Relatório

Nome: João Victor Dias Barroso
Matrícula: 474110
Professor: Atílio Gomes

1 Introdução

Um dos problemas consiste em implementar uma Lista duplamente encadeada circular, com ou sem nó cabeça. A minha implementação usará nó cabeça, pois achei a implementação mais interessante. Foram dadas diversas questões para implementarmos. O outro consiste em, criar operações de conjunto, como: união, intersecção, diferença, entre outros, nesta implementação foi usado lista circular simplesmente encadeada com nó cabeça. Além disso foi criado um menu interativo para testar, executar cada questão.

2 Grupo

Por questões práticas resolvi fazer sozinho. Então todas as tarefas de implementação, documentação foram feitas exclusivamente por mim.

3 Metodologia - Questão 1

Como disse, na introdução, a minha implementação da Lista duplamente encadeada circular usou nó cabeça. E foi utilizado classes na implementação. Inicialmente foi pensado, como era esta lista vazia, então comecei com isso. Ou seja, a primeira codificação feita foi o construtor. Basicamente, a ideia é ter um struct nomeado Node, que tem como elementos uma chave ou key, um ponteiro para o próximo nó e outro para o nó anterior.

3.1 Construtor

```
struct Node {
    int key;
    Node *next;
    Node *ant;

};

List::List() {
    head = new Node;
    head->ant = head;
    head->next = head;
}
```

Nesta implementação é usada uma ponteiro privado da classe chamada **head** que é um ponteiro que aponta para um nó que para esta implementação é inválido. Este ponteiro é alocado, e o anterior dele aponta para ele mesmo, assim como o próximo. Assim se configura uma lista vazia.

3.2 Funções

3.2.1 void pushBack(int key)

Considerando que o anterior do nó cabeça sempre aponta para o último nó, foi usada esta informação para uma inserção no fim da lista em tempo **O(1)**.

3.2.2 int popBack()

Assim como inserir ao final da lista, remover do fim da lista também leva **O(1)** para ser concluído. Ou seja o último nó que é igual a **head->ant** é removido e as coisas que precisam ser feitas após, são somente fazer este apontar para o novo último nó, e o próximo deste apontar para a cabeça.

3.2.3 void insertAfter(int key, int index)

No caso desta função, não foi pensada numa solução muito rápida, pois como é necessário inserir um valor após um determinado índice, tem que, no pior caso percorrer toda a lista, ou seja **O(n)**. Então o que foi feito basicamente, percorrer a lista até um contador ser menor que o índice passado por parâmetro, quando este *loop* para ele deixa um ponteiro para um nó auxiliar apontar para o anterior do elemento que corresponde ao índice passado. E com este é possível inserir o novo nó na posição correspondente.

3.2.4 void removeNode(Node *p)

Esta função como retorna um nó e sabendo que vai ser usada exclusivamente dentro da classe, foi implementada como uma função privada. Primeira coisa a

ser feita é procurar o nó que aponta para o mesmo endereço que o nó passado por parâmetro aponta. E, de posse deste valor é possível removê-lo da lista.

3.2.5 void remove(int key)

Esta função recebe uma chave como parâmetro e remove o primeiro nó ao qual esta chave corresponde. Então com a função **removeNode** já implementada é possivel usá-la para a implementação desta função. Se existir uma chave na lista correspondente a que foi passada por parâmetro, a função removeNode é chamada passando o nó correspondente.

3.2.6 Node *searchNode(int key)

Esta função recebe uma chave como parâmetro e retorna o nó que corresponde a chave na lista, se esta existir, se não retorna a cabeça da lista. É uma função auxiliar privada que foi usada exclusivamente para auxiliar em algumas funções de remoção.

3.2.7 void removeNodeAt(int index)

Recebe um indíce como parâmetro e remove o nó daquela posição. Primeiramente é feita uma verificação se o *index* passado está no intervalo entre 0 e o tamanho da lista. Se estiver neste intervalo, um contador é criado e incrementado até ser menor que o *index* e assim eu tenho um ponteiro auxiliar apontando pro nó a remover, chamo a função **removeNode(aux)** e retorno a chave.

