Universidade Federal de Santa Catarina Centro Tecnológico - CTC
Departamento de Informática e Estatística - INE

Projeto 1

Estrutura de Dados

Prof.
Alexandre Goncalves Silva

Aluno
Augusto de Hollanda Vieira Guerner (22102192)

Introdução

Documentação

Esta seção é destinada à documentação de cada classe utilizada para solucionar o problema que é abordado na seção *Sobre o projeto*. Apesar de não ser necessário, foi também documentado as classes aprendidas e implementadas nas aulas de estrutura de dados.

A seguir haverá uma seção para cada classe e cada seção terá dois tópicos: métodos e membros. Na primeira seção, será comentado sobre parâmetros, retornos e lógica dos métodos. Na segunda, será abordado o porquê de cada membro.

ArrayList

Classe implementada ao longo das aulas de estrutura de dados. Representa uma lista de tamanho fixo, mas escolhido pelo usuário na hora da instanciação. Ou seja, é como o vetor do C, mas com robustez maior.

Métodos

ArrayList(): Construtor padrão do ArrayList. Cria um ArrayList vazio de tamanho máximo de 10 elementos definido por DEFAULT_MAX.

explicit ArrayList(std::size_t max_size): Construtor do ArrayList. Cria um ArrayList vazio de tamanho máximo definido pelo parâmetro max_size_.

~ArrayList: Destrutor do ArrayList. Deleta a memória alocada no construtor.

clear: Método para limpar o Array. Reseta o tamanho do ArrayList para 0.

push_back: Método de inserção de elemento no final do vetor. Insere um dado do tipo T passado por parâmetro no fim do vetor, se ainda não estiver cheio.

push_front: Método de inserção de elemento no início do vetor. Insere um dado do tipo T passado por parâmetro no início do vetor, se ainda não estiver cheio.

insert: Método de inserção de elemento em qualquer posição do vetor. Insere um dado do tipo T passado por parâmetro no índice determinado pelo parâmetro index, se ainda não estiver cheio.

insert_sorted: Método de inserção de elementos em ordem crescente. Insere um dado do tipo T passado por parâmetro em ordem crescente, se ainda não estiver cheio.

pop: Método de deleção de elementos em qualquer posição. Deleta um dado do vetor no índice definido pelo parâmetro index, se esse índice for válido (estar entre o ínicio e o fim do vetor), e o retorna.

pop_back: Método de deleção de elementos no fim do vetor. Deleta o último dado do vetor, se não estiver vazio, e o retorna.

pop_front: Método de deleção de elementos no começo do vetor. Deleta o primeiro dado do vetor, se não estiver vazio, e o retorna.

remove: Método de remoção de elementos do vetor. Remove um dado determinado pelo parâmetro data, se ele existir.

full: Método de status. Retorna verdadeiro se o vetor atingiu seu limite máximo, isto é, se size_ é igual max_size_.

empty: Método de status. Retorna verdadeiro se o vetor está vazio, ou seja, se size_ é igual a zero.

contains: Método auxiliar. Retorna verdadeiro, se achar um dado determinado pelo parâmetro data. Retorna falso, se não achar.

find: Método auxiliar. Retorna o índice de um dado definido pelo parâmetro data. Se não existir, retorna o tamanho atual do vetor.

size: Método de status. Retorna o valor de size_.

max_size: Método de status. Retorna o valor de max_size_.

T& at(std::size_t index)
T& operator[](std::size_t index)
const T& at(std::size_t index) const

const T& operator[](std::size_t index) const: Métodos de acesso. Acessa um elemento no índice determinado pelo parâmetro index e o retorna.

Membros

contents: Vetor de dados genéricos de tamanho definido na hora de instanciar o objeto.

size_: Tamanho atual do vetor. Inicialmente, quando o vetor está vazio, size_ é zero.

max_size_: Tamanho máximo que o vetor pode atingir ($size\_ \leq max\_size\_$).

DEFAULT_MAX: Membro constante de classe. Todas as classes armazenam o mesmo valor constante de 10. Caso nenhum parâmetro seja passado no construtor, o **max_size_** é definido como sendo **DEFAULT_MAX**.

