está descrito no Anexo 1.)

Seguindo essa lógica, vamos precisar do comprimento da string para poder instanciar uma pilha com esse tamanho - note que poderíamos utilizar um valor menor, visto que o arquivo não é composto apenas por tags, mas, dessa forma, anunciamos o tamanho máximo. Assim, a lógica para resolver o problema consistiu em percorrer a string texto até o momento em que

um caracter '<' é encontrado e, a partir disso, escrever a tag que ele abre até o fechamento da mesma no caractere '>'. O motivo de escrevermos a tag inteira é para caso o caractere seguinte a '<' for uma barra de fechamento '/', pois, dessa forma, garantimos que todas estão sendo devidamente validadas.

```
1  bool verifica_aninhamentos(string texto) {
2      // Calcula o comprimento da string de entrada
3      int len = texto.length();
4      // Cria uma pilha para armazenar as tags de abertura
5      structures::ArrayStack<string> stack_tags(len);
6      // Inicializa uma string vazia para armazenar o nome da tag
           atual
7      string tag = "";
8      // Percorre cada caractere da string de entrada
9      for (int i = 0; i < len; i++) {
10         // Se o caractere atual for um '<', significa que uma tag
               vai ser escrita
11         if (texto[i] == '<') {
12             // Verifica se e uma tag de fechamento se o proximo
                   caractere for '/'
13             bool is_closing_tag = (i + 1 < len &&
14                                    texto[i + 1] == '/');
15             // Se for uma tag de fechamento, avanca o indice para
                   pular o '/'
16             if (is_closing_tag) {
17                 i++;
18             }
19             // Percorre a string ate encontrar o '>' para capturar
                   o nome da tag
20             for (int j = i + 1; j < len; j++) {
21                 // Se o caractere atual for '>', a tag foi
                       completamente capturada
22                 if (texto[j] == '>') {
23                     // Atualiza 'i' para a posicao de '>',
                           indicando o fim da tag
24                     i = j;
25                     // Sai do loop interno
26                     break;
27                 }
28                 // Adiciona o caractere atual ao nome da tag
29                 tag += texto[j];
30             }
31
32             // Se for uma tag de fechamento, verifica se a tag de
```

```
                abertura esta no topo da pilha
33              if (is_closing_tag) {
34                  // Se a pilha estiver vazia ou a tag de abertura no
                        topo da pilha nao corresponder a tag de
                       fechamento, retorna falso
35                  if (stack_tags.empty() || stack_tags.pop() != tag)
                       {
36                      return false;
37                  }
38              } else {
39                  // Se for uma tag de abertura, empilha o nome da
                       tag
40                  stack_tags.push(tag);
41              }
42          }
43          // Inicializa a variavel tag como vazia para a proxima
44          tag = "";
45      }
46      // Se a pilha estiver vazia, significa que todas as tags foram
           fechadas
47      return stack_tags.empty();
48  }
```

2.1 Exemplo de Funcionamento

Vamos observar o exemplo da verificação de aninhamentos, com as tags sendo empilhadas e desempilhadas à medida que são analisadas. Considere o código XML:

```
1  <cenarios>
2    <cenario>
3      <nome>cenario-01</nome>
4    </cenario>
5  </cenarios>
```

Com isso, podemos esquematizar a tabela que representa as etapas de empilhamento para verificar os passos dos algoritmo.

Etapa	Pilha
1. Início	∅
2. Tag <cenarios> encontrada	<cenarios>
3. Tag <cenario> encontrada	<cenario> <cenarios>
4. Tag <nome> encontrada	<nome> <cenario> <cenarios>
5. Tag </nome> encontrada e removida	<cenario> <cenarios>
6. Tag </cenario> encontrada e removida	<cenarios>
7. Tag </cenarios> encontrada e removida	∅

Tabela 1 – Exemplo de verificação de aninhamento

A pilha está vazia no final (∅), o que indica que todas as tags de abertura tiveram uma correspondente tag de fechamento, confirmando que o aninhamento está correto.

Também podemos verificar um caso em que tenhamos erros de aninhamento.

```
1 <cenarios>
2   <cenario>
3     <nome>cenario-01<nome> -- erro;
4   </cenario>
5 </cenarios>
```

Etapa	Estado da Pilha
1. Início	∅
2. Tag <cenarios> encontrada	<cenarios>
3. Tag <cenario> encontrada	<cenario> <cenarios>
4. Tag <nome> encontrada	<nome> <cenario> <cenarios>
5. Tag <nome> encontrada	<nome> <nome> <cenario> <cenarios>
6. Tag </cenario> encontrada	<nome> <nome> <dimensoes> <cenario> <cenarios>

Tabela 2 – Processo de Verificação com Erro no Aninhamento

No passo 6 o método irá retornar $false$, pois <cenario> não está no topo da pilha, visto que houve um erro durante o processo. Nesse caso, como a tag <nome> não foi fechada, ela impede o fechamento das anteriores.

