

Universidade Federal de Minas Gerais

Programação Orientada à Objeto

Trabalho Prático 1a – Lista Encadeada



Integrantes: Matheus Ferreira Marques

2011018271

Professores: Raquel Mini

Data: 10/10/2014

1 Introdução

Estrutura de Dados é algo muito utilizado hoje no mundo da computação. Programar é basicamente estruturar dados e construir algoritmos. De acordo com Wirth (1976, p. XII), programas são formulações concretas de algoritmos abstratos, baseados em representações e estruturas específicas de dados.

Para esse Trabalho Prático vamos estudar uma estrutura de dados muito conhecida, a lista. Existem duas estruturas de lista conhecidas: lista encadeada por meio de *arranjos* e lista encadeada por meio de *apontadores*. Construiremos nosso Trabalho Prático baseando-se em listas com *apontadores*.

A diferença entre essas listas, está no tamanho e no acesso. Em listas de arranjo, o acesso é feito por um vetor Lista e seu tamanho é limitado. Em listas de apontadores, o acesso é feito por ponteiros e cada posição dessa lista é nomeado de célula. Em ambos os casos, deve ser explícito o primeiro elemento da lista e o último.

As figuras 1.1 e 1.2 foram tiradas do livro Projeto de Algoritmos do autor Nivio Ziviani e ilustram o que foi explicado acima.

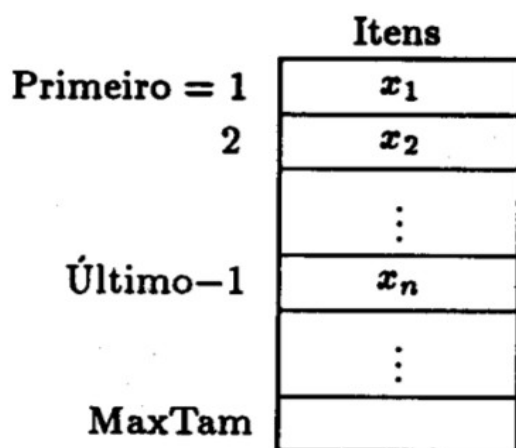


Figura 1.1. Esquema de uma Lista Encadeada por meio de Arranjos (Ziviani, 2011)

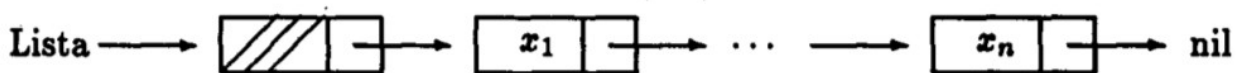


Figura 1.2. Esquema de uma Lista Encadeada por meio de Apontadores (Ziviani, 2011)

Listas podem variar de outras listas com base em sua aplicação. Essas mudanças refletem em quantos métodos (funções) vão ser criadas para ela.

Para esse Trabalho Prático enumeramos alguns métodos que devem ser implementados.

1. Criar uma lista vazia (construtor).
2. Inserir um novo elemento¹ a lista. A lista deve estar sempre em ordem crescente e se o

elemento já existir na lista, ele não é inserido.

3. Verificar se um elemento existe na lista.
4. Retirar um elemento da lista, caso ele exista.
5. Grave os elementos em uma ostream (podendo ser cout ou uma ofstream) recebida como parâmetro.
6. Leia os elementos da lista de uma istream (podendo ser cin ou uma ifstream) recebida como parâmetro.
7. Caso o objeto lista deseja destruído, crie um destrutor que desaloque corretamente.

2 Implementação

Com a ajuda da linguagem C++ fizemos todas essas sete etapas. Foram utilizados três códigos fonte: *main.cpp*, *list.cpp*, *list.h*. Um *Makefile* foi criado para compilação dos códigos.

2.1 Códigos Fonte

List.h

Nesse código declaramos a classe *List*. Ela foi responsável por determinar quais atributos e métodos foram utilizados para solucionar nosso problema. Para que a classe seja útil de se trabalhar e reutilizável, como toda boa prática de orientação à objeto, criei duas classes aninhadas a *List*: classe *Cell* e *Data*. Intuitivamente, *Cell* representa cada célula da nossa lista encadeada e *Data* armazena todos os dados de cada célula. Isso facilita na implementação e na modificação de códigos – caso seja necessário alterar o que a lista armazena, basta modificar a classe *Data* para seus objetivos. Além dessas, foram criadas dois ponteiros do tipo *Cell* para apontar para o início e fim da lista, como toda lista encadeada. Além desses ponteiros, temos, também, em cada célula um ponteiro que aponta para a próxima célula.

