UFC

Relatório

Projeto 01- TAD Lista Circular Duplamente Encadeada e TAD Conjunto

Disciplina: Estruturas de Dados
Professor: Atílio Gomes Luiz

Aluna: Aline Araujo Souza
Matrícula: 473383
Aluno: Davi Teixeira Silva
Matrícula: 433951

Quixadá - 09/2020

1. **Listagem dos programas em c++**

1.1 Lista Circular Duplamente Encadeada

1.1.1 Lista.h

Contém o struct Node e os cabeçalhos das funções que envolvem a lista circular duplamente encadeada.

1.1.2 Lista.cpp

Contém as implementações das funções que serão executadas na lista.

1.1.3 Main.cpp

Contém a função main, o qual pertence o corpo do nosso programa junto com o menu interativo, onde o usuário poderá realizar o teste de todas as funções chamadas pelo arquivo Lista.h presentes no arquivo Lista.cpp solicitadas pelo trabalho.

1.2 TAD Conjunto

1.2.1 conjunto.h

Contém a classe Conjunto composta por um vetor de inteiros, variáveis tamanho, capacidade, e os cabeçalhos das funções que envolvem o TAD Conjunto.

1.2.2 conjunto.cpp

Contém as implementações das funções que serão executadas na lista.

1.2.3 main.cpp

Contém a função main, o qual pertence o corpo do nosso programa junto com o menu interativo, onde o usuário poderá realizar o teste de todas as funções chamadas pelo arquivo conjunto.h presentes no arquivo conjunto.cpp solicitadas pelo trabalho.

2. **Objetivo**

O objetivo desse primeiro projeto é a implementação de duas estrutura de dados usando a linguagem de programação C++ pelos alunos Aline Araujo Souza e Davi Teixeira Silva.

A primeira estrutura é uma TAD de uma Lista Circular Duplamente Encadeada, onde foram implementadas 20 funções visando analisar o desenvolvimento dos estudantes. A segunda estrutura é um TAD Conjunto, onde os estudantes aplicaram os conceitos matemáticos de conjuntos em uma lista sequencial para realizar a sua implementação.

3. **Divisão do trabalho e dificuldades encontradas**

O trabalho foi realizado em dupla, no começo foi sendo implementado o menu e os estudos de forma online por ambos, contudo com a evolução e complexidade que o trabalho apresentava, foi necessário que o projeto fosse feito de forma presencialmente.

Foi necessário um pouco mais de duas semanas para a conclusão do projeto, onde as atividades realizadas (revisão de aulas, estudos e pesquisas,

implementação das funções e realização do relatório), foram feitas em conjunto por ambos os integrantes.

As dificuldades encontradas, como citado acima foi a questão da distância dos membros da equipe, pois ambos moram em cidades diferentes, outro fator que atrapalhou foi a gestão de tempo e complexidade das funções para a realização do trabalho.

4. Descrições

4.1 Lista Circular Duplamente Encadeada e funções utilizadas

Por ser uma lista circular, não existe o conceito de primeiro da lista e último da lista, o que facilita a manipulação, permitindo percorrer a lista nos dois sentidos, cada elemento guarda além do seu próprio valor, um ponteiro para o próximo elemento e um ponteiro para o elemento anterior, sem a obrigatoriedade de estarem no mesmo espaço de memória.

4.2 Descrição e Complexidades das funções:

4.2.1 **List()**: Possuindo complexidade O(1), o Construtor da classe apenas cria um nó alocando e "setando" os campus da Struct Node;

4.2.2 **~List()**: Destrutor da classe, possui complexidade O(1), pois libera memória previamente alocada;

4.2.3 **void push back(int key)**: Insere um inteiro key ao final da lista, possui complexidade O(1), ela vai adicionar um novo nó ao final da lista, o último nó aponta para o novo nó, e faz o novo nó apontar para o primeiro;

4.2.4 **int pop back()**: Remove elemento do final da lista e retorna seu valor, sua complexidade é O(1), remove o último elemento e faz o seu anterior apontar para o início da lista;

4.2.5 **void insertAfter(int key, int k)**: Insere um novo nó com valor key após o k-ésimo nó da lista, sendo assim uma função O(n), pois ao ser adicionado um novo elemento, todas as posições dos nós posteriores devem ser realocadas;

4.2.6 v**oid removeNode(Node *p)**: Essa função é O(1), pois remove da lista o nó apontado pelo ponteiro p, fazendo que o próximo nó aponte para o anterior e no o anterior de p aponte para o próximo;

4.2.7 **void remove(int key)**: Essa função é O(1), pois remove da lista a primeira ocorrência do inteiro key, e puxa os valores das posições posteriores do valor removido;

4.2.8 **void removeAll(int key)**: Essa função é O(n), pois visto que percorre um vetor de inteiro removendo da lista todas as ocorrências do inteiro key;

4.2.9 **int removeNodeAt(int k)**: Remove o k-ésimo nó da lista encadeada e retorna o seu valor, sendo uma função O(n), busca o nó dentre a lista, guarda o valor daquela posição e deleta o nó usando a função removeNode();

4.2.10 **void print()**: Uma função O(1), pois imprime os elementos da lista;

4.2.11 **void printReverse()**: Uma função O(1), pois imprime os elementos da lista em ordem reversa;