3 PROBLEMA 2

Neste problema, o objetivo é calcular a área total que um robô consegue limpar, começando de uma posição inicial (x, y) dentro de uma matriz binária. Cada matriz no arquivo representa um ambiente, e as células com valor '1' indicam áreas que o robô pode limpar, enquanto as células com '0' representam obstáculos ou áreas já limpas.

O algoritmo de limpeza segue uma lógica de propagação a partir da posição inicial (x, y). A partir dessa posição, o robô pode se mover em quatro direções diferentes, que são descritas pelo vetor de direções $\vec{d} = \{(-1, 0), (1, 0), (0, -1), (0, 1)\}$. Essas direções correspondem, respectivamente, aos **movimentos para cima, para baixo, para a esquerda e para a direita**. A cada movimento, as coordenadas (x, y) do robô são atualizadas somando-se os valores dessas direções ao valor atual da posição, permitindo que ele explore áreas adjacentes.

O robô continuará se movendo para todas as células adjacentes que contêm o valor '1' e que ainda não foram visitadas, até que todas as possíveis áreas acessíveis a partir da posição inicial sejam exploradas. Ao final do processo, a **área** total limpa pelo robô será a soma de todas as células conectadas que contêm '1'.

Para isso, alguns códigos para a extração das informações relevantes dos arquivos XML já foram fornecidos pelo docente. Nesse caso, temos a **classe Cenário**, com atributos públicos size_t **nome**, size_t **altura**, size_t **largura**, size_t **x**, size_t **y**, string **matriz** e size_t **índice final** - este último representa o índice final do cenário lido. Vale ressaltar também que o construtor dessa classe recebe como parâmetros a string texto e o índice inicial de leitura do cenário: **Cenario(string texto, size_t indice_inicial)**. Portanto, para o primeiro caso, vamos inicilizar com indice_inicial = 0;

Nota-se, também, que a matriz está sendo dada, na verdade, em forma de string. Então, para acessarmos um índice correspondente à linha i e coluna j, precisaremos realizar o cálculo $E[i \times \text{largura} + j]$, onde E é a matriz string e $0 \leq i <$ largura e $0 \leq j <$ altura.

Sabendo disso, primeiramente criamos um método para percorrer a string texto e retornar a quantidade de matrizes presentes no cenário. Isso será útil para atualizar a lógica de cenários que necessita de um índice inicial na instanciação, como vimos anteriormente.

```
1 size_t quantidade_matrizes(string texto) {
2     size_t count = 0;
3     size_t pos = 0;
4     while ((pos = texto.find("<matriz>", pos)) != string::npos) {
5         count++;
6         pos += 8; // comprimento da tag <matriz>
7     }
8     return count;
9 }
```

Agora, precisamos criar de fato a função que recebe um cenário e extrai a área a ser limpa pelo robô, chamada `int calcula_area(Cenario& c)`. Nesse mesmo método, iremos utilizar uma estrutura de dados de Fila (*Array Queue*), que segue a lógica FIFO (*First in, first out*),

para armazenar os pares (x, y) acessíveis. (O código completo da fila está descrito no Anexo 2.)

```
size_t calcula_area(Cenario& c) {

    // Copia a matriz, posicao inicial do robo e as dimensoes do
       cenario
    string E = c.matriz;
    size_t x = c.x;
    size_t y = c.y;
    size_t altura = c.altura;
    size_t largura = c.largura;

    // Inicializa a variavel de area como 0 (nao ha area limpa
       inicialmente)
    size_t area = 0;

    // Vetor de direcoes movimentos possiveis (cima, baixo,
       esquerda e direita)
    vector<pair<size_t, size_t>> direcoes = {{-1, 0}, {1, 0},
                                             {0, -1}, {0, 1}};

    // Fila para explorar as posicoes acessiveis
    structures::ArrayQueue<pair<size_t, size_t>> queue(altura*
       largura);

    // Vetor de controle que marca as posicoes ja visitadas na
       matriz
    string R = "";
    for (int i = 0; i < E.length(); i++) {
        R += '0';  // Inicializa todas as posicoes como nao
           visitadas
    }

    // Verifica se a posicao inicial (x, y) e uma celula acessivel
       ('1')
    if (E[x * largura + y] == '1') {
        // Marca a posicao como visitada e adiciona a fila de
           exploracao
        R[x * largura + y] = '1';
        queue.enqueue(make_pair(x, y));
        area++;  // Incrementa a area limpa
    }

