	**UNIVERSIDADE FEDERAL DO PARÁ** **FACULDADE DE ENGENHARIA DA COMPUTAÇÃO E TELECOMUNICAÇÕES** **DATA: 13 / 09 / 2021**	**Lista 05**

Alunos: Caio Bernardo Brasil – 202006840008
Daniel Cordeiro Campos - 202006840045
Gabriel Silva Ribeiro – 202007040021
Heitor Mesquita Anglada – 202006840018

1.1- Os arranjos podem ser considerados como as estruturas de dados mais simples, tendo uma notável limitação: São de tamanho fixo.

Listas encadeadas tem a vantagem de ter um tamanho variável, novos itens podem ser adicionados, o que aumenta o seu tamanho.

1.2- Não é necessário definir, no momento da criação da lista, o número máximo de elementos que esta poderá ter. Ou seja, é possível alocar memória "dinamicamente", apenas para o número de nós necessários.

Lista também é usado para várias estruturas de dados concretas que podem ser usadas para implementar listas abstratas, especialmente listas encadeadas.

As chamadas estruturas de lista estática permitem apenas a verificação e enumeração dos valores. Uma lista mutável ou dinâmica pode permitir que itens sejam inseridos, substituídos ou excluídos durante a existência da lista.

2- A forma de acessar os elementos de um arranjo é direta, ao contrário das listas. Isto quer dizer que o elemento desejado obtêm-se a partir do seu índice e não é preciso procurá-lo elemento por elemento. No caso das listas, por exemplo, para alcançar o terceiro elemento terá de aceder primeiro aos dois anteriores (ou bem de guardar um ponteiro que permita aceder de maneira rápida a esse elemento em particular).

3-

```
public void insereEm(int posicao, Elemento<T> e) throws Exception{
    int aux = 0;
    Elemento<T> atual = this.frente;
```

```
        while(aux < posicao - 1){
          atual = atual.getProximo();
          aux++;

        }
        this.insereApos(atual, e.getChave(), e.getDados());

    }
```

a) A inserção pela lista demorou 1.237 segundo e a pelo Arranjo demorou 1.146 segundo

b) Para o correto funcionamento das classes, seria necessário modificar o método. Condicionando para que caso o lugar em que será inserido seja depois da metade da quantidade de elementos, usamos a cauda para se percorrer menos dados juntamente com o elemento anterior. Desse modo, ocorrerá uma diminuição do tempo de execução para casos de inserção na ultima metade dos dados.

4-

```
class PilhaSobreLista {

    ListaDuplaCauda L = new ListaDuplaCauda();

    public int Topo;

    public PilhaSobreLista() {
        this.Topo = 0;

    }



    public void Push(Elemento<T> E) throws Exception {
        this.Topo++;
        Elemento<T> a = L.frente;
        int aux = 0;
        if(this.Topo == 1){
          L.insereInicio(E.getChave(), E.getDados());
          System.out.println(this.Topo);

        }
        else{
            while(aux < this.Topo - 2){
             a = a.getProximo();
             aux++;
            }
            L.insereApos(a , E.getChave(), E.getDados());
```

```
        }
    }

    public Elemento Pop() throws Exception {
        if (this.empty()) {
            throw new Exception("Pilha vazia");
        } else {
            return L.removeFim();

        }
    }

    public Elemento Top(){
        return L.frente;

    }

    public boolean empty(){
        if(L.frente == null)
            return true;
        else
            return false;
    }

     public void Imprime(){
         Elemento<T> atual = L.frente;
        while (atual != null) {
             System.out.print(atual);
             System.out.print("| ");
             atual = atual.getProximo();
         }
         System.out.println("");
     }


}
```

5-

```
 class FilaSobreLista {

    ListaDuplaCauda L = new ListaDuplaCauda();

    public FilaSobreLista() {

    }

    public boolean QueueEmpty() {
        if (L.frente == L.fundo) {
            return true;
        } else {
```

```
            return false;
        }
    }

    public boolean QueueFull() {
        if (L.frente != null && L.fundo !=null) {
            return true;
        } else {
            return false;
        }
    }

    public void Enqueue(Elemento<T> e) throws Exception {

            L.insereFim(e.getChave(), e.getDados());

    }

   public Elemento Dequeue() throws Exception {
        if (this.QueueEmpty()) {
            throw new Exception("Fila vazia");
        }

        return L.removeInicio();
    }

    public void Imprime() {
        System.out.println("==========");
        Elemento<T> atual = L.frente;
        while (atual != null) {
            System.out.print(atual);
            System.out.print("| ");
            atual = atual.getProximo();
        }
        System.out.println("");
    }
}
```

6- class ListaDupla {

No inicio;

No fim;

int tamanho;

public void inserirInicio(String info) {

No no = new No();

```
      no.info = info;
      no.proximo = inicio;
      if(inicio != null) {
         inicio.anterior = no;
      }
      inicio = no;
      if (tamanho == 0){
          fim = inicio;
      }
      tamanho++;
   }

   public String retirarInicio(){
       if(inicio == null){
          return null;
       }
       String out = inicio.info;
       inicio = inicio.proximo;
       if(inicio != null) {
          inicio.anterior = null;
       }
   }
   tamanho--;
   return out;
   System.out.println("tamanho:");
}
```

7-

A) Para inverter a ordem dos termos de uma lista basta fazer o head apontar para o último elemento da lista, inverter o sentido dos ponteiros, fazer o primeiro elemento apontar para o null e o tail apontar para o primeiro elemento. Por exemplo a lista (head > 1 > 2 > 3 > 4 > 5 > null) onde o 1 é a head e o 6 é o

tail que ao inverter fica (null < 1 < 2 < 3 < 4 < 5 < head) onde o 6 é a head e o 1 é o tail.

B) Como na lista duplamente encadeada tem ponteiros apontando tanto para frente quanto para trás e tem elemento nulo tanto no início quanto no fim, basta fazer o head apontar para o último elemento e o tail apontar para o primeiro.

8-

```
ALGORITIMO
metodo tamanho(){
    var atual <- getInicio();
    var tamanho <- 0;
    se (getInicio() = null){
        retorna tamanho;
    }
 se não{
        tamanho <- 1;
        enquanto (atual.getProximo() != null){
            tamanho <- tamanho + 1;
               atual <- atual.getProximo();
        }
        retorne tamanho;
    }
}
```

Outra opção seria adicionar um atributo "tamanho = 0" à classe "ListaSimples" para que sempre que um item for adicionado à lista o tamanho será acrescido em uma unidade e sempre que fosse removido um item o tamanho seria diminuído em uma unidade. A principal vantagem dessa abordagem em relação a criação de um método "tamanho()" é que nesse caso o atributo tamanho é atualizado sem que seja necessário invocar nenhum comando extra. Contudo, para que isto ocorra faz se necessário um poder de processamento maior, visto que a variável é atualizada mesmo que não seja necessário. Problema este, que não ocorre se for usado o método "tamanho()" que é executado somente quando for invocado.

9- Erro: o item não é inserido como segundo item da lista encadeada

10-Considerando que o tamanho dos pacotes é variável: Deve ser feita uma Lista encadeada, pois, tem a vantagem de ter um tamanho variável e novos itens podem ser adicionados, o que aumenta o seu tamanho.

Considerando também que a memória é bastante limitada e que quando a memória restante não permite a adição de um novo pacote, este é descartado: Deve ser utilizada uma Lista Mutável ou Dinâmica, pois, permite que itens sejam inseridos, substituídos ou excluídos durante a existência da lista.