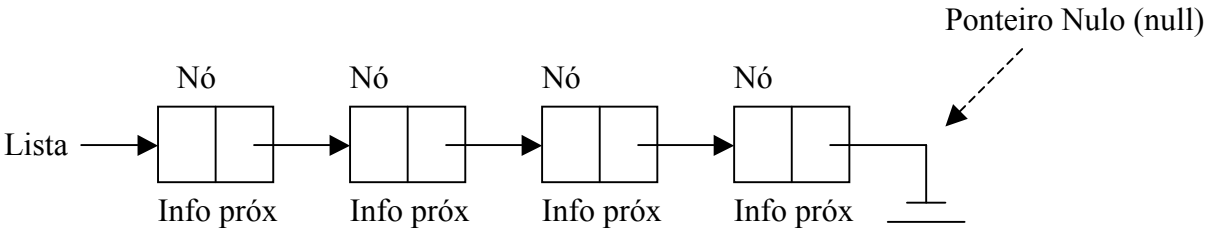
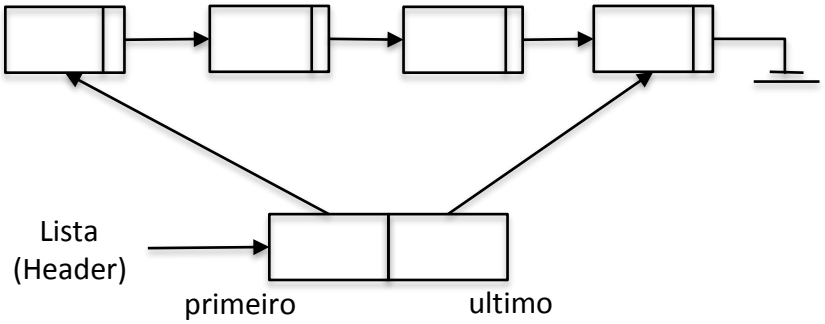
Lista Encadeada

Cada elemento de uma lista encadeada (nó) é constituído por pelo menos dois campos. Um campo contém a informação que é armazenada no nó e o campo próximo que contém o endereço (referência) do próximo elemento na lista. O campo próximo do último elemento da lista contém um valor nulo que denota final da lista. Para que um elemento possa ser acessado em uma lista encadeada, é necessário que o endereço inicial da lista seja conhecido, na representação acima o endereço inicial da lista é representado pela variável *Lista*.

Representação de Lista sem descritor (Header)



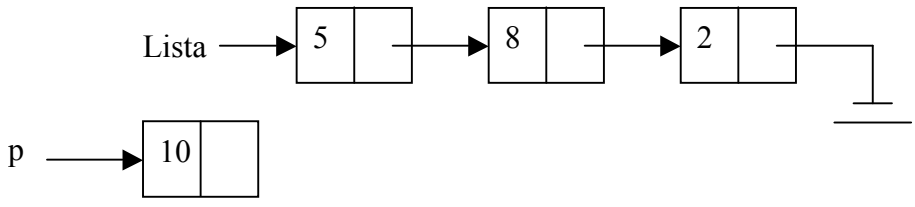
Representação de Lista com descritor (Deque – double ended queue)



Essa representação apresenta maior flexibilidade do que a representação sem descritor (Header), pois permite que operações sejam realizadas tanto no início quanto no final da lista com acesso direto a essas partes da lista. Dessa forma, essa representação pode ser utilizada para se construir outros TADs como Pilha e Fila que manipulam início e fim do encadeamento de forma distinta.

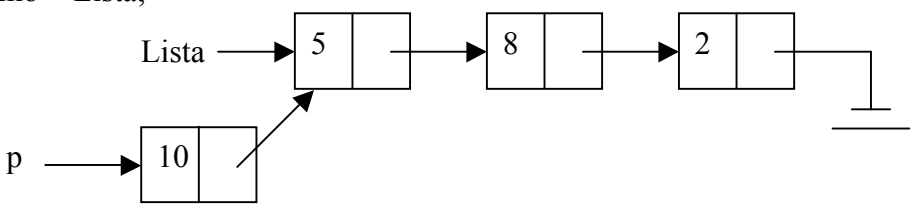
Adicionando um elemento no início da lista

Criar p



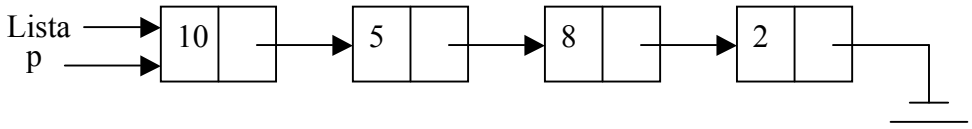
(a)

p.proximo = Lista;

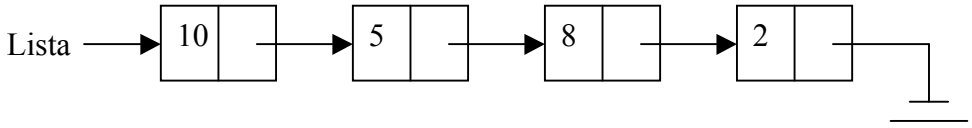


(b)

