DIRETORIA DE GESTÃO E TECNOLOGIA DA INFORMAÇÃO

TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

RELATÓRIO SOBRE DEFINIÇÃO E IMPLEMENTAÇÃO DE UM VETOR DINÂMICO

Nathan Cavalcante
de Lima

Rio Grande do Norte
2024

Sumário

1 - Introdução

Vetores dinâmicos são estruturas de dados que permitem o armazenamento e a manipulação de elementos de forma eficiente. Diferentemente de arrays estáticos, que têm um tamanho fixo definido em tempo de compilação, os vetores dinâmicos podem crescer e encolher conforme necessário, o que os torna ideais para situações em que o número de elementos não é conhecido com antecedência.

2 - Vetores dinâmicos

Um vetor dinâmico é implementado utilizando arrays que podem ser redimensionados. Quando o array atinge sua capacidade máxima e é necessário adicionar mais elementos, um novo array de maior capacidade é alocado e os elementos são copiados para ele. Esse processo de redimensionamento permite que o vetor dinâmico cresça conforme necessário, mas pode ter um custo de desempenho associado à realocação e cópia dos elementos.

3 - Implementação

3.1 - Organização dos Arquivos Fontes

Os arquivos fornecidos estão organizados em duas principais categorias:

- **Headers (.hpp)**: Contêm as definições de classes, “array_list” e “linked_list”.
    - array_list.hpp: Definição da classe para manipulação de arrays dinâmicos.
    - linked_list.hpp: Definição da classe para manipulação de listas ligadas.
- **Testes (.cpp)**: Arquivos que implementam testes para verificar o funcionamento correto das classes.
    - test-pushfront-array-list-01.cpp: Testes para a operação “push_front” na classe “array_list”.
    - test-pushfront-linked-list-01.cpp: Testes para a operação “push_front” na classe “linked_list”.
    - test-removeat-array-list-01.cpp: Testes para a operação “remove_at” na classe “array_list”.
    - test-removeat-linked-list-01.cpp: Testes para a operação “remove_at” na classe “linked_list”.

Além disso, há uma pasta de testes (tests) contendo scripts e arquivos de entrada para realizar testes específicos. Os testes foram mal sucedidos devido a um erro.

```
PS C:\Users\Nathan\OneDrive\Área de Trabalho\cpp> ./test-removeat-array-list-01 <
tests/remove_at/e1.txt

No linha:1 caractere:31

+ ./test-removeat-array-list-01 < tests/remove_at/e1.txt

+                               ~

Operador '<' reservado para uso futuro.

    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException

    + FullyQualifiedErrorId : RedirectionNotSupported
```

(Mensagem de erro no prompt de comando)