LinkedStack

Classe implementada ao longo das aulas. Representa uma pilha de tamanho dinâmico, isto é, pode ser tão grande quanto a memória possibilitar e a lógica de entrada e saída de elementos é LIFO “last in, first out”.

Métodos

LinkedStack: Construtor padrão da classe. Inicializa as variaveis mebros com seus valores padrões de modo que a pilha esteja vazia.

~LinkedStack: Destrutor padrão da classe. Aciona o método clear que deleta toda memória alocada em tempo de execução e esvazia a pilha.

clear: Método de reset. Limpa a pilha deletando todos elementos alocados dinamicamente e esvazia a pilha.

push: Método de inserção. Empurra um dado definido pelo parâmetro data para o topo da pilha.

pop: Método de deleção. Se a pilha não estiver vazia, deleta o elemento do topo da pilha e o retorna.

top: Método de status. Retorna o elemento que está no topo da pilha.

empty(): Método de status. Retorna verdadeiro se a pilha estiver vazia e falso se ela não estiver.

***size*:** Método de status. Retorna o size_ (tamanho atual da pilha).

Membros

top_: Ponteiro para o nó do topo da pilha. Cada nó aponta para o próximo nó.

size_: Tamanho atual da pilha. Como a pilha é encadeada, então o valor de size_ pode ser tão grande quanto a memória do computador suportar.

Point

Classe que representa um ponto de 2 dimensões. Todos os métodos e membros são públicos.

Métodos

***Point()*:** Construtor padrão da classe. Não faz nada.

***Point(const Point &other)*:** Construtor de movimentação. Geralmente necessário para a implementação do operator=.

***Point(int x_, int y_)*:** Construtor com parâmetro. O parâmetro x_ define o membro x e o y_, o membro y.

***operator=*:** Implementação da método de atribuição.

Membros

***x*:** Valor da coordenada x do ponto.

***y*:** Valor da coordenada y do ponto.

XmlParser

Métodos

XmlParser: Construtor da classe. Recebe as tags que ele será capaz de analisar.

~XmlParser: Destrutor da classe. Deleta toda memória alocada dinamicamente.

is_valid: Método que verifica se o xml passado por parâmetro é válido. Isto é, o método verifica se toda chave que foi aberta foi fechada corretamente e se a estrutura está coerente.

get_next_closing_tag: Método auxiliar que retorna o índice da próxima tag de fechamento. O método poderia ser privado.

ArrayList<std::string> *get_tags_contents(std::string xml, std::string tag): Método que retorna uma lista de conteúdos do xml de acordo com a tag passada por parâmetro. Deve-se fazer um delete no retorno quando terminar de usá-lo.

ArrayList<std::string>*get_tags_contents(std::string xml, ArrayList<std::string > &tag_hierarchy): Método que retorna uma lista de conteúdos do xml dada uma hierarquia de tags. Deve-se fazer um delete no retorno quando terminar de usá-lo.

Membros

Vale observar que as tags de abertura tem uma correspondência com as tags de fechamento no índice. Assim, com o mesmo índice da lista de um, é possível acessar a contraparte no outro.

opening_tags: Lista das tags de abertura.

closing_tags: Lista das tags de fechamento.

Analyzer

Classe responsável por analisar cada cenário do problema. Só tem um método que analisa o cenário a partir da posição do robô, da dimensão do cenário e da matriz.

Métodos

analyze: Método que analisa o cenário a partir das regras citadas na seção *Sobre o projeto*. O retorno é o número de casas que o robô irá limpar.

Membros

Sem membros.

Solver

Métodos

Solver: Construtor da classe. Aqui é criado o membro parser e definido quais tags ele conseguirá analisar.

~Solver: Destrutor da classe. Deleta tudo alocado dinamicamente no construtor.

convert: Método auxiliar. Converte uma matriz em string para uma matriz de inteiros feitas com vetores.

solve: Método que soluciona o problema todo do projeto de acordo com suas exigências.

Membros

XmlParser *parser: Ponteiro para um analisador de xml.

Sobre o projeto

Antes de falar sobre a solução é interessante comentar sobre o projeto no geral. Assim, a seguir está a apresentação dele retirada do próprio moodle.