4.2.12 **bool empty()**: Uma função O(1), faz uma comparação e retorna true se a lista estiver vazia e false caso contrário;
4.2.13 **int size()**: Uma função O(1), pois com um while ela conta e retorna o número de nós da lista;
4.2.14 **void clear()**: Sendo uma Função O(n), ela percorre toda a lista nas duas direções certificando que seja removido todos os elementos dos nós alocados deixando assim apenas o nó cabeça;
4.2.15 **void concat(List *lst)**: Uma Função O(n), que concatena a lista atual com a lista lst passada por parâmetro, primeiro verificando se a lista passada não está vazia, concatena assim os valores com a função push_back(), limpa a lista passada com a função clear().e por fim remove um nó auxiliar com a função removeNode();
4.2.16 **List *copy()**: Uma função O(n), pois retorna um ponteiro para uma cópia desta lista, e caso a lista original tenha elementos a função *copy() tem que adicioná-los a nova lista, usando assim a função push_back();
4.2.17 **void copyArray(int n, int arr[])**: Uma função O(n), pois copia os elementos do array arr para a lista, tendo que criar todos os procedimentos da lista para acrescentá-los;
4.2.18 **bool equal(List *lst)**: Uma função O(n), pois recebe duas lista de tamanho indefinido, cria nós auxiliares para ajudar na manipulação, percorre todas as posições das duas listas e verifica se a lista passada por parâmetro é igual à lista em questão;
4.2.19 **List* separate(int n)**: Função O(n), pois recebe como parâmetro um valor inteiro n é divide a lista em duas, de forma à segunda lista começar no primeiro nó logo após a primeira ocorrência de n na lista original. Infelizmente a equipe não concluiu a implementação a tempo para a entrega do trabalho.
4.2.20 **void merge lists(List *list2)**: Esta é uma função O(n), pois recebe uma List como parâmetro e verifica se ela não está vazia, constrói uma nova lista com auxílio da função insertAfter(), intercalando os nós da lista original com os nós da lista passada por parâmetro, alternando a posição de inserção sempre de 2 em 2, depois chamo o removeNode() , assim será removido o nó auxiliar criado para percorrer a lista.

4.3 TAD Conjunto e funções utilizadas

O uso de estruturas se fez importante na programação, usando TAD, com ele o programador garante que os elementos são acessíveis apenas via os algoritmos correspondentes a lista de funções predeterminada no nosso arquivo conjunto.h, assim é possível realizar diversas operações entre os conjuntos.

4.4 Descrição e Complexidades das funções:

4.4.1 **união(A,B,C):** recebe os conjuntos A e B como parâmetro e retorna o conjunto C = A ∪ B. Função O(n), verifica todos os elementos do conjunto

passado, com a função membro(), depois verifica se o elemento já existe no conjunto principal, senão insere o elemento.

4.4.2 **intersecção(A,B,C):** recebe os conjuntos A e B como parâmetro e retorna o conjunto C = A ∩ B. Função O(n), pois usa a função membro() verificando se o valor passado é membro dos dois conjuntos, se sim, adiciona ao vetor temporário.

4.4.3 **diferença(A,B,C):** recebe os conjuntos A e B como parâmetro e retorna o conjunto C = A - B. Função O(n), depois de selecionado o conjunto, é chamada a função membro(), para certificar que o valor pertence ao conjunto, ao encontrar a posição correspondente ao valor que quer remover , modifica as posições, e diminui o tamanho do conjunto.

4.4.4 **diferença simétrica(A,B,C):** recebe os conjuntos A e B como parâmetro e retorna o conjunto C = A O B. Infelizmente a equipe não concluiu a implementação a tempo para a entrega do trabalho.

4.4.5 **membro(y,A):** recebe o conjunto A e um elemento y e retorna um 1 se y ∈ A e 0 caso contrário. Função O(1), pois compara o elemento passado a todas as posições do vetor.

4.4.6 **criaConjVazio():** cria um conjunto vazio e retorna o conjunto criado. Possuindo complexidade O(1), o Construtor da classe apenas cria um conjunto alocando e "setando" os campus da classe Conjunto;

4.4.7 **insere(y,A):** recebe o conjunto A e um elemento y e adiciona y ao conjunto A, isto é, A = A ∪ y. Possui complexidade O(1), ela vai adicionar um novo elemento ao final do conjunto, caso seja atingido o limite, a função realoca mais memória e aumenta a capacidade do conjunto de acordo com a quantidade de elementos passados ;

4.4.8 **remove(y,A):** recebe o conjunto A e um elemento y e remove y do conjunto A, isto é, A = A - y. Essa função é O(1), pois remove do conjunto o inteiro y, e puxa os valores das posições posteriores do valor removido;

4.4.9 **copia(A,B):** faz uma cópia do conjunto A em B. Essa função é O(n), pois copia todos os elementos do conjunto principal para o secundário.

4.4.10 **min(A):** É uma função O(1), pois retorna o valor mínimo do conjunto escolhido;

4.4.11 **max(A):** É uma função O(1), pois retorna o valor máximo do conjunto escolhido;

4.4.12 **igual(A,B):** É uma função O(n) pois retorna true se os conjuntos A e B são iguais e false caso contrário, passando assim por vários testes para saber se os valores são iguais usando a função membro().