3.2 - Array com alocação dinâmica

```
#ifndef __ARRAY_LIST_IFRN__
#define __ARRAY_LIST_IFRN__


class array_list {
private:
    int* data;
    unsigned int size_, capacity_;
    void increase_capacity() { //aumenta a capacidade do vetor
        int add_capacity = this->capacity_;
        int* new_data = new int[this->capacity_+add_capacity];
        for (int i=0 ; i<this->size_ ; ++i) {
            new_data[i] = data[i];
        }
        delete [] this->data;
        this->data = new_data;
        this->capacity_ = this->capacity_+add_capacity;
    }
public:
    array_list() { // Construtor
        data = new int[8];
        this->size_ = 0;
        this->capacity_ = 8;
    }
    ~array_list() { // Destrutor
        delete[] data;
    }
    unsigned int size() { // Retorna a quantidade de elementos armazenados
        return this->size_;
    }
    unsigned int capacity() { // Retorna o espaço reservado para armazenar os elementos
        return this->capacity_;
    }
     double percent_occupied() { // Retorna um valor entre 0.0 a 1.0 com o percentual da
memória usada
        return static_cast<double>(this->size_) / this->capacity_;
    }
    bool insert_at(unsigned int index, int value) { // Insere elemento no índice index
        if (index > this->size_) {
            return false; // Não inseriu
        }
        if (this->size_ == this->capacity_) {
            this->increase_capacity();
        }
        for (int i = this->size_; i > index; i--) {
```

```
            this->data[i] = this->data[i - 1];
        }
        this->data[index] = value;
        this->size_++;
        return true; // Inseriu
    }
    bool remove_at(unsigned int index) { // Remove elemento do índice index
        if (index >= this->size_) {
            return false; // Não removeu
        }
        for (int i = index + 1; i < this->size_; i++) {
            this->data[i - 1] = this->data[i];
        }
        this->size_--;
        return true; // Removeu
    }
     int get_at(unsigned int index) { //Retorna elemento no índice index, -1 se índice
inválido
        if (index >= this->size_) {
            return -1; // Índice inválido
        }
        return this->data[index];
    }
    void clear() { // Remove todos os elementos, deixando o vetor no estado inicial
        for (int i = 0; i < this->size_; i++) {
            this->data[i] = 0;
        }
        this->size_ = 0;
    }
    void push_back(int value) { // Adiciona um elemento no ``final'' do vetor
        if (this->size_ == this->capacity_) {
            this->increase_capacity();
        }
        this->data[size_++] = value;
    }
    void push_front(int value) { // Adiciona um elemento no ``início'' do vetor
        this->insert_at(0, value);
    }
    bool pop_back() { // Remove um elemento do ``final'' do vetor
        this->remove_at(this->size_--);
    }
    bool pop_front() { // Remove um elemento do ``início'' do vetor
        this->remove_at(0);
    }
    int front() { // Retorna o elemento do ``início'' do vetor
        if (this->size_ == 0) {
            return -1;
        }
        return this->data[0];
    }
    int back() { // Retorna o elemento do ``final'' do vetor
        if (this->size_ == 0) {
            return -1;
        }
        return this->data[this->size_--];
    }
    bool remove(int value) { // Remove value do vetor caso esteja presente
        int index = find(value);
        if (index == -1) {
            return false; // Valor não encontrado
        }
        this->remove_at(index);
    }
    int find(int value) { // Retorna o índice de value, -1 caso value não esteja presente
        int index = -1;
        for (int i = 0; i < this->size_; i++) {
            if (this->data[i] == value) {
```

```
                index = i;
                break;
            }
        }
        return index;
    }
    int count(int value) { // Retorna quantas vezes value occorre no vetor
        int count = 0;
        for (int i = 0; i < this->size_; i++) {
            if (this->data[i] == value) {
                count++;
            }
        }
        return count;
    }
    int sum() { // Retorna a soma dos elementos do vetor
        long long sum;
        if (this->size_ == 0){
            return 0;
        }
        for (int i = 0; i < this->size_; i++) {
            sum = sum + this->data[i];
        }
        return sum;
    }

};


#endif // __ARRAY_LIST_IFRN__
```

(código da classe "array_list")

O arquivo "array_list.hpp" implementa uma classe que gerencia arrays dinâmicos. A implementação inclui métodos para adicionar, remover e acessar elementos. Uma característica importante é a capacidade de redimensionar o array quando necessário, o que envolve a alocação de um novo array maior e a cópia dos elementos para ele.

3.2.1 - Lista de desempenho dos métodos (notação Big O):

- **Construtor ("array_list"):** Inicializa o array com capacidade de 8 elementos e tamanho 0.
  Complexidade: **O(1)**
- **Destrutor ("~array_list"):** Libera a memória alocada para o array.
  Complexidade: **O(1)**
- **increase_capacity():** Dobra a capacidade do array quando ele está cheio. Copia os elementos para o novo array.
  Complexidade: **O(n)**
- **size():** Retorna o número de elementos armazenados.
  Complexidade: **O(1)**
- **capacity():** Retorna a capacidade atual do array.
  Complexidade: **O(1)**
- **percent_occupied():** Retorna a porcentagem de ocupação do array.
  Complexidade: **O(1)**

- **get_at(unsigned int index):** Retorna o elemento na posição especificada, ou -1 se o índice for inválido.
  Complexidade: **O(1)**
- **front():** Retorna o primeiro elemento do array.
  Complexidade: **O(1)**
- **back():** Retorna o último elemento do array.
  Complexidade: **O(1)**
- **insert_at(unsigned int index, int value):** Insere um elemento na posição especificada. Se necessário, aumenta a capacidade do array.
  Complexidade: **O(n)**
- **remove_at(unsigned int index):** Remove o elemento na posição especificada.
  Complexidade: **O(n)**
- **push_back(int value):** Adiciona um elemento ao final do array.
  Complexidade: **O(1)**
- **push_front(int value):** Adiciona um elemento ao início do array.
  Complexidade: **O(n)**
- **pop_back():** Remove o último elemento do array.
  Complexidade: **O(1)**
- **pop_front():** Remove o primeiro elemento do array.
  Complexidade: **O(n)**
- **clear():** Remove todos os elementos do array.
  Complexidade: **O(1)**
- **remove(int value):** Remove a primeira ocorrência de um valor específico no array.
  Complexidade: **O(n)**
- **find(int value):** Retorna o índice da primeira ocorrência de um valor específico, ou -1 se não encontrado.
  Complexidade: **O(n)**
- **count(int value):** Conta quantas vezes um valor aparece no array.
  Complexidade: **O(n)**
- **sum():** Retorna a soma de todos os elementos no array.
  Complexidade: **O(n)**

