Estrutura de Dados - Lista 4

José Gildásio Freitas do Ó - 473901

Junho 2023

1. Apresente o pseudocódigo de uma função RemoveRepetidos(v) que recebe como entrada o nó cabeça v de uma pilha, remove os nós com chaves repetidas (mantendo apenas o que está mais acima na pilha) e retorna a pilha resultante. Qual a complexidade desta função?

   - **Resposta:**
     - **Algoritmo:** RemoveRepetidos_pilha(Pilha P)
     - **Entrada:** Pilha
     - **Saida:** Pilha com os nós de chaves repetidas removidos.
     - **Complexidade:** O(n^2)

```
1– Pilha aux = criarPilha(P.tam)
2– int x
3– enquanto P.topo ≠ -1 faça
4– |   x = P.V[P.topo]
5– |   P = remover_da_pilha(P, x)
6– |   se x ≠ P.V[P.topo'] então
7– |   |   aux = inserir_na_pilha(aux, x)
8– enquanto aux.topo ≠ -1 faça
9– |   x = aux.V[aux.topo]
10– |   aux = remover_da_pilha(aux, x)
11– |   P = inserir_na_pilha(P, x)
12– return P
```

2. O seu método RemoveRepetidos(v) da questão anterior também funciona caso a estrutura passada como entrada seja uma fila? Ou seja, este método removeria todos os nós com chaves repetidas, mantendo apenas o que está mais a frente na pilha? Justifique.

   - **Resposta:** No caso da fila tem sua organização diferente, os métodos implementados são outros pelo fato de o primeiro elemento inserido será o primeiro a ser removido.
     - **Algoritmo:** RemoveRepetidos_fila(Fila F)
     - **Entrada:** Fila
     - **Saída:** Fila com os nós com chaves repetidas removidas.
     - **Complexidade:** O(n^2)

```
1– Fila aux = criarFila(F.tam)
2– int x
3– enquanto F.card ≠ 0 faça
4– |   x = F.V[F.i]
5– |   F = remover_fila(F, &x)
6– |   se x ≠ F.V[F.i] então
7– |   |   aux = inserir_na_fila(aux, x)
8– enquanto aux.card ≠ 0 faça
9– |   x = aux.V[aux.i]
10– |   aux = remover_fila(aux, &x)
11– |   F = inserir_na_fila(F, x)
12– return F
```

3. Suponha que apliquemos a ideia de uma estrutura duplamente encadeada a pilha e fila. Considerando apenas os métodos vistos em sala (criar, topo/frente, incluir e remover), há vantagem na aplicação desta estrutura sobre uma pilha simples? E sobre uma fila simples?
   Se houver vantagem, aponte o(s) método(s) no(s) qual(is) há vantagem. Se não houver vantagem, justifique.
   Apresente os pseudocódigos das operações Criar, Incluir, Remover e Buscar para essas estrutura de dados.

   - **Resposta:** Considerando o conceito de duplamente encadeada auxilia na manipulação da pilha e fila, mas aumenta a dificuldade de implementar as suas estruturas, Sendo possível acessar o ultimo nó de forma mais rápida do que na simples.

- **Pilha_Dupla_Encadeada:**
  - **Algoritmo:** criarPilhaDuplaEncadeada()
  - **Entrada:** Nenhuma entrada.
  - **Saída:** Retorna a pilha criada.
  - **Complexidade:** O(1)

```
1– Pilha_dupla_encadeada P
2– P.cabeca = NULL
3– P.cauda = NULL
4– P.card = 0
5– return P
```

  - **Algoritmo:** inserir_na_pilha_encadeada(Pilha_dupla_encadeada P, int x)
  - **Entrada:** P pilha a qual vai inserir e o elemento x que o valor a ser inserido.
  - **Saída:** Pilha com o valor x inserido.
  - **Complexidade:** O(1)

```
1– NO *novo = (NO *) malloc(sizeof(NO))
2– novo → chave = x
3– novo → prox = NULL
4– novo → ant = P.cauda
5– se P.cabeca == NULL então
6– |   P.cabeca = novo
7– |   P.cauda = novo
8– senão
9– |   P.cauda → prox = novo
10– |   P.cauda = novo
11– P.card++
12– return P
```

- **Algoritmo:** remover_da_pilha_dupla_encadeada(Pilha_dupla_encadeada P, int x)
- **Entrada:** Pilha e x que é o elemento a ser removido da Pilha.
- **Saída:** Pilha com o valor x removido.
- **Complexidade:** O(n)

```
1– NO *aux = P.cabeca
2– NO *ant = NULL
3– enquanto aux ≠ NULL && aux→ ≠ x faça
4– |   ant = aux
5– |   aux = aux→ prox
6– se aux == NULL então
7– |   cout << "valor não encontrado"
8– |   return P
9– se ant == NULL então
10– |   P.cabeca = aux→prox
11– senão
12– |   ant→prox = aux→prox
13– se aux→prox == NULL então
14– |   P.cauda = aux→ant
15– senão
16– |   aux→prox→ant = aux→ant
17– free(aux)
18– P.card = P.card - 1
19– return P
```

- **Algoritmo:** buscar_na_pilha_dupla_encadeada(Pilha_dupla_encadeada P, int x)
- **Entrada:** Pilha que vai ser feita a busca e o x que é o elemento a ser buscado.
- **Saída:** No que a chave é x.
- **Complexidade:** O(n)

```
1– NO *aux = P.cabeca
2– enquanto aux ≠ NULL && aux→chave ≠ x faça
3– |   aux = aux→prox
4– return aux
```