Lista = p;



(c)



(d)

Exercícios:

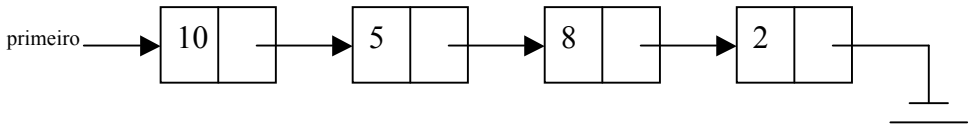
1. Dada uma representação de Lista Simplesmente Encadeada com Descritor (Deque), deduzir a seqüência de instruções necessárias para realizar as seguintes operações:

a) Inserir um elemento como o primeiro da lista (inserePrimeiro)
b) Inserir um elemento depois de um dado elemento de referência *p* (insereDepois)
c) Inserir um elemento como último da lista (insereUltimo)
d) Encontrar um determinado elemento na lista (encontraRef)
e) Determinar se um determinado elemento existe na lista (existe, retornar *true* ou *false*)
f) Remover o primeiro elemento da lista dado o seu valor (removePrimeiro)
g) Remover o elemento posterior a um dado elemento de referência é *p* (removePosterior)
h) Remover um determinado elemento da lista dado o seu valor (removeValor)
i) Remover o ultimo elemento da lista (removeUltimo)

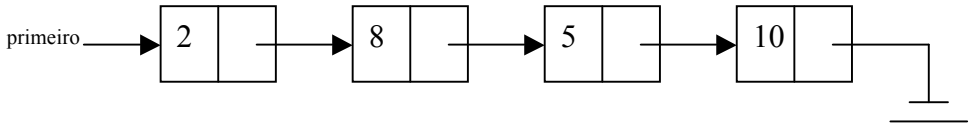
Note que um item do exercício anterior pode necessitar das mesmas instruções de itens anteriores. Isso sugere que cada item do exercício seja desenvolvido em forma de função, procedimentos ou métodos computacionais que podem ser chamados por outras funções, procedimentos ou métodos.

2. Crie um procedimento que inverta uma lista encadeada *Lista*, como ilustrado no exemplo a seguir:

Entrada:



Saída:



Representação Dinâmica de Lista Simplesmente Encadeada

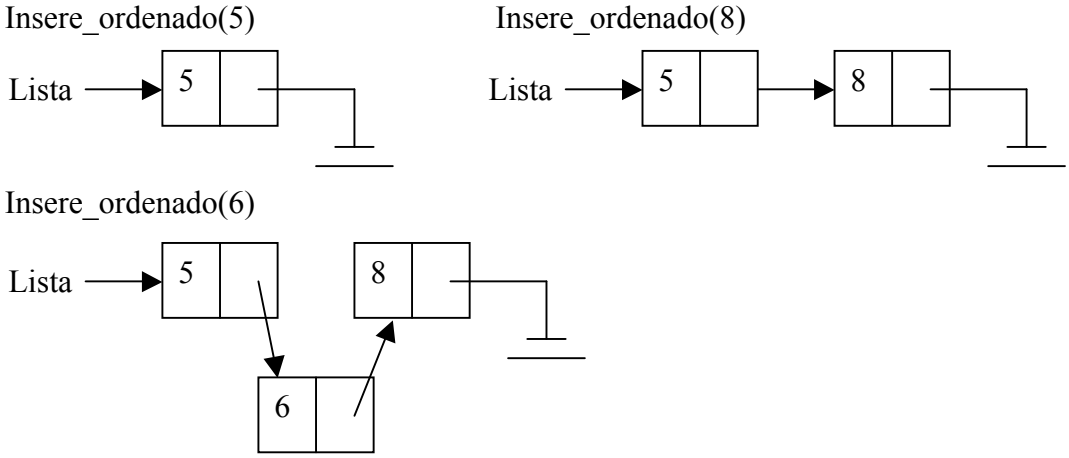
```
Estrutura No{
        Int dado = 0;
```

```
        No proximo = null;
}

Estrutura Lista_Encadeada{
        No Lista = null;
}
```

Fila de Prioridade

Uma lista encadeada pode ser utilizada para representar uma Fila de Prioridade. Quando uma lista utiliza algum critério adicional (por exemplo, ordem ascendente ou descendente do conteúdo dos nós) na implementação das operações de inserção e remoção de elementos, esta lista pode ser chamada de Fila de Prioridade.



Uma fila de prioridade pode ser implementada facilmente quando na implementação de lista encadeada está disponível a função *insereOrdenado*, que deve apresentar tratamento para 4 casos particulares:

- Lista vazia
- Elemento menor que primeiro
- Elemento maior que último
- Elemento é intermediário

Atividade

1) Deduza as instruções necessárias para a implementação da operação *insereOrdenado*.
2) Reutilize os métodos e funções do Exercício 1 (pag. 2) para implementação da operação *insereOrdenado*.