3.3 - Lista duplamente ligada

```
#ifndef __LINKED_LIST_IFRN__
#define __LINKED_LIST_IFRN__


class linked_list {
private:
    struct int_node {
        int value;
        int_node* next, * prev;
    };
    int_node* head, * tail;
    unsigned int size_;
```

```
public:
    linked_list() {
        this->head = 0;
        this->tail = 0;
        this->size_ = 0;
    }
    ~linked_list() {
        int_node* current = this->head;
        while (current != nullptr) {
            int_node* to_remove = current;
            current = current->next;
            delete to_remove;
        }
    }
    unsigned int size() { // Retorna a quantidade de elementos armazenados
        return this->size_;
    }
    unsigned int capacity() { // Retorna o espaço reservado para armazenar os elementos
        return this->size_;
    }
     double percent_occupied() { // Retorna um valor entre 0.0 a 1.0 com o percentual da
memória usada.
        return this->size_ / static_cast<double>(this->capacity());
    }
    bool insert_at(unsigned int index, int value) { // Insere elemento no índice index
        if (index > this->size_) {
            return false;
        }

        int_node* new_node = new int_node;
        new_node->value = value;

        if (index == 0) {
            new_node->next = this->head;
            new_node->prev = nullptr;
            if (this->head != nullptr) {
                this->head->prev = new_node;
            }
            this->head = new_node;
            if (this->tail == nullptr) {
                this->tail = new_node;
            }
        } else if (index == this->size_) {
            new_node->next = nullptr;
            new_node->prev = this->tail;
            if (this->tail != nullptr) {
                this->tail->next = new_node;
            }
            this->tail = new_node;
        } else {
            int_node* current = this->head;
            for (unsigned int i = 0; i < index; ++i) {
                current = current->next;
            }
            new_node->next = current;
            new_node->prev = current->prev;
            if (current->prev != nullptr) {
                current->prev->next = new_node;
            }
            current->prev = new_node;
        }
        this->size_++;
        return true;
    }
    bool remove_at(unsigned int index) { // Remove elemento do índice index
        if (index >= this->size_)
            return false; // Não removeu
```

```
        int_node* to_remove = this->head;
        for (unsigned int i = 0; i < index; ++i)
            to_remove = to_remove->next;
        if (to_remove->prev != nullptr)
            to_remove->prev->next = to_remove->next;
        if (to_remove->next != nullptr)
            to_remove->next->prev = to_remove->prev;
        delete to_remove;
        this->size_--;
        return true; // Removeu
    }
     int get_at(unsigned int index) { // Retorna elemento no índice index, -1 se índice
inválido
        if (index >= this->size_) {
            return -1;
        }

        int_node* current = this->head;
        for (int i = 0; i < index; i++) {
            current = current->next;
        }
        return current->value;
    }
    void clear() { // Remove todos os elementos, deixando o vetor no estado inicial
        int_node* current = this->head;
        while (current != nullptr) {
            int_node* to_remove = current;
            current = current->next;
            delete to_remove;
        }
        this->head = nullptr;
        this->tail = nullptr;
        this->size_ = 0;
    }
    void push_back(int value) { // Adiciona um elemento no ``final'' do vetor
        int_node* new_node = new int_node;
        new_node->value = value;
        new_node->next = nullptr;
        new_node->prev = this->tail;

        if (this->tail != nullptr) {
            this->tail->next = new_node;
        }
        this->tail = new_node;

        if (this->head == nullptr) {
            this->head = new_node;
        }

        this->size_++;
    }
    void push_front(int value) { // Adiciona um elemento no ``início'' do vetor
        int_node* new_node = new int_node;
        new_node->value = value;
        new_node->next = this->head;
        new_node->prev = nullptr;

        if (this->head == nullptr) {
            this->tail = new_node;
        }
        else {
            this->head->prev = new_node;
        }
        this->head = new_node;
        this->size_++;
    }
    bool pop_back() { // Remove um elemento do ``final'' do vetor
```

```
        if (this->tail == nullptr) {
            return false;
        }
        int_node* to_remove = this->tail;
        this->tail = this->tail->prev;
        if (this->tail != nullptr)
            this->tail->next = nullptr;
        else {
            this->head = nullptr;
        }
        delete to_remove;
        this->size_--;
        return true;
    }
    bool pop_front() { // Remove um elemento do ``início'' do vetor
        if (this->head == nullptr) {
            return false;
        }
        int_node* to_remove = this->head;
        this->head = this->head->next;
        if (this->head != nullptr) {
            this->head->prev = nullptr;
        } else {
            this->tail = nullptr;
        }

        delete to_remove;
        this->size_--;
        return true;
    }
    int front() { // Retorna o elemento do ``início'' do vetor
        if(this->head == nullptr) {
            return -1;
        }
        return this->head->value;
    }
    int back() { // Retorna o elemento do ``final'' do vetor
        if (this->tail == nullptr) {
            return -1;
        }
        return this->tail->value;
    }
    bool remove(int value) { // Remove value do vetor caso esteja presente
        int_node* current = this->head;
        while (current != nullptr) {
            if (current->value == value) {
                if (current->prev != nullptr) {
                    current->prev->next = current->next;
                } else {
                    this->head = current->next;
                }
                if (current->next != nullptr) {
                    current->next->prev = current->prev;
                } else {
                    this->tail = current->prev;
                }
                delete current;
                this->size_--;
                return true;
            }
            current = current->next;
        }
        return false;
    }
    int find(int value) { // Retorna o índice de value, -1 caso value não esteja presente
        int_node* current = this->head;
        int index = 0;
```

```
        while (current != nullptr) {
            if (current->value == value) {
                return index;
            }
            current = current->next;
            index++;
        }
        return -1;
    }

    int count(int value) { // Retorna quantas vezes value occorre no vetor
        int_node* current = this->head;
        int count = 0;
        while (current != nullptr) {
            if (current->value == value) {
                count++;
            }
            current = current->next;
        }
        return count;
    }
    int sum() { // Retorna a soma dos elementos do vetor
        int_node* current = this->head;
        int sum = 0;
        while (current != nullptr) {
            sum += current->value;
            current = current->next;
        }
        return sum;
    }
};

#endif // __LINKED_LIST_IFRN__
```