```

```
34      // Loop para processar todas as celulas acessiveis
35      while (!queue.empty()) {
36          // Desenfileira a proxima posicao a ser processada
37          pair<size_t, size_t> aux = queue.dequeue();
38
39          // Tenta explorar as 4 direcoes
40          for (int i = 0; i < direcoes.size(); i++) {
41              // Novas posicoes de x e y
42              size_t nx = aux.first + direcoes[i].first;
43              size_t ny = aux.second + direcoes[i].second;
44
45              // Verifica se a nova posicao esta dentro dos limites
                   da matriz
46              if (nx < altura && ny < largura) {
47                  size_t idx = nx * largura + ny;
48                  // Se a nova posicao for acessivel ('1') e ainda
                       nao foi visitada
49                  if (E[idx] == '1' && R[idx] == '0') {
50                      // Enfileira a nova posicao e marca como
                           visitada
51                      queue.enqueue(make_pair(nx, ny));
52                      R[idx] = '1';
53                      area++;  // Incrementa a area limpa
54                  }
55              }
56          }
57      }
58      return area;
59  }
```

3.1 Exemplo de Funcionamento

Vamos exemplificar o funcionamento do algoritmo em uma matriz pequena para o problema de cálculo de área, presente no código XML seguinte.

```
<cenarios>
  <cenario>
    <nome>cenario-01</nome>
    <dimensoes><altura>5</altura><largura>5</largura></dimensoes>
    <robo><x>2</x><y>2</y></robo>
    <matriz>
      11100
      10100
      10111
      10001
      11111
    </matriz>
  </cenario>
</cenarios>
```

Para esse exemplo, temos que a posição inicial do robô será no ponto (2, 2). A tabela abaixo ilustra o processo de cálculo da área que o robô pode limpar, partindo da posição inicial. A coluna à esquerda mostra a etapa, e a coluna à direita mostra o estado da fila e as posições sendo processadas.

Etapa	**Estado da Fila**
1. Início	(2, 2)
2. Posição (2, 2) processada, posições vizinhas enfileiradas:	(1, 2), (2, 3)
3. Posição (1, 2) processada, novas posições enfileiradas:	(2, 3), (0, 2)
4. Posição (2, 3) processada, novas posições enfileiradas:	(0, 2), (2, 4)
5. Posição (0, 2) processada, novas posições enfileiradas:	(2, 4), (0, 1)
6. Posição (2, 4) processada, novas posições enfileiradas:	(0, 1), (3, 4)
7. Posição (0, 1) processada, novas posições enfileiradas:	(3, 4), (0, 0)
8. Posição (3, 4) processada, novas posições enfileiradas:	(0, 0), (4, 4)
9. Posição (0, 0) processada, novas posições enfileiradas:	(4, 4), (1, 0)
10. Posição (4, 4) processada, novas posições enfileiradas:	(1, 0), (4, 3)
11. Posição (1, 0) processada, novas posições enfileiradas:	(4, 3), (2, 0)
12. Posição (4, 3) processada, novas posições enfileiradas:	(2, 0), (4, 2)
13. Posição (2, 0) processada, novas posições enfileiradas:	(4, 2), (3, 0)
14. Posição (4, 2) processada, novas posições enfileiradas:	(3, 0), (4, 1)
15. Posição (3, 0) processada, novas posições enfileiradas:	(4, 1), (4, 0)
16. Posição (4, 1) processada, sem novas posições:	(4, 0)
17. Posição (4, 0) processada, sem novas posições:	$\emptyset$

Tabela 3 – Processo de Cálculo de Área Usando Fila

4 ARQUIVO MAIN

Tendo criado os métodos que resolvem os problemas, precisamos chamá-los na função main(). É importante ressaltar que para o segundo problema, como podemos ter diversas matrizes no mesmo XML, é necessário fazer um loop que instância um *Cenario* com posição inicial referente à posição final do último cenário lido. Desse modo, teremos a função main completa.

```
1  int main() {
2
3      string filename;
4      cin >> filename;
5
6      ifstream filexml(filename);
7      if (!filexml.is_open()) {
8          cerr << "Erro ao abrir o arquivo " << filename << endl;
9          throw runtime_error("Erro no arquivo XML");
10     }
11
12     // Leitura do XML completo para 'texto'
13     string texto;
14     char character;
15     while (filexml.get(character)) {
16         texto += character;
17     }
18
19     // Problema 1
20     bool aninhamentos = verifica_aninhamentos(texto);
21
22     if (!aninhamentos) {
23         cout << "Erro de aninhamentos" << endl;
24     } else {
25         // Problema 2
26          size_t num_matrizes = quantidade_matrizes(texto);
27          size_t i = 0;
28          for (size_t m = 0; m < num_matrizes; m++) {
29              Cenario c(texto, i);
30              size_t area = calcula_area(c);
31              cout << c.nome << " " << area << endl;
32              i = c.indice_final;
33          }
34     }
35     return 0;
36 }
```

5 CONCLUSÃO

Como conclusão, vamos apontar as dificuldades na resolução dos problemas.