- **Fila_Dupla_Encadeada**
  - **Algoritmo:** criarFilaDuplaEncadeada()
  - **Entrada:** Nenhuma entrada.
  - **Saída:** Retorna a fila criada.
  - **Complexidade:** O(1)

```
1– Fila_dupla_encadeada F
2– F.cabeca = NULL
3– F.cauda = NULL
4– F.card = 0
5– return F
```

  - **Algoritmo:** inserir_na_fila_encadeada(Fila_dupla_encadeada F, int x)
  - **Entrada:** F fila a qual vai inserir e o elemento x que o valor a ser inserido.
  - **Saída:** fila com o valor x inserido.
  - **Complexidade:** O(1)

```
1– NO *novo = (NO *) malloc(sizeof(NO))
2– novo → chave = x
3– novo → prox = NULL
4– novo → ant = F.cauda
5– se F.cabeca == NULL então
6– |   F.cabeca = novo
7– |   F.cauda = novo
8– senão
9– |   F.cauda → prox = novo
10– |   F.cauda = novo
11– F.card++
12– return F
```

- **Algoritmo:** remover_da_fila_dupla_encadeada(Fila_dupla_encadeada F, int x)
- **Entrada:** Fila e x que é o elemento a ser removido da Fila.
- **Saída:** Fila com o valor x removido.
- **Complexidade:** O(n)

```
1– NO *aux = F.cabeca
2– NO *ant = NULL
3– enquanto aux ≠ NULL && aux→ ≠ x faça
4– |   ant = aux
5– |   aux = aux→ prox
6– se aux == NULL então
7– |   cout << "valor não encontrado"
8– |   return F
9– se ant == NULL então
10– |   F.cabeca = aux→prox
11– senão
12– |   ant→prox = aux→prox
13– se aux→prox == NULL então
14– |   F.cauda = aux→ant
15– free(aux)
16– F.card = F.card - 1
17– return F
```

- **Algoritmo:** buscar_na_fila_dupla_encadeada(Fila_dupla_encadeada F, int x)
- **Entrada:** Fila que vai ser feita a busca e o x que é o elemento a ser buscado.
- **Saída:** No que a chave é x.
- **Complexidade:** O(n)

```
1– NO *aux = F.cabeca
2– enquanto aux ≠ NULL && aux→chave ≠ x faça
3– |   aux = aux→prox
4– se aux == NULL então
5– |   cout << "não encontrado"
6– |   return NULL
7– return aux
```

4. Em sala, vimos a aplicação de pilha sequencial, onde sua estrutura é representada por um vetor. Porém, podemos adaptar esta ideia para representar, ao mesmo tempo, duas pilhas em um único vetor. Apresente os pseudocódigos das operações Criar, Incluir, Remover e Buscar para essa estrutura de dados. Determine a complexidade de cada uma dessas operações.

   - **Resposta:**
     - **Algoritmo:** criarPilhaDividida(int tam)
     - **Entrada:** Tamanho da Pilha
     - **Saída:** Pilha dividida em duas
     - **Complexidade:** O(1)

```
1– Pilha_dividida P
2– P.V = (int *) malloc(tam * sizeof(int))
3– P.tam = tam
4– P.topo1 = -1
5– P.topo2 = tam
6– return P
```

     - **Algoritmo:** inserir_na_pilha_dividida(Pilha_dividida P, int x, int pilha)
     - **Entrada:** Pilha, valor que vai ser inserido e qual das pilhas vai ser inserido o valor.
     - **Saída:** Pilha com o valor inserido.
     - **Complexidade:** O(1)

```
1– se P.topo1 == P.topo2 então
2– |   cout << "Pilha Cheia"
3– |   return P
4– senão
5– |   se pilha == 1 então
6– |   |   P.topo1++
7– |   |   P.V[P.topo1] = x
8– |   senão
9– |   |   P.topo2 = P.topo2 - 1
10– |   |   P.V[P.topo2] = x
11– return P
```

– **Algoritmo:** remover_da_pilha_dividida(Pilha_dividida P, int x, int pilha)
– **Entrada:** Pilha, valor que vai ser removido e qual pilha terá esse valor removido.
– **Saída:** Pilha com valor removido.
– **Complexidade:** O(1)

```
1– se pilha == 1 então
2– |   se P.topo1 == -1
3– |   |   cout << "Pilha Vazia"
4– |   |   return P
5– |   senão
6– |   |   x = P.V[P.topo1]
7– |   |   P.topo1 = P.topo1 - 1
8– senão
9– |   se P.topo2 == P.tam então
10– |   |   cout << "Pilha Vazia"
11– |   |   return P
12– |   senão
13– |   |   x = P.V[P.topo2]
14– |   |   P.topo2++
15– return P
```

– **Algoritmo:** busca_na_pilha_dividida(Pilha_dividida P, int x)
– **Entrada:** Pilha, x o valor a ser buscado na pilha
– **Saída:** índex do valor que foi encontrado.
– **Complexidade:** O(P.tam)

```
1– int i
2– para i = 0, i <= P.topo1, i++ faça
3– |   se P.V[i] == x então
4– |   |   return i
5– para i = P.tam -1, i >= P.topo2, i++ faça
6– |   se P.V[i] == x então
7– |   |   return i
8– return -1
```