Objetivo

Este projeto consiste na utilização de estruturas lineares, vistas até o momento no curso, e aplicação de conceitos de pilha e/ou fila para o processamento de arquivos XML contendo matrizes binárias que representam cenários de ação de um robô aspirador. A implementação deverá resolver dois problemas (listados a

seguir), e os resultados deverão ser formatados em saída padrão de tela de modo que possam ser automaticamente avaliados no VPL.

- Entradas:
    - cenarios.zip
- Saídas esperadas:
    - vpl_evaluate.cases
- Dica:
    - utilize "avaliar" (e não "executar") para que as entradas sejam automaticamente carregadas

Materiais

De modo a exemplificar uma entrada para o seu programa, segue o arquivo XML utilizado no primeiro teste:

- cenarios1.xml, cenarios2.xml, cenarios3.xml, cenarios4.xml, cenarios5.xml, cenarios6.xml
    - Arquivo compactado com todos os arquivos XML: cenarios.zip
- dicas sobre leitura e escrita com arquivos em C++
    - http://www.cplusplus.com/doc/tutorial/files/
- para a criação e concatenação de palavras/caracteres, sugere-se o uso da classe `string`:
    - http://www.cplusplus.com/reference/string/string/

Primeiro problema: validação de arquivo XML

Para esta parte, pede-se exclusivamente a verificação de aninhamento e fechamento das marcações (*tags*) no arquivo XML (qualquer outra fonte de erro pode ser ignorada). Se houver um erro de aninhamento, deve se impresso `erro` na tela. Um identificador constitui uma marcação entre os caracteres `<` e `>`, podendo ser de abertura (por exemplo: `<cenario>`) ou de fechamento com uma `/` antes do identificador (por exemplo: `</cenario>`). Como apresentando em sala de aula, o algoritmo para resolver este problema é baseado em pilha (LIFO):

- Ao encontrar uma marcação de abertura, empilha o identificador
- Ao encontrar uma marcação de fechamento, verifica se o topo da pilha tem o mesmo identificador e desempilha. Aqui duas situações de erro podem ocorrer:

- Ao consultar o topo, o identificador é diferente (ou seja, uma marcação aberta deveria ter sido fechada antes)
- Ao consultar o topo, a pilha encontra-se vazia (ou seja, uma marcação é fechada sem que tenha sido aberta antes)

- Ao finalizar a análise (*parser*) do arquivo, é necessário que a pilha esteja vazia. Caso não esteja, mais uma situação de erro ocorre, ou seja, há marcação sem fechamento

Segundo problema: determinação de área do espaço que o robô deve limpar

Cada XML, contém matrizes binárias, com altura e largura, definidas respectivamente pelas marcações `<altura>` e `<largura>`, e sequência dos pontos, em modo texto, na marcação `<matriz>`. Cada ponto corresponde a uma unidade de área, sendo 0 para não pertencente ou 1 para pertencente ao espaço que deve ser limpo, como passo mínimo do robô em uma de quatro direções possíveis (vizinhança-4), Para cada uma dessas matrizes, pretende-se determinar a área (quantidade de pontos iguais a 1 na região do robô) que deve ser limpa, conforme a posição inicial, linha `<x>` e coluna `<y>`, do robô (primeira linha e primeira coluna são iguais a zero). Para isso, seguem algumas definições importantes:

- A vizinhança-4 de um ponto na linha x e coluna y, ou seja, na coordenada (x,y), é um conjunto de pontos adjacentes nas coordenadas:
  - $(x-1,\ y)$
  - $(x+1,\ y)$
  - $(x,\ y-1)$
  - $(x,\ y+1)$
- Um caminho entre um um ponto p_1 e outro p_n é em um sequência de pontos distintos $<p_1, p_2, \ldots, p_n>$, de modo que p_i é vizinho-4 de p_{i+1}., sendo $i=1,2,\ldots,n-1$
- Um ponto p é conexo a um ponto q se existir um caminho de p a q (no contexto deste trabalho, só há interesse em pontos com valor 1, ou seja, pertencentes ao espaço a ser limpo)

- Um componente conexo é um conjunto maximal (não há outro maior que o contenha) *C* de pontos, no qual quaisquer dois pontos selecionados deste conjunto *C* são conexos