Em relação ao primeiro, o principal desafio se deu pelo fato de que não podíamos tratar como um simples problema de aninhamentos de abertura e fechamento de chaves '<' e '>'. Por isso, a solução adotada buscou implementar uma pilha que armazenava as tags completas. Assim, quando a sequência '</' fosse identificada, era necessário verificar se a string formada até o fechamento da chave correspondia à que estava no topo da pilha, garantindo que os pares de abertura e fechamento fossem corretos. Essa abordagem se mostrou eficaz para lidar com a estrutura de aninhamento mais complexa, além de verificar possíveis erros estruturais nos arquivos XML fornecidos.

Já no segundo problema, a estrutura de fila permitiu que o processamento das células visitadas fosse realizado de forma simples. Dessa forma, o uso da fila, em conjunto com a busca em largura, facilitou o processo de expansão da área de limpeza, mantendo o controle das posições já processadas e evitando que o robô saísse dos limites da string matriz.

6 REFERÊNCIAS

JOYANES AGUILAR, Luis. **Programação em C++: algoritmos, estruturas de dados e objetos**. São Paulo: McGraw Hill, 2008. xxxi, 768 p. ISBN 9788586804816.

PANDEY, Himanshu. **Stack and Queue**. University of Lucknow. Disponível em: <link>. Acesso em 12 de outubro de 2024.

ANEXO 1 - ArrayStack.h

```
1  // Copyright [2024] <LEONARDO DE SOUSA MARQUES>
2  #ifndef STRUCTURES_ARRAY_STACK_H
3  #define STRUCTURES_ARRAY_STACK_H
4
5  #include <cstdint>  // std::size_t
6  #include <stdexcept>  // C++ exceptions
7
8  namespace structures {
9
10 template<typename T>
11 //! CLASSE PILHA
12 class ArrayStack {
13  public:
14     //! construtor simples
15     ArrayStack();
16     //! construtor com parametro tamanho
17     explicit ArrayStack(std::size_t max);
18     //! destrutor
19     ~ArrayStack();
20     //! metodo empilha
21     void push(const T& data);
22     //! metodo desempilha
23     T pop();
24     //! metodo retorna o topo
25     T& top();
26     //! metodo limpa pilha
27     void clear();
28     //! metodo retorna tamanho
29     std::size_t size();
30     //! metodo retorna capacidade maxima
31     std::size_t max_size();
32     //! verifica se esta vazia
33     bool empty();
34     //! verifica se esta cheia
35     bool full();
36
37  private:
38     T* contents;
39     int top_;
40     std::size_t max_size_;
41
```

```
42      static const auto DEFAULT_SIZE = 10u;
43  };
44
45  }  // namespace structures
46
47  #endif
48
49
50  template<typename T>
51  structures::ArrayStack<T>::ArrayStack() {
52      max_size_ = DEFAULT_SIZE;
53      contents = new T[max_size_];
54      top_ = -1;
55  }
56
57  template<typename T>
58  structures::ArrayStack<T>::ArrayStack(std::size_t max) {
59      max_size_ = max;
60      contents = new T[max];
61      top_ = -1;
62  }
63
64  template<typename T>
65  structures::ArrayStack<T>::~ArrayStack() {
66      delete [] contents;
67  }
68
69  template<typename T>
70  void structures::ArrayStack<T>::push(const T& data) {
71      if (full()) {
72          throw std::out_of_range("pilha cheia");
73      } else {
74          top_ += 1;
75          contents[top_] = data;
76      }
77  }
78
79  template<typename T>
80  T structures::ArrayStack<T>::pop() {
81      if (empty()) {
82          throw std::out_of_range("pilha vazia");
83      }
84      T beforePop = contents[top_];
```

```
85      top_ -= 1;
86      return beforePop;
87  }
88
89  template<typename T>
90  T& structures::ArrayStack<T>::top() {
91      return contents[top_];
92  }
93
94  template<typename T>
95  void structures::ArrayStack<T>::clear() {
96      top_ = -1;
97  }
98
99  template<typename T>
100 std::size_t structures::ArrayStack<T>::size() {
101     return top_ + 1;
102 }
103
104 template<typename T>
105 std::size_t structures::ArrayStack<T>::max_size() {
106     return max_size_;
107 }
108
109 template<typename T>
110 bool structures::ArrayStack<T>::empty() {
111     return (top_ == -1);
112 }
113
114 template<typename T>
115 bool structures::ArrayStack<T>::full() {
116     int max = static_cast<int> (max_size());
117     return ( max == top_ + 1);
118 }
```