Além desses atributos, temos os comportamentos dessa classe *List* que são as sete etapas citadas acima. Cada uma foi implementada no código *List.cpp* com nomes bastantes intuitivos para ajudar na legibilidade do código.

A figura 2.1 mostra o código da classe *List*.

```
class List
{
    /* Data for each cell. On this case, just a integer*/
    typedef class Data
    {
        public:
            int item;
    }Data;
```

```

/* Each cell of list */
typedef class Cell
{
    public:
        Data data;
        class Cell *next;

}Cell;

/* Pointers for first and last cells of list */
Cell *first, *last;

public:
    List();
    ~List();
    bool HaveThisNumber(int, Cell *&);
    bool HaveThisNumber(int);
    bool Insert(int);
    bool Remove(int);
    bool WriteData(ostream &);
    bool ReadData(istream &);
};

```

Figura 2.1. Classe List

List.cpp

Nesse código estão todas as implementações do comportamento de *List*. Ressaltos algumas decisões de implementação:

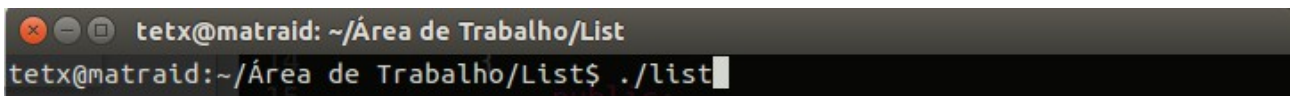
1. *HaveThisNumber* é um método com um parâmetro opcional, *class Cell*. Isso, pois o método *Remove* precisa de pesquisar ao longo da lista para remover um elemento da lista. Caso ele encontre, a lista já está na célula que precisa ser removida, sem ter que refazer a busca dentro de *Remove*, já que *HaveThisNumber* já a realizou.
2. *ReadData* recebe uma *istream* por referência que pode ser tanto um arquivo para leitura ou a entrada padrão *cin*. Se cair na segunda condição, o usuário tem a liberdade de escrever quantos números quiser delimitado por um espaço. Alguns testes em *main.cpp* foram feitos para ilustrar esse caso.
3. *Insert* não insere um elemento já existente, como pedido. Uma mensagem na tela indica que não foi possível inserir.
4. *Remove* cai no mesmo caso de *Insert*, mas para a condição de que o elemento não existe para ser removido.
5. Todos os métodos retornam *bool*. Caso uma implementação futura precise fazer testes de comportamento da classe, isto está disponível.

6. O construtor *List* aloca espaço para a cabeça da lista e aponta para o início dela sendo *NULL* determinando que esta está vazia.
7. O destrutor *~List* é chamado assim que o objeto da classe *List* sai de seu escopo (no nosso caso quando a main termina sua execução). Ele é responsável pela desalocação da lista, caso ainda exista elementos nela.

Main.cpp

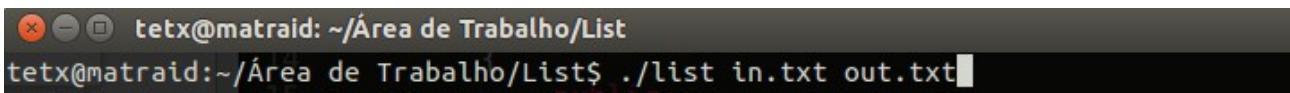
Esse código foi criado para fins de teste. Um destaque desse código é sua facilidade de utilizar arquivos ou a entrada padrão para os testes. Caso queira utilizar arquivos, basta acrescentar na linha de comando o nome do arquivo de entrada e saída para os testes. Caso queira utilizar *cin* e *cout* basta escrever apenas o nome do executável, sem o nome dos arquivos.

As figuras 2.2 e 2.3 ilustram melhor a explicação acima.



```
tetx@matraid: ~/Área de Trabalho/List
tetx@matraid:~/Área de Trabalho/List$ ./list
```

Figura 2.2 Comando para execução do código por meio de cin e cout



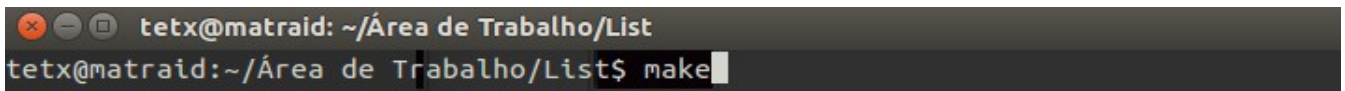
```
tetx@matraid: ~/Área de Trabalho/List
tetx@matraid:~/Área de Trabalho/List$ ./list in.txt out.txt
```