3.2.8 void removeAll(int key)

Esta função recebe uma chave e remove todas as ocorrências desta. Com a função de buscar por um nó ja implementada, só é necessário, fazer:

```
1 while(searchNode(key) != head)  remove(key);
```

3.2.9 void print()

Esta função imprime todos os elementos da lista. Ou seja, foi criado um auxiliar que aponta para o primeiro nó válido, ou seja, **head->next** e enquanto este fosse diferente de **head** imprimia o **aux->value** e incrementava usando aux->next.

3.2.10 void printReverse()

Função que imprime a lista ao contrário. A mesma lógica da função anterior, mas neste caso o auxiliar aponta para o último nó válido, ou seja **head->ant**. Neste caso **aux = aux->ant**.

3.2.11 void concat(List *list)

Esta função concatena duas listas, ou seja, a ideia é fazer o próximo último nó da primeira lista, ou seja **head->ant** apontar para o primeiro nó válido da lista passada por parâmetro. Para isso. simplesmente cria-se um auxiliar apontando para **(list->head)->next** e usa-se a função pushBack para cada valor contido nos nós.

3.2.12 bool isEmpty()

Retorna *true* se se head == head->next e *false* caso contrário.

3.2.13 List *copy()

Esta função cria uma cópia de uma lista. Nela foi criada uma nova lista e todos os elementos da lista original foram adicionados com pushBack nesta nova lista e retorno esta nova lista.

3.2.14 void copyArray(int arr[ ], int size)

Esta função adiciona todos os elementos do array passado por parâmetro, ou seja, para cada valor do array eu adiciono na ultima posição da lista usando a função pushBack. Como todos os valores do array precisam ser adicionados no pior caso este algoritmo leva tempo **O(n)**, onde n é igual a size.

3.2.15 bool equal(List *list)

Função que verifica se as listas são iguais em tamanho e valores. Primeira verificação se as listas são vazias ou se os tamanhos são diferentes, isso configura listas diferentes. Se passar por essas verificações um contador é criado e conta os elementos das duas listas comparando a igualdade, de cada valor em suas respectivas posições. Quando esse laço acabar retorna-se true se o contador for igual a size e false caso contrário. Como todos os valores do nó precisam ser testados, o pior caso desta implementação leva tempo **O(n)**.

3.2.16 void clear()

Se a lista não estiver vazia, deleta todos os nós. Na implementação foi criado um auxiliar que aponta para o último nó válido da lista. Um laço é feito usando este ponteiro para cada "iteração"chama-se a função popBack() que vai removendo o último enquanto houver elementos.

3.2.17 List *separate(int key)

Função que separa a lista após a chave passada por parâmetro.

3.2.18 List* mergeLists(List *list)

Função que intercala os nós da lista passada por parâmetro com a lista que referenciou a função. Implementada criando uma nova lista, a cada elemento de cada lista foi dado um pushBack na key para a nova lista, e no fim esta foi retornada.

3.2.19 List() - Destrutor

Destrutor da lista, chamado automaticamente toda vez que o objeto é destruído.

4 Metodologia - Questão 2

Na questão 2 foi usado classes para implementação e a lista simplesmente encadeada circular com nó cabeça para representação dos conjuntos. Por uma questão de programador todas as funções foram nomeadas em inglês. Assim como o nome dos arquivos que são: Set.cpp, main.cpp e Set.h.

4.1 Construtor

```
1  struct Node {
2      int value;
3      Node *next;
4  };
5
6  // Lista circular simplesmente encadeada
7  Set::Set() {
8      head = new Node;
9      head->next = head;
10 }
```

Como é possível perceber no construtor um né cabeça é alocado e o próximo nó ao qual ele aponta é ele mesmo, configurando uma lista vazia.

4.2 Funções

4.2.1 void insert(int value)

Essa função adiciona um elemento ao fim do conjunto.