Para a determinação da área a ser limpa, é necessário identificar quantos pontos iguais a 1 estão na região em que o robô se encontra, ou seja, é preciso determinar a área do componente conexo. Conforme apresentado em aula, segue o algoritmo de reconstrução de componente conexo usando uma fila (FIFO):

- Criar uma matriz R de 0 (zeros) com o mesmo tamanho da matriz de entrada E lida
- Inserir (x,y) na fila
    - na coordenada (x,y) da imagem R, atribuir 1
- Enquanto a fila não estiver vazia
    - (x,y) ← remover da fila
    - inserir na fila as coordenadas dos quatro vizinhos que estejam dentro do domínio da matriz (não pode ter coordenada negativa ou superar o número de linhas ou de colunas), com intensidade 1 (em E) e ainda não tenha sido visitado (igual a 0 em R)
        - na coordenada de cada vizinho selecionado, na imagem R, atribuir 1

O conteúdo final da matriz R corresponde ao resultado da reconstrução. A quantidade de 1 (uns) deste único componente conexo é a resposta do segundo problema.

Soluções

Agora já é possível partir para as soluções, visto que os problemas já foram corretamente introduzidos pela seção anterior, a qual foi retirada do próprio moodle. Desse modo, segue dois tópicos sobre a solução do primeiro e do o segundo problema respectivamente.

Primeiro problema

O primeiro problema consiste basicamente na validação de um xml, isto é, verificar se todas as tags foram corretamente fechadas. Para isso, a própria

apresentação do problema mostra uma possível solução usando pilhas. Com efeito, se não for a única possível solução para o problema, é quase certo que é uma das poucas. Sendo assim, a solução deste projeto foi totalmente embasada na proposta pelo enunciado.

Primeiramente, idealizou-se que uma classe analisadora de xml seria muito proveitosa, uma vez que facilmente ela poderia implementar funções de validação e de resgate de informação dentro de tags. Tendo isso em mente, criou-se o XmlParser apresentado em linhas gerais na seção da documentação.

O XmlParser possui dois métodos principais: is_valid e get_tags_contents. O método que se focará nesta seção será o is_valid. O método, como o próprio nome sugere, vê se o xml é válido ou não, retornando, respectivamente, verdadeiro ou falso. Para se fazer essa análise, como mencionado antes, utiliza-se a estrutura de dados pilha. Ela inicialmente se encontra vazia **(imagem 1)**, mas, à medida que percorre o xml **(imagem 2)**, ela vai empilhando tags de abertura **(imagem 3)** e desempilhando quando encontra tags fechamento. Se, na hora de desempilhar, a tag de fechamento não corresponder com a tag de abertura no topo da pilha ou se a pilha estiver vazia, o xml é inválido **(imagem 4)**. Do mesmo modo, se ao final do processo, a pilha não estiver vazia, o xml é também inválido, pois indica que ao menos uma tag foi aberta e nunca fechada **(imagem 5)**. A **imagem 6** ilustra o funcionamento do algoritmo desenvolvido.

Fonte: Acervo do autor.

```
274         // Instanciação da pilha de tags, à principio vazia.
275         LinkedStack<std::string> tags_stack;
```

Imagem 1 - Criando uma pilha vazia.

Fonte: Acervo do autor.

```
277         // O for vai iteirar por todos o xml (xml_len).
278         // Isto é, ele vai de caracter por caracter
279         for (size_t i = 0; i < xml_len; i++) {
```

Imagem 2 - For que percorre todo o xml caracter por caracter.

Fonte: Acervo do autor.

```
281             // Este primeiro for é para verificar se os próximos caracteres
282             // são uma tag de abertura.
283             for (std::size_t j = 0; j < opening_tags->size(); j++)
284                 if (opening_tags->at(j).length() <= xml_len - i)
285                     if (xml.substr(i, opening_tags->at(j).length()) == opening_tags->at(j))
286                         tags_stack.push(opening_tags->at(j));
```

Imagem 3 - For para verificar se há tag de abertura para empilhá-la.