Figura 2.3 Comando para execução do código por meio de arquivos

Makefile

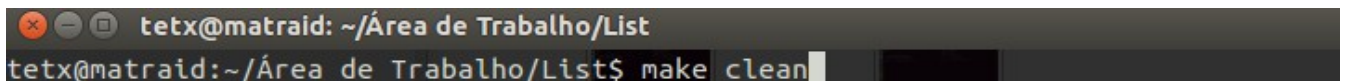
Um Makefile foi criado para facilitar a compilação dos códigos. Alguns comandos ágeis:

1. *make remove_objects* para remover os arquivos do tipo objeto que são indesejáveis após a criação do executável.
2. *make clean* para remover todos os arquivos (executável e objeto) que não são essenciais para a compilação.
3. *make* para compilar os códigos (.cpp e .h) essenciais para gerar o executável com o nome *list*.



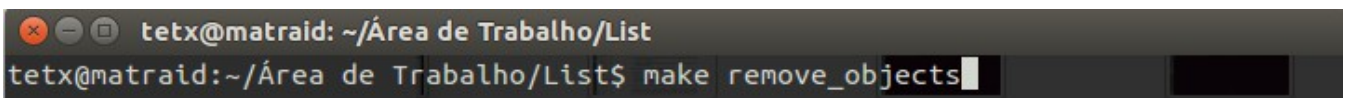
```
tetx@matraid: ~/Área de Trabalho/List
tetx@matraid:~/Área de Trabalho/List$ make
```

Figura 2.4 comando make



```
tetx@matraid: ~/Área de Trabalho/List
tetx@matraid:~/Área de Trabalho/List$ make clean
```

Figura 2.5 comando make clean



```
tetx@matraid: ~/Área de Trabalho/List
tetx@matraid:~/Área de Trabalho/List$ make remove_objects
```

Figura 2.6 comando make remove_objects

2.2 Informações Técnicas

Todo o Trabalho Prático foi desenvolvido no mesmo computador e escrito no editor de texto Sublime.

Sistema Operacional: Ubuntu 14.04 LTS 32-bits.

Processador: Intel Core i7-3612QM CPU @ 2.10GHz x 8

Gráficos: Intel Ivybridge Mobile x86/MMX/SSE2

Memória: 8GB

Versão do compilador: GCC version 4.8.2 (Ubuntu 4.8.2-19ubuntu1)