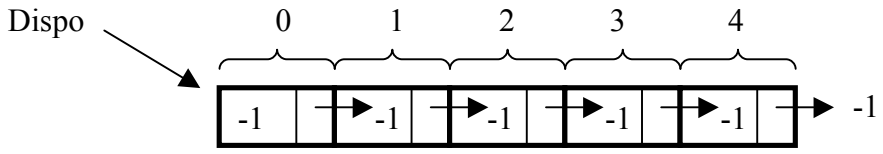
Representação Estática (em Array) de Lista Simplesmente Encadeada

```
Estrutura No {
        Int dado;
        Int proximo = 0;
}

Estrutura Lista_Encadeada{
        Int Dispo = 0;          // Posição do Primeiro Item da lista de Disponíveis
        No[] Lista = No[NumElementos];    // Cria lista de Disponíveis
```

```
        Int primeiro = -1;
}
```

Inicialização da Lista de disponíveis



Operações de alocação de memória ficam sob a responsabilidade do programador.

```
int aloca_no(){
        if (dispo == -1)
                return -1;
        else
                int aux = dispo;
                dispo = Lista[dispo].proximo;
                Lista[aux].proximo = -1;
                return aux;

}

boolean desaloca_no(int p){
        if (p == -1)
                return false;
        else
                Lista[p].proximo = dispo;
                dispo = p;
                return true;

}
```

Operação de Inserção de um elemento na lista na posição posterior a um elemento *p*

```
boolean insere_posterior(int p, int info){
  if (dispo == -1)
        return false;
  Lista[dispo].dado  = info;
  Lista[p].proximo = dispo;
  dispo = Lista[dispo]. Proximo;
  return true;
}
```

Operação de remoção de um elemento na lista na posição posterior a um elemento *p*

```
int remove_posterior(int p) {
  int q, r;
```

```
  if ((p == -1) || (Lista[p].proximo == -1))          // p está na lista de disponíveis ou só há o elemento p na lista
        return –1;                  // Remoção não pode ser realizada
  q = Lista[p].proximo;
  Lista[p].proximo = Lista[q].proximo;
  Lista[q].proximo = Dispo;
  Dispo = q;
  r =  Lista[q].dado;
  Lista[q].dado = -1;
  return r;
}
```

Comparação entre as Representações Dinâmica e Estática de TADs

Em geral, um TAD pode ser implementado considerando alocação estática ou dinâmica de memória. Essa escolha deve ser feita pelo desenvolvedor dependendo das características da aplicação que se está desenvolvendo. O quadro a seguir apresenta uma comparação geral entre implementações de TADs estáticos e dinâmicos.

Representação Estática (Array)	Representação Dinâmica
Operações para manipulação de elementos, geralmente são mais complexas, exigindo algumas estratégias e recursos adicionais extras para gerenciamento de espaço dentro do array.	As operações sobre elementos são, geralmente, muito simples.
Sempre há um determinado número de elementos alocados em memória, mesmo quando nem todos esses elementos estejam sendo utilizados.	O número de elementos é otimizado, ou seja, elementos são criados de acordo com a necessidade do programa.
Requer pelo menos uma estimativa do número de elementos a serem criados.	Não é necessário estimar o número de elementos.
O tempo necessário para execução de operações é geralmente menor, pois há apenas uma alocação de espaço de dados.	Operações de inserção e remoção envolvem métodos de alocação de memória que apresentam um custo computacional considerável, pois há alocação e desalocação em cada uma dessas operações.
O número máximo de elementos na lista é limitado ao tamanho da array utilizada.	O número máximo de elementos está limitado à quantidade de memória disponível na máquina.
Na representação em array o acesso a um elemento consiste em calcular o endereço a partir de um endereço base e do índice do elemento na array.	Dada a referência de um elemento, o elemento é acessado diretamente sem a necessidade de cálculos de endereço.

Aula Prática

Implementar em JAVA, C ou C++ uma lista Encadeada de inteiros com representação dinâmica que disponibilize as seguintes operações:

```
boolean vazia(); // Verifica se a lista L está vazia
void inserePrimeiro(int info);  // Insere o elemento info como primeiro na lista L
void insereDepois(Node No, int info);  // Insere o elemento info depois do nó No
void insereUltimo(int info);   // Insere o elemento info como último na lista L
void insereOrdenado(int info); // Insere o elemento info de maneira ordenada na lista
```

```
void mostraLista();          // Mostra em Tela os Elementos da Lista L
int retiraPrimeiro();        // Retira o primeiro elemento da Lista e retorna o valor da info do No
int retiraUltimo();          // Retira o último elemento da Lista e retorna a informação do Nó
int retiradepois(Node No);   // Retira o elemento posterior ao nó No e retorna a sua informação
int ultimoElemento();        // Retorna a informação do último elemento da Lista
```