Fonte: Acervo do autor.

```
288         // O segundo for é para verificar se os mesmos próximos caracteres
289         // são uma tag de fechamento.
290         for (std::size_t j = 0; j < closing_tags->size(); j++)
291             if (closing_tags->at(j).length() <= xml_len - i)
292                 if (xml.substr(i, closing_tags->at(j).length()) == closing_tags->at(j)) {
293                     if (tags_stack.empty())
294                         return false;
295                     else if (tags_stack.top() == opening_tags->at(j))
296                         tags_stack.pop();
297                     else
298                         return false;
299                 }
```

Imagem 4 - For que verifica a existência de tag de fechamento e, caso haja, faz as verificações necessárias.

Fonte: Acervo do autor.

```
302         // Se no fim estiver vazia a pilha de tag é por que o xml é válido
303         return tags_stack.empty();
```

Imagem 5 - O retorno vai ser verdadeiro caso a pilha esteja vazia, do contrário será falso.

Fonte: Acervo do autor.

Imagem 6 - Ilustração do algoritmo desenvolvido para validar o xml.

Segundo problema

Assim como o primeiro problema, o próprio enunciado propõe uma possível solução, no entanto, desta vez a solução é feita com fila, apesar de ser possível usar pilha. Assim, para se fazer esta solução, foi feita uma classe chamada Analyzer, a qual, como o próprio nome indica, analisa um cenário do problema, dado a posição do robô, a matriz e a dimensão dela.

Antes de falar um pouco sobre a solução é interessante comentar que a classe Analyzer possui um único método chamado *analyze*. Tal método é

responsável por análise do cenário, retornando como resposta o número de casas que o robô deverá limpar.

Primeiramente, a solução começa instanciando a fila vazia para ir armazenando as casas de análise, uma matriz R do mesmo tamanho que o cenário, mas com todos os valores em zero, e um acumulador para contar as casas que serão limpas **(imagem 7)**. Depois disso, é inserido na fila o robô e, seguidamente, verificado se ele caiu dentro de uma área suja. Caso ele não esteja dentro de uma área suja, a função retorna zero, já que não há nada a ser limpo **(imagem 8)**. Caso ele caia dentro de uma área a ser limpa, então o algoritmo inicia um while que só termina quando a fila de áreas sujas estiver vazia. A cada iteração do while é posto 1 na matriz R na posição que está sendo verificada e, em seguida, é testado as posições de cima, de baixo, da esquerda e da direita para ver se estão sujas. Caso uma delas esteja, então ela é inserida dentro da fila de casas a serem limpas e marcado como um na matriz R **(Imagem 9)**. Depois de a fila ficar vazia e, consequentemente, de o algoritmo ter saído do while, é feita a contagem de casas sujas percorrendo a matriz R, que agora vai estar com as casas a serem limpas com valor de um **(imagem 10)**, a deleção da matriz R e retorno de sum.

Fonte: Acervo do autor.

```
617         // Caminho
618         ArrayQueue<Point> way(dimension.x * dimension.y);
619         // Area que o robô limpará
620         ListList *R = new ListList(dimension.y);
621
622         // O tanto que será limpo (retorno)
623         int sum = 0;
624
625         // Criando a área a ser limpa vazia
626         for (int i = 0; i < dimension.y; i++) {
627             R->push_back(new ListInt(dimension.x));
628             for (int j = 0; j < dimension.x; j++)
629                 R->at(i)->push_back(0);
630         }
```

Imagem 7 - Inicializando o problema.

Fonte: Acervo do autor.

```
632         // Colocando o robô no way
633         way.enqueue(Point(robot.y, robot.x));
634         Point aux = way.back();
635
636         // Verificando se o robô está dentro de uma área suja
637         // Se não tiver retorna 0
638         if (aux.x < dimension.x && aux.x >= 0 &&
639             aux.y < dimension.y && aux.y >= 0)
640             if (!matrix.at(aux.y)->at(aux.x))
641                 return sum;
642
643         R->at(aux.y)->at(aux.x) = 1;
```

Imagem 8 - Colocando o robô na fila e testando se está dentro de uma área a ser limpa.