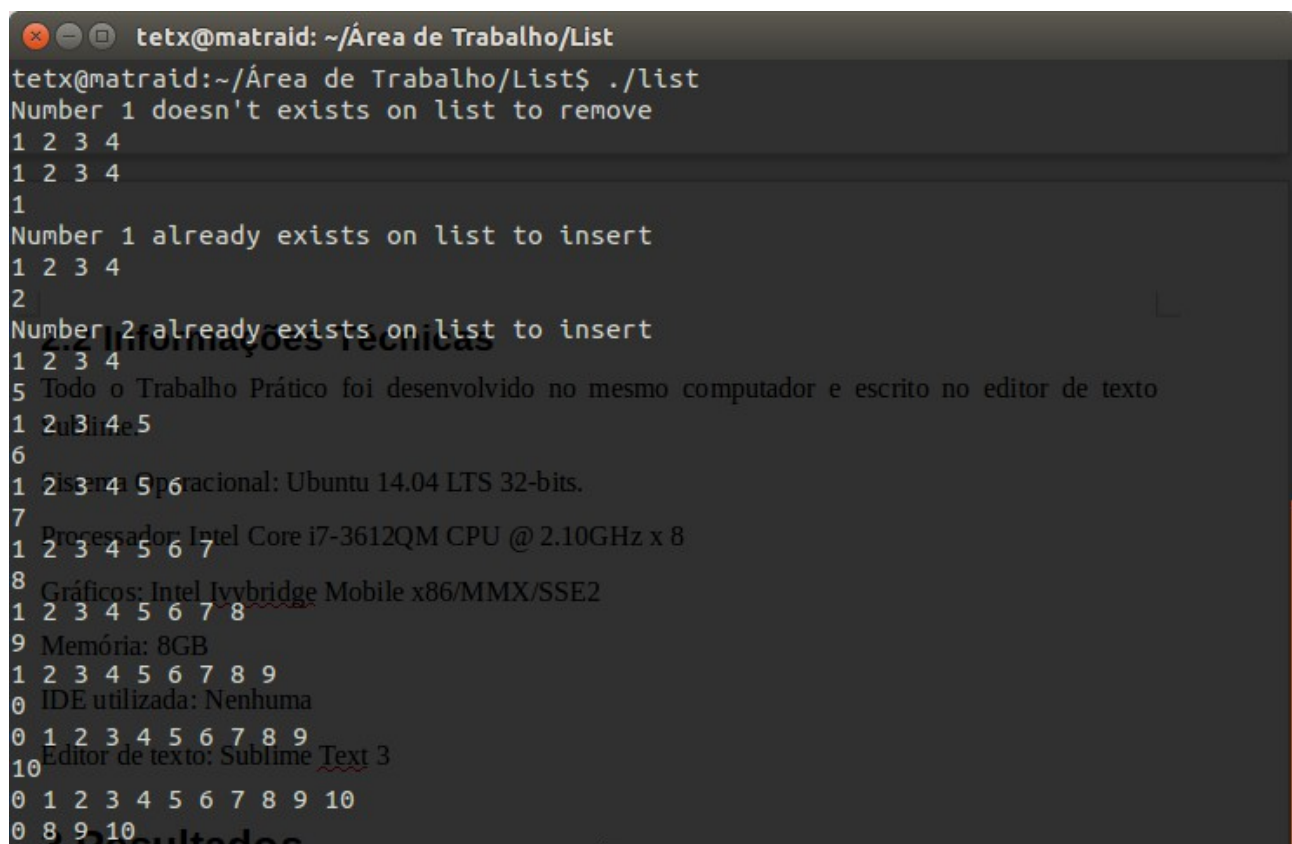
IDE utilizada: Nenhuma

Editor(es) de texto utilizado(s): Sublime Text 3 / gedit

3 Resultados

As figuras 3.1 e 3.2 ilustram os resultados coletados.

No primeiro teste vemos o funcionamento do Trabalho Prático com as entradas padrões. No segundo, os testes foram feitos com os arquivos *in.txt* e *out.txt*. Nesse mesmo segundo, vemos que os erros são registrados via console.



```
tetx@matraid: ~/Área de Trabalho/List
tetx@matraid:~/Área de Trabalho/List$ ./list
Number 1 doesn't exists on list to remove
1 2 3 4
1 2 3 4
1
Number 1 already exists on list to insert
1 2 3 4
2
Number 2 already exists on list to insert
1 2 3 4
5
Todo o Trabalho Prático foi desenvolvido no mesmo computador e escrito no editor de texto
1 2 3 4 5
6
Sistema Operacional: Ubuntu 14.04 LTS 32-bits.
7
Processador: Intel Core i7-3612QM CPU @ 2.10GHz x 8
8
Gráficos: Intel Ivybridge Mobile x86/MMX/SSE2
9
Memória: 8GB
10
IDE utilizada: Nenhuma
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 10
0 8 9 10
```

Figura 3.1 Funcionamento do programa para cin e cout

```
tetx@matraid: ~/Área de Trabalho/List
tetx@matraid:~/Área de Trabalho/List$ ./list in.txt out.txt
Number 4 already exists on list to insert
Number 5 already exists on list to insert
Number 1 already exists on list to insert
Number 7 already exists on list to insert
Number 8 already exists on list to insert
Number 800 already exists on list to insert
tetx@matraid:~/Área de Trabalho/List$

in.txt x
800 -800 12 45 67 -50 -2 -3 1 4 5 6 7 8 4 5 1 2 3 7 8 100 800 78 989 1231231

out.txt x
-800 -50 -3 -2 1 2 3 4 5 6 7 8 12 45 67 78 100 800 989 1231231
```

Figura 3.2. Funcionamento do programa para entrada e saída de arquivos

4 Conclusão

O Trabalho Prático proporcionou uma experiência bastante interessante para relembrar algumas implementações básicas de uma lista encadeada por meio de classes. Além disso, tive como principal objetivo generalizar a implementação dessa lista para que fosse uma lista de qualquer fim. Para o nosso caso, uma lista de inteiros.

Acredito que o Trabalho ficou bem organizado e bem legível. A linguagem C++ ajuda nesse quesito e os comentários ao longo do código também.

Todo o código foi escrito e comentado baseando-se na língua inglesa por preferência pessoal.

5 Bibliografia

1. <http://stackoverflow.com/>
2. <http://www.cplusplus.com>
3. ZIVIANI, Nivio. **Projeto de Algoritmos**: com implementações em PASCAL e C. 3. ed. São Paulo: Cengage Learning, 2011.