(código da classe "array_list")

O arquivo "linked_list.hpp" implementa uma lista duplamente ligada. Esta implementação permite a inserção e remoção de elementos de forma eficiente em qualquer posição. Cada nó na lista contém um valor e ponteiros para o nó anterior e o próximo, permitindo a navegação bidirecional. Ao contrário dos arrays dinâmicos, as listas ligadas não necessitam de um bloco contínuo de memória, o que facilita a inserção e remoção de elementos.

3.3.1 - Lista de desempenho dos métodos (notação Big O):

- **Construtor (linked_list()):** Inicializa uma lista vazia com ponteiros de cabeça e cauda nulos e tamanho zero.
  Complexidade: **O(1)**
- **Destrutor (~linked_list()):** Libera a memória de todos os nós na lista.
  Complexidade: **O(n)**
- **size():** Retorna o número de elementos armazenados na lista.
  Complexidade: **O(1)**
- **capacity():** Retorna o espaço reservado para armazenar os elementos (igual ao tamanho).
  Complexidade: **O(1)**
- **percent_occupied():** Retorna a porcentagem da capacidade usada (sempre 1.0 para uma lista ligada).
  Complexidade: **O(1)**

- **insert_at(unsigned int index, int value):** Insere um elemento na posição especificada.
  Complexidade: **O(n)**
- **remove_at(unsigned int index):** Remove o elemento na posição especificada.
  Complexidade: **O(n)**
- **get_at(unsigned int index):** Retorna o elemento na posição especificada, ou -1 se o índice for inválido.
  Complexidade: **O(n)**
- **clear():** Remove todos os elementos da lista, deixando-a vazia.
  Complexidade: **O(n)**
- **push_back(int value):** Adiciona um elemento ao final da lista.
  Complexidade: **O(1)**
- **push_front(int value):** Adiciona um elemento ao início da lista.
  Complexidade: **O(1)**
- **pop_back():** Remove o último elemento da lista.
  Complexidade: **O(1)**
- **pop_front():** Remove o primeiro elemento da lista.
  Complexidade: **O(1)**
- **front():** Retorna o primeiro elemento da lista.
  Complexidade: **O(1)**
- **back():** Retorna o último elemento da lista.
  Complexidade: **O(1)**
- **remove(int value):** Remove a primeira ocorrência de um valor específico na lista.
  Complexidade: **O(n)**
- **find(int value):** Retorna o índice da primeira ocorrência de um valor específico, ou -1 se não encontrado.
  Complexidade: **O(n)**
- **count(int value):** Conta quantas vezes um valor aparece na lista.
  Complexidade: **O(n)**
- **sum():** Retorna a soma de todos os elementos na lista.
  Complexidade: **O(n)**