Fonte: Acervo do autor.

```
644     // Se tiver, começa a analisar a vizinhança até acabar
645     while (!way.empty()) {
646         // Retira a vizinhaça
647         Point p = way.dequeue();
648
649         // Analisa as casas de cima e de baixo da atual.
650         // Se tiver suja e dentro da matriz, insere no way.
651         for (int i = -1; i < 2; i += 2)
652             if (p.y + i < dimension.y && p.y + i > -1 &&
653                 p.x < dimension.x && p.x > -1)
654                 if (matrix.at(p.y + i)->at(p.x) && !R->at(p.y + i)->at(p.x)) {
655                     way.enqueue(Point(p.x, p.y + i));
656                     // Define 1 para não analisar novamente a área
657                     R->at(p.y + i)->at(p.x) = 1;
658                 }
659
660         // Analisa as casas do lado e do outro da atual.
661         // Se tiver suja e dentro da matriz, insere no way.
662         for (int i = -1; i < 2; i += 2)
663             if (p.x + i < dimension.x && p.x + i > -1 &&
664                 p.y < dimension.y && p.y > -1)
665                 if (matrix.at(p.y)->at(p.x + i) && !R->at(p.y)->at(p.x + i)) {
666                     way.enqueue(Point(p.x + i, p.y));
667                     // Define 1 para não analisar novamente a área
668                     R->at(p.y)->at(p.x + i) = 1;
669                 }
670     }
```

Imagem 9 - Ações feitas dentro do while: colocar um na matriz na posição atual e verificar posições adjacentes para colocar dentro da fila.

Fonte: Acervo do autor.

```
502         // Faz a contagem do número de casas que serão limpas
503         for (std::size_t i = 0; i < R->size(); i++)
504             for (std::size_t j = 0; j < R->at(i)->size(); j++)
505                 sum += R->at(i)->at(j);
506
507         // Deletando a matriz R auxiliar
508         for (std::size_t i = 0; i < R->size(); i++)
509             delete R->at(i);
510         delete R;
511
512         return sum;
```

Imagem 10 - Cálculo do sum, deleção da matriz criada dinamicamente e retorno de sum.

Vale destacar que isso é feito para todos os cenários provenientes de um XML válido na classe Solver, precisamente no método solve **(Imagem 11)**. Além disso, como é possível notar na **imagem 12**, primeiro é pego as informações de cada cenário do xml e, em seguida, em cada ciclo do for, é feita a análise dele já mostrando na saída o resultado. É interessante observar que a classe que extraia as informações do xml é justamente o XmlParser com seu método get_tags_contents. Nesse método é passado por parâmetro o xml e a tag que se quer obter o conteúdo. O retorno é uma lista com todos os conteúdos do xml dentro da tag desejada, que foi passada por parâmetro. Ademais, o modo que se faz essa extração de informação é muito similar ao modo como se valida o xml.

Fonte: Acervo do autor.

```
592         // Verificando se o xml é válido
593         if (!parser->is_valid(xml)) {
594             std::cout << "erro" << std::endl << std::endl;
595             return;
596         }
```

Imagem 11 - Verificando se o xml é válido, caso não seja, mostra erro e termina a execução com o return.

Fonte: Acervo do autor.

```
598         // Obtendo as informações de cada cenário
599         ListString *names = parser->get_tags_contents(xml, "nome");
600         ListString *width = parser->get_tags_contents(xml, "largura");
601         ListString *height = parser->get_tags_contents(xml, "altura");
602         ListString *robot_xs = parser->get_tags_contents(xml, "x");
603         ListString *robot_ys = parser->get_tags_contents(xml, "y");
604         ListString *matrices = parser->get_tags_contents(xml, "matriz");
605
606
607         // For que passará por cada cenário analisando
608         Analyzer analyzer;
609         for (std::size_t i = 0; i < names->size(); i++) {
610
611             // Analisando o cenário
612             Point robot = Point(std::stoi(robot_xs->at(i)), std::stoi(robot_ys->at(i)));
613             Point dimension = Point(std::stoi(width->at(i)), std::stoi(height->at(i)));
614             ListList *matrix = convert(matrices->at(i), dimension);
615             std::cout << names->at(i) << " " << analyzer.analyze(robot, dimension, *matrix) << std::endl;
616
617             // Destruindo a metriz criada
618             for (std::size_t i = 0; i < matrix->size(); i++)
619                 delete matrix->at(i);
620             delete matrix;
621         }
622         std::cout << std::endl;
```

Imagem 12 - Pegando as informações de cada cenário e fazendo a análise de cada um.