ANEXO 2 - ArrayQueue.h

```
1  // Copyright [2024] <LEONARDO DE SOUSA MARQUES>
2  #include <cstdint>  // std::size_t
3  #include <stdexcept>  // C++ Exceptions
4
5  namespace structures {
6  template<typename T>
7  //! classe ArrayQueue
8  class ArrayQueue {
9   public:
10      //! construtor padrao
11      ArrayQueue();
12      //! construtor com parametro
13      explicit ArrayQueue(std::size_t max);
14      //! destrutor padrao
15      ~ArrayQueue();
16      //! metodo enfileirar
17      void enqueue(const T& data);
18      //! metodo desenfileirar
19      T dequeue();
20      //! metodo retorna o ultimo
21      T& back();
22      //! metodo limpa a fila
23      void clear();
24      //! metodo retorna tamanho atual
25      std::size_t size();
26      //! metodo retorna tamanho maximo
27      std::size_t max_size();
28      //! metodo verifica se vazio
29      bool empty();
30      //! metodo verifica se esta cheio
31      bool full();
32
33   private:
34      T* contents;
35      std::size_t size_;  // tamanho atual da fila
36      std::size_t max_size_;  // tamanho maximo que a fila pode ter
37      int begin_;  // indice do inicio (para fila circular)
38      int end_;  // indice do fim (para fila circular)
39      static const auto DEFAULT_SIZE = 10u;
40 };
41
```

```
42 } // namespace structures
43
44 //! construtor padrao
45 template<typename T>
46 structures::ArrayQueue<T>::ArrayQueue() {
47     max_size_ = DEFAULT_SIZE;
48     contents = new T[max_size_];
49     begin_ = 0;
50     end_ = -1;
51     size_ = 0;
52 }
53
54 //! construtor com parametro
55 template<typename T>
56 structures::ArrayQueue<T>::ArrayQueue(std::size_t max) {
57     max_size_ = max;
58     contents = new T[max_size_];
59     begin_ = 0;
60     end_ = -1;
61     size_ = 0;
62 }
63
64 // Destrutor
65 template<typename T>
66 structures::ArrayQueue<T>::~ArrayQueue() {
67     delete [] contents;
68 }
69
70 // M todo para enfileirar - fila circular
71 template<typename T>
72 void structures::ArrayQueue<T>::enqueue(const T& data) {
73     if (full()) {
74         throw std::out_of_range("fila cheia");
75     }
76     end_ = (end_ + 1) % max_size_;
77     contents[end_] = data;
78     size_++;
79 }
80
81 //! metodo desenfileirar
82 template<typename T>
83 T structures::ArrayQueue<T>::dequeue() {
84     if (empty()) {
```

```
85          throw std::out_of_range("fila vazia");
86      }
87
88      T aux = contents[begin_];
89      begin_ = (begin_ + 1) % max_size_;
90      size_--;
91
92      return aux;
93  }
94
95  //! metodo retorna o ultimo
96  template<typename T>
97  T& structures::ArrayQueue<T>::back() {
98      if (empty()) {
99          throw std::out_of_range("fila vazia");
100     }
101     return contents[end_];
102 }
103
104 //! metodo limpa a fila
105 template<typename T>
106 void structures::ArrayQueue<T>::clear() {
107     size_ = 0;
108     begin_ = 0;
109     end_ = -1;
110 }
111
112 //! metodo retorna tamanho atual
113 template<typename T>
114 std::size_t structures::ArrayQueue<T>::size() {
115     return size_;
116 }
117 //! metodo retorna tamanho maximo
118 template<typename T>
119 std::size_t structures::ArrayQueue<T>::max_size() {
120     return max_size_;
121 }
122
123 //! metodo verifica se vazio
124 template<typename T>
125 bool structures::ArrayQueue<T>::empty() {
126     return (size_ == 0);
127 }
```

```
128
129 //! metodo verifica se esta cheio
130 template<typename T>
131 bool structures::ArrayQueue<T>::full() {
132     return (max_size_ == size_);
133 }
```