4.2.2 Set *unionSet(Set *set1, Set *set2)

Esta função recebe dois conjuntos por parâmetro e retorna a união destes que no caso, é basicamente fazer o último nó do conjunto 1 receber o endereço do primeiro nó válido do conjunto 2 e retornar como um novo conjunto.

4.2.3 Set *intersectionSet(Set *set1, Set *set2)

Retorna a intersecção dos conjuntos 1 e 2, ou seja, todos os elementos que estão contidos em Set1 e Set2 ao mesmo tempo. A estratégia usada foi, comparar as chaves dos nós e usar a função insert para uma nova lista os valores que são iguais nas duas listas.

4.2.4 Set *diference(Set *set1, Set *set2)

Retorna a diferença dos conjuntos 1 e 2. A implementação desta função foi, verificar todos os elementos que só pertencem ao conjunto 1 e retornar somente eles.

4.2.5 int min(Set *set)

Retorna o menor elemento do conjunto. Primeiramente verifica-se há elementos no conjunto, compara todos os elementos com a chave do primeiro nó e verifica se existe algum menor que esse, no fim retorna o valor do menor atualizado. Nesta implementação no pior caso, o algoritmo vai levar tempo O(n), por que todos os nós devem ser percorridos.

4.2.6 Node *search(int value)

Retorna a primeira ocorrência de value no conjunto, se houver. Se não retorna o nó cabeça do conjunto.

4.2.7 int max(Set *set)

Retorna o maior elemento do conjunto. Primeiramente verifica-se há elementos no conjunto, compara todos os elementos com a chave do primeiro nó e verifica se existe algum maior que esse, no fim retorna o valor do menor atualizado. Nesta implementação no pior caso, o algoritmo vai levar tempo O(n), por que todos os nós devem ser percorridos.

4.2.8 int size(Set *a)

Retorna o tamanho do conjunto, contando todos os nós/.

4.2.9 bool isEmptySet(Set *a)

Retorna *true* se conjunto vazio, retorno a comparação de **head == head->next** se esta comparação retornar verdadeiro significa conjunto vazio.

4.2.10 void remove(int value)

Remove o nó com a primeira ocorrência de value. Com a função de buscar um nó implementada, só é necessário criar um ponteiro para nó para receber o retorno de search, ou seja **Node *noRem = search(value)**, se a função

search retornar um no válido, cria-se um auxiliar que aponta para onde a cabeça aponta e percorre este até ser igual ao **noRem**, quando isso acontecer só ajustar os ponteiros e deletar noRem.

4.2.11 bool isEqual(Set *a, Set *b)

Esta função retorna *true* se os conjuntos a e b são iguais e *false* caso contrário. Nesta implementação se um dos dois conjuntos for vazio ou tiverem tamanhos diferentes o retorno é falso. Se não, só é necessário comparar cada chave do conjunto A com cada chave do conjunto B, se no fim um contador for igual ao tamanho fda lista retorna verdadeiro.

4.2.12 bool contains(int value)

Retorna verdade se o conjunto contém o valor passado por parâmetro e falso caso contrário.

4.2.13 int member(int value)

Com a função de contém implementada, só é necessário chamá-la passando o valor e retornar 1 se é membro e 0 caso contrário.

4.2.14 Set *simetricDiference(Set *a, Set *b)

Retorna a diferença simétrica entre os dois conjuntos. Diferença simétrica é basicamente, tudo que não é iontersecção dos dois conjuntos. Na implementação foi usado como auxiliar a função de diferença, ou seja, primeiro foi chamado diference(a, b) e depois diference(b, a) e assim retornando esses conjuntos.

5 Dificuldades

Inicialmente não houveram tantas dificuldades, mas ao longo das implementações algumas apareceram, como: Inserir depois de um índice específico sem uso de vetor, intercalar nós de duas listas. Ocorreram dificuldades neste relatório, pois não sabia bem como estruturá-lo, então decidi fazer explicação de função por função.