Conclusão

O projeto foi permeado de dificuldades, desde as menores, como esquecer um ponto e vírgula, até encontrar o ponteiro mal, que aponta para coisa para a qual não deveria. Mas certamente os problemas de maior impacto foram o tempo de execução, a implementação do método *operator*=, a posição do robô e a implementação de um analisador de xml, que faz a validação do xml e a busca de conteúdo nele.

No que se refere ao tempo de execução, o problema foi obtido quando foi feita a reescrita do código usando as classes implementadas em aula e não as classes já oferecidas pelo C++, uma exigência do professor. Quando terminou a refatoração, o código, por estar usando a LinkedList implementada na aula no lugar do vector do C++, começou a demonstrar uma demora inesperada para analisar os xml´s, principalmente o último (cenarios6.xml), que, aliás, até hoje está rodando. O problema foi resolvido reescrevendo novamente o código usando o ArrayList, uma lista em vetor estático. Essa solução foi proposta pelo professor ao perguntar para ele o motivo da demora. Embora não tão rápida quanto a primeira com vector, esta solução ao menos terminou de rodar e o vpl foi capaz de ser avaliado.

Já no que concerne a implementação do operator=, foi um problema levantado por conta de uma frescura de não querer trabalhar com ponteiros ou referência nos parâmetros e nos retornos dos métodos/funções. Foi tentado diversas formas para que o operator= funcionasse para um ArrayList de ArrayList de

inteiros. No entanto, todas as formas geraram programas que terminavam abruptamente. Assim, para solucionar o problema, foi descartada a ideia de implementar esse método para a classe ArrayList. Vale observar que a classe Point tem o método, pois ela não lida com alocação dinâmica.

Antes de falar sobre o problema encontrado ao desenvolver o analisador de xml, é interessante comentar um pouco sobre o problema do robô. Se foi ou não um descuido, não se sabe, mas as coordenadas do robô estão invertidas. Ou seja, o x é o y e o y é o x. Isso foi um erro que demandou tempo, pois, para encontrá-lo, era necessário analisar um xml à mão. Além disso, para alguns cenários, o erro até gerava a resposta certa de modo que dificultou ainda mais o debug.

Por último, a dificuldade em analisar um arquivo .xml. Na verdade, facilmente foi feito no início um analisador que era capaz de tirar 10 no vpl, contudo ele não era o mais geral possível. O problema não acontece em nenhum xml do projeto, pois ele só ocorre quando tem tags iguais uma dentro da outra. Por exemplo, a primeira versão não conseguiria analisar o xml da **imagem 13**, uma vez que tem tag example dentro de tag example (a solução para isso é abordada na seção *Sobre o projeto).*

Fonte: Acervo do autor.

```
<root>
      <example>
            <example>
            </example>
      </example>
</root>
```

Imagem 13 - Exemplo de XML incapaz de ser lido na primeira versão do analisador.

De acordo com o supracitado, o problema do xml não ocorre em nenhum arquivo do projeto. Na verdade, não se sabe se pode acontecer em um xml real. Desse modo, mais uma vez, o problema foi levantado apenas por conta de “frescura” em querer deixar o programa o mais robusto possível. Aqui se fala em frescura assim como em alguns parágrafos acima, mas não no sentido ruim de ser apenas uma futilidade, mas sim apenas uma forma de falar que tem outras formas de se fazer sem a necessidade de deparar com problemas.

Como visto, ao se fazer o projeto, ocorreram diversos problemas e cada um foi solucionado de uma forma. Tudo isso serviu de aprendizado, não só porque foi necessário o estudo cuidadoso do código, mas também porque foi necessário a pesquisa de como funciona a linguagem C++ e as estruturas de dados. Enfim, embora sejam considerados problemas, eles no final das contas são os que nos fazem evoluir e pensar fora da caixa.

Referências

- C++ Reference. cppreference, c2023. Página inicial. Disponível em: <https://en.cppreference.com/w/>. Acesso em: 1 de mai. de 2023.
- Tenenbaum, Aaron Ai. Estrutura de dados usando C. 1º edição. Editora ABDR, 26 junho 1995.